Chapter 1: Introduction

- Pseudocode
- Abstract data type
- Algorithm efficiency
Pseudocode

• What is an algorithm?
Pseudocode

• What is an algorithm?
  – The logical steps to solve a problem.
Pseudocode

• What is a program?
  – Program = Data structures + Algorithms (Niklaus Wirth)
Pseudocode

- The most common tool to define algorithms.
- English-like representation of the code required for an algorithm.
Pseudocode

• Pseudocode = **English** + **Code**

  relaxed syntax being easy to read
  instructions using basic control structures
  (sequential, conditional, iterative)

Cao Hoang Tru
CSE Faculty - HCMUT

10 September 2008
Pseudocode

Algorithm Header

Algorithm Body
Pseudocode

• Algorithm Header:
  – Name
  – Parameters and their types
  – Purpose
    • what the algorithm does
  – Precondition
    • precursor requirements for the parameters
  – Postcondition
    • taken action and status of the parameters
  – Return condition
    • returned value
Pseudocode

• Algorithm Body:
  – Statements
  – Statement numbers
    • decimal notation to express levels
  – Variables
    • important data
  – Algorithm analysis
    • comments to explain salient points
  – Statement constructs
    • sequence, selection, iteration
Example

Algorithm average

Pre nothing

Post numbers read and their average printed

1 \( i = 0 \)
2 loop (all data not read)
   1 \( i = i + 1 \)
   2 read number
   3 \( \text{sum} = \text{sum} + \text{number} \)
3 \( \text{average} = \text{sum} / i \)
4 print average
5 return

End average
Algorithm Design

- Divide-and-conquer
- Top-down design
- Abstraction of instructions
- Step-wise refinement
Abstract Data Type

• What is a data type?
  – Class of data objects that have the same properties
Abstract Data Type

• Development of programming concepts:
  – **GOTO programming**
    • control flow is like spaghetti on a plate
  – **Modular programming**
    • programs organized into subprograms
  – **Structured programming**
    • structured control statements (sequence, selection, iteration)
  – **Object-oriented programming**
    • encapsulation of data and operations
Abstract Data Type

• ADT = Data structures + Operations
Abstract Data Type

User knows **what** a data type **can do**.

**How it is done is hidden.**
Abstract Data Type

data structure

function A

internal function

function B

data
Example: Variable Access

- **Rectangle: r**
  - length: x
  - width: y

- **Rectangle: r**
  - length: x (hidden)
  - width: y (hidden)
  - `get_length()`
  - `get_width()`
Example: List

• **Interface:**
  - **Data:**
    - sequence of components of a particular data type
  - **Operations:**
    - accessing
    - insertion
    - deletion

• **Implementation:**
  - **Array,** or
  - **Linked list**
Algorithm Efficiency

- How fast an algorithm is?
- How much memory does it cost?
- **Computational complexity**: measure of the difficulty degree (time or space) of an algorithm.
Algorithm Efficiency

• General format:

\[ f(n) \]

\( n \) is the size of a problem (the key number that determines the size of input data)
Linear Loops

1 \hspace{1cm} i = 1
2 \hspace{1cm} loop (i <= 1000)  
   1 \hspace{1cm} application code
   2 \hspace{1cm} i = i + 1

The number of times the body of the loop is replicated is 1000

1 \hspace{1cm} i = 1
2 \hspace{1cm} loop (i <= 1000)  
   1 \hspace{1cm} application code
   2 \hspace{1cm} i = i + 2

The number of times the body of the loop is replicated is 500
Linear Loops

\[ f(n) = n \cdot T \]
\[ f(n) = (n/2) \cdot T \]
Logarithmic Loops

Multiply loops
1   \( i = 1 \)
2   loop (\( i \leq 1000 \))
    1   application code
    2   \( i = i \times 2 \)

The number of times the body of the loop is replicated is \( \log_2 n \)
Logarithmic Loops

Multiply loops
1. $i = 1$
2. loop ($i \leq 1000$)
   1. application code
   2. $i = i \times 2$

Divide loops
1. $i = 1000$
2. loop ($i \geq 1$)
   1. application code
   2. $i = i \div 2$

The number of times the body of the loop is replicated is $\log_2 n$
Logarithmic Loops

\[ f(n) = (\log_2 n) \cdot T \]
Nested Loops

Iterations = Outer loop iterations \times Inner loop iterations
Linear Logarithmic Loops

\begin{verbatim}
1 i = 1
2 loop (i <= 10)
   1 j = 1
   2 loop (j <= 10)
      1 application code
      2 j = j × 2
   3 i = i + 1
\end{verbatim}

The number of times the body of the loop is replicated is $n \log_2 n$
Linear Logarithmic Loops

\[ f(n) = (n \log_2 n) \cdot T \]
Quadratic Loops

1  i = 1
2  loop (i <= 10)
   1  j = 1
   2  loop (j <= 10)
      1  application code
      2  j = j + 1
   3  i = i + 1

The number of times the body of the loop is replicated is $n^2$
Dependent Quadratic Loops

\begin{verbatim}
1  i = 1
2  loop (i <= 10)
   1  j = 1
   2  loop (j <= i)
      1  application code
      2  j = j + 1
   3  i = i + 1

The number of times the body of the loop is replicated is
1 + 2 + \ldots + n = \frac{n(n + 1)}{2}
\end{verbatim}
Quadratic Loops

\[ f(n) = n^2 \cdot T \]
Asymptotic Complexity

• Algorithm efficiency is considered with only big problem sizes.

• We are not concerned with an exact measurement of an algorithm's efficiency.

• Terms that do not substantially change the function’s magnitude are eliminated.
Big-O Notation

- $f(n) = c \cdot n \Rightarrow f(n) = O(n)$.
- $f(n) = n(n + 1)/2 = n^2/2 + n/2 \Rightarrow f(n) = O(n^2)$. 
Big-O Notation

• Set the coefficient of the term to one.

• Keep the largest term and discard the others.

\[ \log_2 n \quad n \quad n \log_2 n \quad n^2 \quad n^3 \quad \ldots \quad n^k \quad \ldots \quad 2^n \quad n! \]
# Standard Measures of Efficiency

<table>
<thead>
<tr>
<th>Efficiency</th>
<th>Big-O</th>
<th>Iterations</th>
<th>Est. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>logarithmic</td>
<td>$O(\log_2 n)$</td>
<td>14</td>
<td>microseconds</td>
</tr>
<tr>
<td>linear</td>
<td>$O(n)$</td>
<td>10,000</td>
<td>.1 seconds</td>
</tr>
<tr>
<td>linear logarithmic</td>
<td>$O(n \log_2 n)$</td>
<td>140,000</td>
<td>2 seconds</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(n^2)$</td>
<td>$10,000^2$</td>
<td>15-20 min.</td>
</tr>
<tr>
<td>polynomial</td>
<td>$O(n^k)$</td>
<td>$10,000^k$</td>
<td>hours</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^n)$</td>
<td>$2^{10,000}$</td>
<td>intractable</td>
</tr>
<tr>
<td>factorial</td>
<td>$O(n!)$</td>
<td>10,000!</td>
<td>intractable</td>
</tr>
</tbody>
</table>

Assume instruction speed of 1 microsecond and 10 instructions in loop.

$\ n = 10,000$
Standard Measures of Efficiency

$O(n)$

$n^2$, $n\log_2 n$, $n$, $\log_2 n$
Big-O Analysis Examples

Algorithm addMatrix (val matrix1 <matrix>, val matrix2 <matrix>,
val size <integer>, ref matrix3 <matrix>)
Add matrix1 to matrix2 and place results in matrix3
Pre matrix1 and matrix2 have data
size is number of columns and rows in matrix
Post matrices added - result in matrix3
1  r = 1
2  loop (r <= size)
   1  c = 1
   2  loop (c <= size)
      1  matrix3[r, c] = matrix1[r, c] + matrix2[r, c]
      2  c = c + 1
   3  r = r + 1
3  return
End addMatrix
Big-O Analysis Examples

Algorithm addMatrix (val matrix1 <matrix>, val matrix2 <matrix>,
                    val size <integer>, ref matrix3 <matrix>)
Add matrix1 to matrix2 and place results in matrix3
Pre  matrix1 and matrix2 have data
     size is number of columns and rows in matrix
Post  matrices added - result in matrix3

1  r = 1
2  loop (r <= size)
   1  c = 1
   2  loop (c <= size)
      1  matrix3[r, c] = matrix1[r, c] + matrix2[r, c]
      2  c = c + 1
   3  r = r + 1
3  return
End  addMatrix

Nested linear loop: f(size) = O(size^2)

Cao Hoang Tru
CSE Faculty - HCMUT
10 September 2008
Time Costing Operations

• The most time consuming: data movement to/from memory/storage.

• Operations under consideration:
  – Comparisons
  – Arithmetic operations
  – Assignments
Recurrence Equation

• An equation or inequality that describes a function in terms of its value on smaller input.
Recurrence Equation

• Example: binary search.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>14</td>
<td>21</td>
<td>22</td>
<td>36</td>
<td>62</td>
<td>77</td>
<td>81</td>
<td>91</td>
</tr>
</tbody>
</table>
Recurrence Equation

- Example: binary search.

\[
f(n) = 1 + f(n/2) \Rightarrow f(n) = O(\log_2 n)
\]
Best, Average, Worst Cases

- **Best case**: when the number of steps is smallest.

- **Worst case**: when the number of steps is largest.

- **Average case**: in between.
Best, Average, Worst Cases

• Example: sequential search.

Best case: $f(n) = O(1)$
Worst case: $f(n) = O(n)$
Best, Average, Worst Cases

• Example: **sequential search.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>21</td>
<td>14</td>
<td>22</td>
<td>36</td>
<td>62</td>
<td>91</td>
<td>77</td>
<td>81</td>
</tr>
</tbody>
</table>

Average case: $f(n) = \sum i \cdot p_i$

$p_i$: probability for the target being at $a[i]$

$p_i = 1/n \Rightarrow f(n) = (\sum i)/n = O(n)$
P and NP Problems

• P: Polynomial (can be solved in polynomial time on a deterministic machine).

• NP: Nondeterministic Polynomial (can be solved in polynomial time on a non-deterministic machine).
P and NP Problems

Travelling Salesman Problem:
A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.
P and NP Problems

Travelling Salesman Problem:

Deterministic machine: \( f(n) = n(n-1)(n-2) \ldots 1 = O(n!) \)

\( \Rightarrow \) NP problem
P and NP Problems

• **NP-complete**: NP and every other problem in NP is polynomially reducible to it.

• Open question: $P = NP$?