Design and Analysis of Algorithms
演算法設計與分析

Lecture 1
February 25, 2010
洪國寶
Outline

- Course information
- Motivation
- Outline of the course
- Introduction to design, analysis, algorithms
Course information (1/5)

• Instructor: Professor Gwoboa Horng
• Basic assumption
  It is assumed that students in this course have had some exposure to algorithms and their analysis.
Course information (2/5)

• **Textbook**

• **Solutions to exercises and problems:**

• **Course web page:**
  [http://140.120.14.97/course.htm](http://140.120.14.97/course.htm)
  Password: *ailab*
Course information (3/5)

• The **goal** of this class is to present fundamental problem-solving techniques, for designing efficient computer algorithms, proving their correctness, and analyzing their performance.
Course information (4/5)

• This class is
  – Not a lab or programming course
  – Not a math course, either
Course information (5/5)

• Grading (Tentative)

Homework 25%

(You may collaborate when solving the homework, however when writing up the solutions you must do so on your own. Handwritten only.)

Midterm exam 30% (Open book and notes)
Final exam 30% (Open book and notes)
Class participation 15%
Why you need to take this course?

- You are preparing your Ph.D. exam
- Your advisor asks you to
- You love theoretical stuff
- You have nothing to do
Why you need to drop this course?

• A lot of homework
• Low grade
Outline

• Course information
• **Motivation**
• Outline of the course
• Introduction to design, analysis, algorithms
Motivation (1/6)

• Why study algorithms?
  Computers are used to solve problems.
  
  **Now**: Computer = Tool = Passive device
  Need to tell it **how to do**
  → Algorithms

  **Future**: Computer = Assistant = Active device
  Need to tell it **what to do**
  → Artificial Intelligence
Motivation (2/6)

• Problems: concrete vs. **abstract**
  – The human genome project
  – Internet related problem (Search engine etc)
  – Electronic commerce (Cryptography etc)
  – Manufacturing and other commercial settings (scheduling)

• Will focus on abstract problems
Motivation (3/6)

• Why abstract problems?
  – One algorithm many applications
    Matrix chain multiplication, optimal polygon triangulation
    Optimal merge pattern, Huffman code
  – Applications to be found later (kill time)
    FFT (signal analysis)
    Cutting rectangular (VLSI)
Motivation (4/6)

• From a research point of view, what do they imply?

RE-SEARCH

李家同教授

重視基礎科學 也做無目的的學問

【2000/07/02/聯合報 民意論壇 】
Motivation (5/6)

• **The importance of efficient algorithms:**
  - Deep Blue (Chess)
  - Factorization (RSA)

<table>
<thead>
<tr>
<th>Number of Decimal Digits</th>
<th>Approximate Number of Bits</th>
<th>Date Achieved</th>
<th>MIPS-years</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>332</td>
<td>April 1991</td>
<td>7</td>
<td>quadratic sieve</td>
</tr>
<tr>
<td>110</td>
<td>365</td>
<td>April 1992</td>
<td>75</td>
<td>quadratic sieve</td>
</tr>
<tr>
<td>120</td>
<td>398</td>
<td>June 1993</td>
<td>830</td>
<td>quadratic sieve</td>
</tr>
<tr>
<td>129</td>
<td>428</td>
<td>April 1994</td>
<td>5000</td>
<td>quadratic sieve</td>
</tr>
<tr>
<td>130</td>
<td>431</td>
<td>April 1996</td>
<td>1000</td>
<td>generalized number field sieve</td>
</tr>
<tr>
<td>140</td>
<td>465</td>
<td>February 1999</td>
<td>2000</td>
<td>generalized number field sieve</td>
</tr>
<tr>
<td>155</td>
<td>512</td>
<td>August 1999</td>
<td>8000</td>
<td>generalized number field sieve</td>
</tr>
</tbody>
</table>

Current record: **Factorization of a 768-bit RSA modulus**
Motivation (6/6)

• **The importance of efficient algorithms**
  
  Total system performance depends on choosing *efficient algorithms* as much as on choosing *fast hardware*.

  A faster algorithm running on a slower computer will always win for sufficiently large instances.
Outline

- Course information
- Motivation
- **Outline of the course**
- Introduction to design, analysis, algorithms
Outline of the course (1/1)

• Introduction (1-4)
• Data structures (10-14)
• Dynamic programming (15)
• Greedy methods (16)
• Amortized analysis (17)
• Advanced data structures (6, 19-21)
• Graph algorithms (22-25)
• NP-completeness (34-35)
• Other topics (5, 27, 31)
Outline

• Course information
• Motivation
• Outline of the course
• Introduction to design, analysis, algorithms
Introduction to Design, Analysis, Algorithms

• Introduction to Algorithms
• Introduction to Analysis
• Introduction to Design (next lecture)
Introduction to Algorithms (1/)

• Definition of algorithms

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. (p. 5)
Introduction to Algorithms (2/)

Properties of algorithms:

- **Input.** An algorithm has input values from a specified set.
- **Output.** For each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem.
- **Definiteness.** The steps of an algorithm must be defined precisely.
- **Correctness.** An algorithm should produce the correct output values for each set of the input values.
- **Finiteness.** An algorithm should produce the desired output values after a finite number of steps for any input in the set.
- **Effectiveness.** It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- **Generality.** The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.
Introduction to Algorithms (3/)

- We can also view an algorithm as a tool for solving a well-specified computational problem. (p.5)

- A program is an expression of an algorithm in a programming language.
Introduction to Algorithms (4/)

• Example of a well-specified computational problem: sorting problem (p. 5)

*Input:* sequence \( \langle a_1, a_2, \ldots, a_n \rangle \) of numbers.

*Output:* permutation \( \langle a'_1, a'_2, \ldots, a'_n \rangle \) such that \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \).
Introduction to Algorithms (5/)

• An instance of a sorting problem:

  Input: 31, 26, 52, 41, 58, 55
  Output: 26, 31, 41, 52, 55, 58
Introduction to Algorithms (6/)

• A simple “algorithm” for sorting
  
  ```
  i=1
  while i < n
    swap $a_i$ and $a_{i+1}$
  i=i+2
  ```

• Does it produce correct output for the previous instance?

• Is it a correct algorithm for the sorting problem?
Introduction to Algorithms (7/)

• An algorithm is said to be **correct** if for every input instance, it halts with correct output.

• **Correctness is not obvious.** (see slides 32 – 38)
Introduction to Algorithms (8/)

• How to describe algorithms?
  – Flow chart
  – Pseudo-code like Pascal
Introduction to Algorithms (9/)

- **Pseudo-code conventions** (pp. 20 - 22)
  - *Indentation* indicates block structure.
  - The looping constructs **while**, **for**, and **repeat** and the conditional constructs **if**, **then**, and **else** have the same interpretation as in Pascal.
  - The symbol “#” indicates that the remainder of the line is a comment.
  - A *multiple assignment* of the form \( i \leftarrow j \leftarrow e \) assigns to both variables \( i \) and \( j \) the value of expression \( e \); it should be treated as equivalent to the assignment \( j \leftarrow e \) followed by the assignment \( i \leftarrow j \).
Introduction to Algorithms (10/)

• Pseudo-code conventions (pp. 20 - 22)
  – **Variables** (such as $i, j,$ and $key$) are local to the given procedure. We shall not use global variables without explicit indication.
  – **Array elements** are accessed by specifying the array name followed by the index in square brackets.
  – **Compound data** are typically organized into **objects**, which are comprised of **attributes or fields**. A particular field is accessed using the field name followed by the name of its object in **square brackets**.
  – **Parameters** are passed to a procedure **by value**: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is not seen by the calling routine. When objects are passed, the pointer to the data representing the object is copied, but the object's fields are not.
  – The **boolean operators** **and** and **or** are short circuiting.
Introduction to **Algorithms** (11/)

- More about sorting

Why don’t CS professors ever stop talking about sorting?
Introduction to Algorithms (12/)

- Correctness is not obvious (1/7)

The following problem arises often in manufacturing and transportation testing applications.

Suppose you have a robot arm equipped with a tool, say a soldering iron. To enable the robot arm to do a soldering job, we must construct an ordering of the contact points, so the robot visits (and solders) the first contact point, then visits the second point, third, and so forth until the job is done.

Since robots are expensive, we need to find the order which minimizes the time (i.e., travel distance) it takes to assemble the circuit board.

You are given the job to program the robot arm. Give me an algorithm to find the best tour!
Introduction to Algorithms (13/)

- Correctness is not obvious (2/7) (nearest neighbor)

A very popular solution starts at some point $p_0$ and then walks to its nearest neighbor $p_1$ first, then repeats from $p_1$, etc. until done.

Pick and visit an initial point $p_0$

$p = p_0$

$i = 0$

While there are still unvisited points

$i = i + 1$

Let $p_i$ be the closest unvisited point to $p_{i-1}$

Visit $p_i$

Return to $p_0$ from $p_i$
Introduction to Algorithms (14/)

• Correctness is not obvious (3/7)

Always starting from the leftmost point or any other point will not fix the problem.
Introduction to Algorithms (15/)

- Correctness is not obvious (4/7) (closest pair)

Always walking to the closest point is too restrictive, since that point might trap us into making moves we don't want.

Another idea would be to repeatedly connect the closest pair of points whose connection will not cause a cycle or a three-way branch to be formed, until we have a single chain with all the points in it.

Let \( n \) be the number of points in the set
\[
d = \infty
\]
For \( i = 1 \) to \( n - 1 \) do
  For each pair of endpoints \((x, y)\) of partial paths
    If \( \text{dist}(x, y) \leq d \) then
      \[
      x_m = x, \quad y_m = y, \quad d = \text{dist}(x, y)
      \]
      Connect \((x_m, y_m)\) by an edge
    Connect the two endpoints by an edge.
Introduction to Algorithms (15/)

- Correctness is not obvious (5/7)

Although it works correctly on the previous example, other data causes trouble:

This algorithm is not correct!
Introduction to Algorithms (16/)

- Correctness is not obvious (6/7) A correct algorithm

We could try all possible orderings of the points, then select the ordering which minimizes the total length:

\[ d = \infty \]

For each of the \( n! \) permutations \( \Pi_i \) of the \( n \) points

\[ \text{If } (\text{cost}(\Pi_i) \leq d) \text{ then} \]

\[ d = \text{cost}(\Pi_i) \text{ and } P_{\min} = \Pi_i \]

Return \( P_{\min} \)
Introduction to Algorithms (17/)

- Correctness is not obvious (7/7)

Since all possible orderings are considered, we are guaranteed to end up with the shortest possible tour.

Because it tries all $n!$ permutations, it is extremely slow, much too slow to use when there are more than 10-20 points.

No efficient, correct algorithm exists for the traveling salesman problem, as we will see later.

To show an algorithm is incorrect is much easier:
Just give an instance that the algorithm produces incorrect output. (consider previous sorting “algorithm”)
Introduction to **Algorithms** (18/)

- A correct sorting algorithm: insertion sort

*Figure 2.1  Sorting a hand of cards using insertion sort.*
Introduction to Algorithms (19/)

• To sort \( n \) elements:
  – Return if \( n = 1 \)
  – Sort the first \( n-1 \) elements
    Insert the \( n \)-th element in the right place

( insert \( a_n \) after \( a’_{n-1} \) if \( a’_{n-1} < a_n \); or
insert \( a_n \) between \( a’_i \) and \( a’_{i+1} \) if \( a’_i < a_n \leq a’_{i+1} \); or
insert \( a_n \) before \( a’_1 \) if \( a_n < a’_1 \))
Introduction to Algorithms (20/)

- To sort \( n \) elements:
  - Return if \( n = 1 \)
  - Sort the first \( n-1 \) elements
    Insert the \( n \)-th element in the right place

**Insertion-sort(A,n)**

  if \( n=1 \) then return \( A[n] \)
  else Insertion-sort(A,n-1)
    Insert(A, n-1, A[n])
Introduction to Algorithms (21/)

Insert(A, n-1, A[n])

Insert the n-th element in the right place

2 3 4 8 9 6

Insert(A, j, key)

i = j;
while i > 0 and A[i] > key
A[i+1] = A[i]
    i = i - 1
A[i+1] = key

n=6
A[n]=6
j=n-1=5
key=6
Introduction to Algorithms (22/)

Insertion sort (recursive version)

Insertion-sort(A,n)
  if n=1 then return A[n]
  else Insertion-sort(A,n-1)
    Insert(A, n-1, A[n])

Insert(A, j, key)
  i = j;
  while i > 0 and A[i] > key
    A[i+1] = A[i]
    i = i - 1
  A[i+1] = key
Running \textbf{Insertion-sort}(A,n)

\begin{align*}
\text{Insertion-sort}(A,n) & \\
\text{Insertion-sort}(A,n-1) & \\
\text{Insertion-sort}(A,n-2) & \\
\text{Insertion-sort}(A,4) & \\
\text{Insertion-sort}(A,3) & \\
\text{Insertion-sort}(A,2) & \\
\text{Insertion-sort}(A,1) & \\
\text{return } A[1] & \\
\text{Insert}(A, 1, A[2]) & \\
\text{Insert}(A, 2, A[3]) & \\
\text{Insert}(A, 3, A[4]) & \\
\text{Insert}(A, n-2, A[n-1]) & \\
\text{Insert}(A, n-1, A[n]) & \\
\text{Insert}(A, n-1, A[n]) & \\
\end{align*}
Running $\text{Insertion-sort}(A,n)$

\begin{align*}
\text{return } A[1] \\
\text{Insert}(A, 1, A[2]) \\
\text{Insert}(A, 2, A[3]) \\
\quad \ldots \\
\text{Insert}(A, n-3, A[n-2]) \\
\text{Insert}(A, n-2, A[n-1]) \\
\text{Insert}(A, n-1, A[n])
\end{align*}

\begin{align*}
\text{Insert-Sort}(A) \\
1 & \text{ for } j \leftarrow 2 \text{ to } \text{length}[A] \\
2 & \quad \text{do } key \leftarrow A[j] \\
3 & \quad \quad > \text{ Insert } A[j] \text{ into the sorted sequence } A[1..j-1]. \\
4 & \quad \quad i \leftarrow j - 1 \\
5 & \quad \quad \text{while } i > 0 \text{ and } A[i] > key \\
6 & \quad \quad \quad \text{do } A[i+1] \leftarrow A[i] \\
7 & \quad \quad \quad i \leftarrow i - 1 \\
8 & \quad \quad \quad A[i+1] \leftarrow key
\end{align*}

From recursion to iteration
Insertion sort (iterative version)

\[
\text{INSERTION-SORT}(A)
\]
\[
1 \quad \text{for } j \leftarrow 2 \text{ to } \text{length}[A]
\]
\[
2 \quad \text{do } key \leftarrow A[j]
\]
\[
3 \quad \triangleright \text{Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j - 1].
\]
\[
4 \quad i \leftarrow j - 1
\]
\[
5 \quad \text{while } i > 0 \text{ and } A[i] > key
\]
\[
6 \quad \text{do } A[i + 1] \leftarrow A[i]
\]
\[
7 \quad i \leftarrow i - 1
\]
\[
8 \quad A[i + 1] \leftarrow key
\]
Think twice

• We have explained
  – problems, instances
  – algorithms
  – recursion, iteration
  – correctness

• If you have problems understanding the above discussion, please consider drop this course.
Example of insertion sort

\[
\begin{array}{cccccc}
8 & 2 & 4 & 9 & 3 & 6
\end{array}
\]

\[j=2\]

**Insertion-Sort**(\(A\))

1. for \(j \leftarrow 2\) to \(\text{length}[A]\)
2. \hspace{1em} do \(key \leftarrow A[j]\)
3. \hspace{2em} \triangleright \text{Insert } A[j] \text{ into the sorted sequence } A[1 \ldots j - 1].
4. \hspace{2em} i \leftarrow j - 1
5. \hspace{2em} while \(i > 0\) and \(A[i] > key\)
6. \hspace{3em} do \(A[i + 1] \leftarrow A[i]\)
7. \hspace{3em} \hspace{1em} i \leftarrow i - 1
8. \hspace{2em} A[i + 1] \leftarrow key
Example of insertion sort

\[8 \rightarrow 2 \rightarrow 4 \rightarrow 9 \rightarrow 3 \rightarrow 6\]

**INSERTION-SORT**(A)
1. for \( j \leftarrow 2 \) to \( \text{length}[A] \)
2. do \( \text{key} \leftarrow A[j] \)
3. \( \triangleright \) Insert \( A[j] \) into the sorted sequence \( A[1..j-1] \).
4. \( i \leftarrow j - 1 \)
5. while \( i > 0 \) and \( A[i] > \text{key} \)
6. do \( A[i+1] \leftarrow A[i] \)
7. \( i \leftarrow i - 1 \)
8. \( A[i+1] \leftarrow \text{key} \)
Example of insertion sort

Insertion-Sort($A$)
1. for $j \leftarrow 2$ to $\text{length}[A]$
2. do $key \leftarrow A[j]$
4. $i \leftarrow j - 1$
5. while $i > 0$ and $A[i] > key$
6. do $A[i + 1] \leftarrow A[i]$
7. $i \leftarrow i - 1$
8. $A[i + 1] \leftarrow key$

j=3
Example of insertion sort

\[ 8 \quad 2 \quad 4 \quad 9 \quad 3 \quad 6 \]

\[ 2 \quad 8 \quad 4 \quad 9 \quad 3 \quad 6 \]
Example of insertion sort

8  2  4  9  3  6

2  8  4  9  3  6

2  4  8  9  3  6

j=4
Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
Example of insertion sort

8  2  4  9  3  6
2  8  4  9  3  6
2  4  8  9  3  6
2  4  8  9  3  6

\( j = 5 \)
Example of insertion sort

\[
\begin{array}{cccccc}
8 & 2 & 4 & 9 & 3 & 6 \\
2 & 8 & 4 & 9 & 3 & 6 \\
2 & 4 & 8 & 9 & 3 & 6 \\
2 & 4 & 8 & 9 & 3 & 6 \\
\end{array}
\]
Example of insertion sort

\[
\begin{array}{cccccc}
8 & 2 & 4 & 9 & 3 & 6 \\
2 & 8 & 4 & 9 & 3 & 6 \\
2 & 4 & 8 & 9 & 3 & 6 \\
2 & 4 & 8 & 9 & 3 & 6 \\
2 & 3 & 4 & 8 & 9 & 6 \\
\end{array}
\]

\( j = 6 \)
Example of insertion sort

8  2  4  9  3  6
2  8  4  9  3  6
2  4  8  9  3  6
2  4  8  9  3  6
2  3  4  8  9  6
Example of insertion sort

1. 8 2 4 9 3 6
2. 2 8 4 9 3 6
3. 2 4 8 9 3 6
4. 2 4 8 9 3 6
5. 2 3 4 8 9 6
6. 2 3 4 6 8 9

done
Introduction to Algorithms

• Sorting Algorithm Animations
  http://www.sorting-algorithms.com/
Introduction to Algorithms (37/)

- Correctness of insertion sort
  - By induction
  - By loop invariant
Correctness of insertion sort

• By induction
  Use blackboard.
Loop Invariants

• A Method for Proving the Correctness of Iterative Algorithms
Loop Invariants

• A *loop invariant* states a condition that is true immediately before every iteration of a loop.

• The invariant should be true also immediately after the last iteration.
Loop Invariants

• We have to show:
  – **Initialization:**
    The loop invariant holds before the first iteration
  – **Maintenance:**
    If the loop invariant holds before the i-th iteration, it holds before the (i+1)-st iteration.
  – **Termination:**
    If the loop invariant holds after the last iteration, the algorithm does what we want.
Loop invariants

- **Initialization:**
  The invariant holds here.

- **Maintenance:**
  If the invariant holds here…
  …it holds here.

- **Termination:**
  If the invariant holds here, the algorithm is correct.
A Loop Invariant for Insertion Sort

• **Invariant:** Before the iteration of the outer loop where \( j = k + 1 \), array \( A[1..k] \) stores the \( k \) elements initially in \( A[1..k] \) in sorted order. Array \( A[k+1..n] \) is unaltered.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2     do key ← A[j]
3     > Insert A[j] into the sorted sequence A[1..j-1].
4     i ← j - 1
5     while i > 0 and A[i] > key
6     do A[i+1] ← A[i]
7     i ← i - 1
8     A[i+1] ← key
```
Correctness of insertion sort

• By loop Invariant
Use blackboard.
Introduction to Algorithms (45/45)

• Algorithm animation
Introduction to Analysis (1/)

- **Analysis**: predict the cost of an algorithm in terms of resources and performance
- Resource: *time*, space, bandwidth, random bits, etc.
Introduction to Analysis (2/)

• Why not just use a supercomputer?
Introduction to Analysis (3/)

Purpose of time analysis

• To estimate how long a program will run.
• To estimate the largest input that can reasonably be given to the program.
• To compare the efficiency of different algorithms.
• To help focus on the parts of code that are executed the largest number of times.
• To choose an algorithm for an application.
Introduction to Analysis (4/)

• **Machine Model**
  
  Generic Random Access Machine (RAM)
  
  Executes operations sequentially
  
  Each memory access takes exactly 1 step
  
  Set of primitive operations:
  
  -- Arithmetic, Logical, Comparisons, Function calls
  
  Loop and subroutine calls are not primitive ops.

• **Simplifying assumption: all ops cost 1 unit**
  
  – Eliminates dependence on the speed of our computer, otherwise impossible to verify and to compare
Introduction to Analysis (5/)

• Under the RAM model, the running time of an algorithm depends mainly on **input size**.
• Input size: depends on the problem
  - number of items (sorting problem)
  - number of bits (number theoretic problems)
  - number of edges and vertices (graph problems)
### Introduction to Analysis (6/)

- Exact analysis of insertion sort

```plaintext

**Insertion-Sort**($A$)

1. **for** $j$ $\leftarrow$ 2 to length[$A$]
2. **do** $key$ $\leftarrow$ $A[j]$
4. $i$ $\leftarrow$ $j - 1$
5. **while** $i > 0$ and $A[i]$ $>$ $key$
6. **do** $A[i+1]$ $\leftarrow$ $A[i]$
7. $i$ $\leftarrow$ $i - 1$
8. $A[i+1]$ $\leftarrow$ $key$

<table>
<thead>
<tr>
<th>Step</th>
<th>Cost</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$c_1$</td>
<td>$n$</td>
</tr>
<tr>
<td>2</td>
<td>$c_2$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>4</td>
<td>$c_4$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>5</td>
<td>$c_5$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>6</td>
<td>$c_6$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>7</td>
<td>$c_7$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>8</td>
<td>$c_8$</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>
Introduction to Analysis (7/)

- Running time
  - best case
  - worst case
  - average case
Introduction to Analysis (8/)

- Running time of insertion sort
  - best case: linear time
  - worst case: quadratic
  - average case:
Introduction to Analysis (9/)

• Reasons for worst case analysis
  - upper bound
  - worst case occurs fairly often
  - average case is often roughly as bad as worst case
Introduction to Analysis (10/)

• Notations (for the asymptotic running time)
### Notations

<table>
<thead>
<tr>
<th>Type</th>
<th>Formula</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theta</td>
<td>$f(n) = \theta(g(n))$</td>
<td>$f(n) \approx c \cdot g(n)$</td>
</tr>
<tr>
<td>BigOh</td>
<td>$f(n) = \mathcal{O}(g(n))$</td>
<td>$f(n) \leq c \cdot g(n)$</td>
</tr>
<tr>
<td>Omega</td>
<td>$f(n) = \Omega(g(n))$</td>
<td>$f(n) \geq c \cdot g(n)$</td>
</tr>
<tr>
<td>Little Oh</td>
<td>$f(n) = o(g(n))$</td>
<td>$f(n) \ll c \cdot g(n)$</td>
</tr>
<tr>
<td>Little Omega</td>
<td>$f(n) = \omega(g(n))$</td>
<td>$f(n) \gg c \cdot g(n)$</td>
</tr>
</tbody>
</table>
Upper Bound Notation

• We say InsertionSort’s run time is $O(n^2)$
  – Properly we should say run time is in $O(n^2)$
  – Read $O$ as “Big-O” (you’ll also hear it as “order”)

• In general a function
  – $f(n)$ is $O(g(n))$ if there exist positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

• Formally
  – $O(g(n)) = \{ f(n) : \exists$ positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n) \ \forall \ n \geq n_0$
Insertion Sort Is $O(n^2)$

• Proof
  – Suppose runtime is $an^2 + bn + c$
    • If any of $a$, $b$, and $c$ are less than 0 replace the constant with its absolute value
  – $an^2 + bn + c \leq (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
  – $\leq 3(a + b + c)n^2$ for $n \geq 1$
  – Let $c' = 3(a + b + c)$ and let $n_0 = 1$

• Questions
  – Is InsertionSort $O(n^3)$?
  – Is InsertionSort $O(n)$?
Big O Fact

• A polynomial of degree $k$ is $O(n^k)$

• Proof:
  
  – Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    
    • Let $a_i = |b_i|$
      
    – $f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$
      
      $\leq n^k \sum a_i \frac{n}{n^k} \leq n^k \sum a_i \leq cn^k$
Lower Bound Notation

• We say InsertionSort’s run time is $\Omega(n^2)$

• In general a function
  
  – $f(n)$ is $\Omega(g(n))$ if $\exists$ positive constants $c$ and $n_0$ such that $0 \leq c \cdot g(n) \leq f(n)$ $\forall$ $n \geq n_0$
Asymptotic Tight Bound

• A function $f(n)$ is $\Theta(g(n))$ if $\exists$ positive constants $c_1$, $c_2$, and $n_0$ such that

\[ c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall \quad n \geq n_0 \]

• Theorem
  – $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$
Introduction to Analysis (17/)

• Asymptotic notations
  Figure 3.1
Figure 3.1  Graphic examples of the $\Theta$, $O$, and $\Omega$ notations. In each part, the value of $n_0$ shown is the minimum possible value; any greater value would also work. (a) $\Theta$-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants $n_0$, $c_1$, and $c_2$ such that to the right of $n_0$, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. (b) $O$-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants $n_0$ and $c$ such that to the right of $n_0$, the value of $f(n)$ always lies on or below $cg(n)$. (c) $\Omega$-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants $n_0$ and $c$ such that to the right of $n_0$, the value of $f(n)$ always lies on or above $cg(n)$.
Introduction to Analysis (19/)

• Asymptotic notations
  Examples:
Introduction to Analysis (20/)

• **Asymptotic notation properties: (pp. 48-50)**
  For the following, assume that \( f(n) \) and \( g(n) \) are asymptotically positive.
  
  – **Transitivity:**
    
    \[ f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)), \]
    \[ f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n)), \]
    \[ f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n)), \]
  
  – **Reflexivity:** \( f(n) = \Theta(f(n)) \), \( f(n) = O(f(n)) \), \( f(n) = \Omega(f(n)) \).
  
  – **Symmetry:** \( f(n) = \Theta(g(n)) \) if and only if \( g(n) = \Theta(f(n)) \).

• Although any two real numbers can be compared, **not all functions are asymptotically comparable.**
  
  – That is, for two functions \( f(n) \) and \( g(n) \), it may be the case that neither \( f(n) = O(g(n)) \) nor \( f(n) = \Omega(g(n)) \) holds.
Asymptotic Notation (Recap)

- The statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ $\forall \ n \geq n_0$.
- Most frequently used asymptotic bounds:
  - $O(1)$ means that a computing time is a constant
  - $O(\log n)$ logarithmic
  - $O(n)$ linear
  - $O(n \log n)$
  - $O(n^2)$ quadratic
  - $O(n^3)$ cubic
  - $O(2^n)$ exponential
Introduction to Design

• There are many way to design algorithms
  - induction (incremental approach):
    insertion sort
  - divide and conquer:
    merge sort