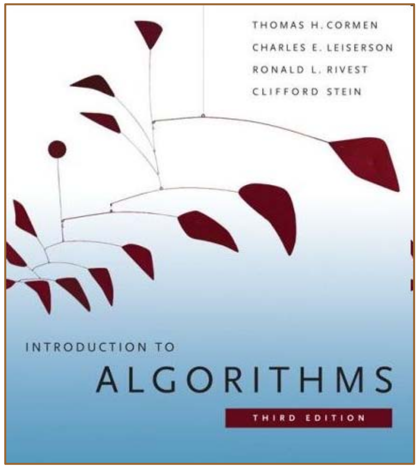
Design and Analysis of Computer Algorithms
演算法分析與設計

Yuan-Hao Chang (張原豪)
johnsonchang@ntut.edu.tw
Department of Electronic Engineering
National Taipei University of Technology



Course Information



- 授課教師: 張原豪 (207-2 室、分機 2288)
- 上課時間: 星期一 1:10 pm ~ 4:00 pm
- 教室: 六教 626
- 參考書目：
  - Introduction to Algorithms, 3rd Edition, 2009
    Authors: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
    Publisher: (開發圖書代理)
    ISBN: 978-0-262-53305-8
- 課程網頁:
  - http://www.ntut.edu.tw/~johnsonchang/courses/Algorithm201008/
- 成績評量：(subject to changes)
  - 作業: (30%), 期中考(30%), 期末考(30%), 平時表現(10%)



Why We Need Algorithms?

- Algorithms help us to understand ***scalability***.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behavior.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!



Outline of the Course

- Getting started – asymptotic notation
- Divide-and-conquer
- Dynamic programming
- Greedy algorithms
- Amortized analysis
- NP-Completeness
- Approximation algorithms
- Linear programming (optional)
- Graph algorithms (optional)
- String matching (optional)
- Probabilistic analysis and randomized algorithms (optional)



Topic 1:
Getting Started
– Asymptotic Notation

Outline

- Insertion sort
- Analyzing algorithms
- Growth of functions



Insertion Sort

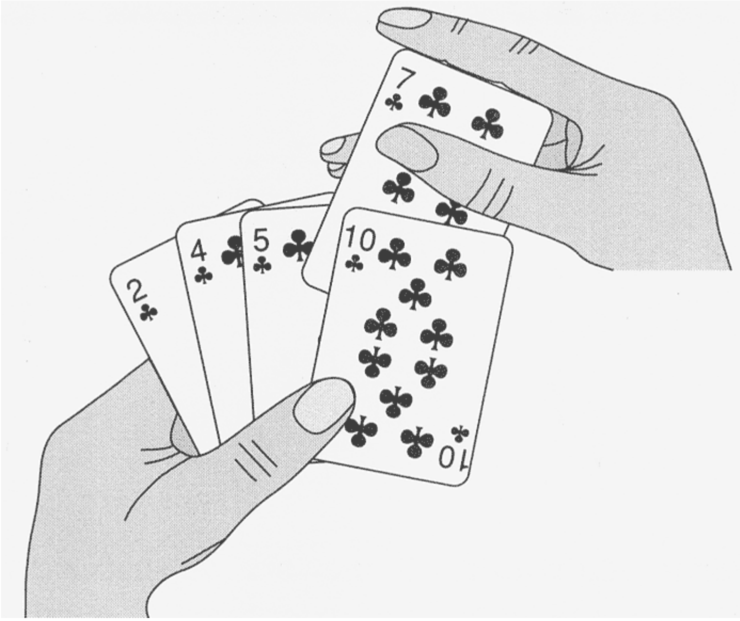


The Problem of Sorting

- **Algorithm:**
  - Sorting problem.
- **Input :**
  - Sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.
- **Output**:
  - A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Insertion Sort

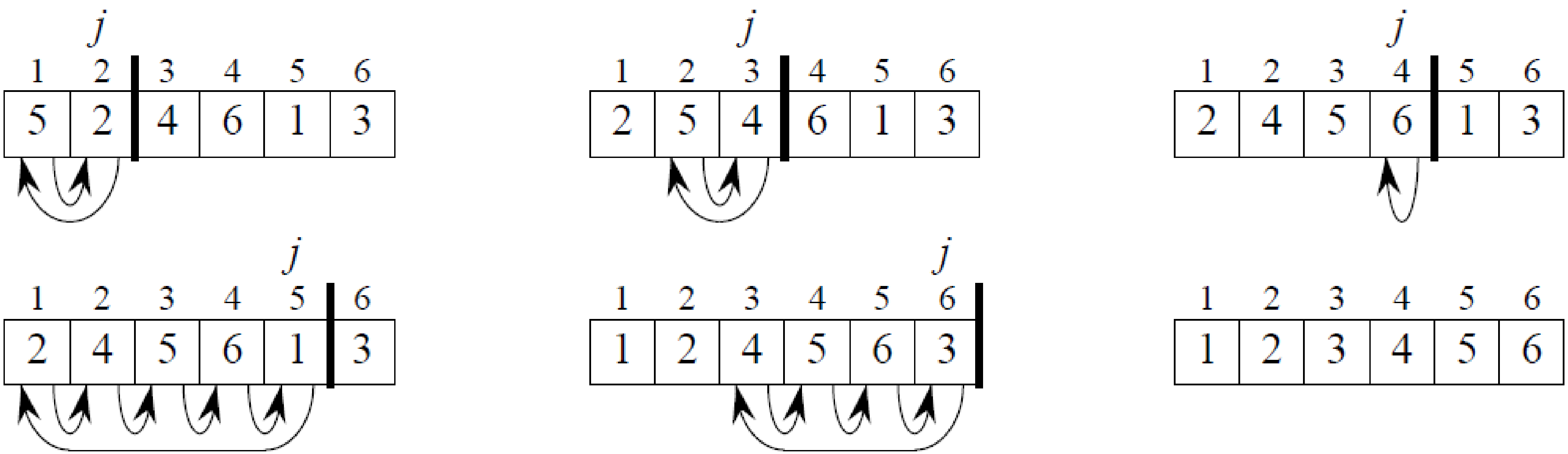
- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.



Insertion Sort – Example 1

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	
2	3	4	6	8	9	*done*

Insertion Sort – Example 2



Insertion Sort - Pseudocode

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$\quad i = j-1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\qquad i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

t_j: the number of times "the ***while*** loop test" is executed for that value of j.



Loop Invariant

- To use a loop invariant to prove correctness, we must show three things about it:
  - Initialization
    - It is true prior to the first iteration of the loop.
  - Maintenance
    - If it is true before an iteration of the loop, it remains true before the next iteration.
  - Termination
    - When the loop terminates, the invariant – usually along with the reason that the loop terminated – gives us a useful property that helps show that the algorithm is correct.



Loop Invariant for Insertion Sort

- Initialization
  - Before the first iteration (j=2), the subarray A[1..j-1] = A[1], which is sorted.
- Maintenance
  - In each iteration, the searched *key* (i.e., A[j]) is inserted to the proper position. Each iteration moves one position to the right.
- Termination
  - When j > n (i.e., j=n+1), the original subarray A[1..n] is orginally sorted.



Analyzing Algorithms



Random-Access Machine (RAM) Model

- Instructions are executed one after another. No concurrent operations.
- It is too tedious to define each of the instructions and their associated time costs.
- We use instructions commonly found in real computers, and each instruction takes a constant amount of time:
  - Arithmetic: add, substract, multiply, divide, remainder, floor, ceiling).
  - Data movement: load, store, copy.
  - Control: conditional/unconditional branch, subroutine call and return.



Input Size

- Input size depends on the problem being studied.
  - The number of items in the input.
    - Like the size *n* of the array being sorted.
  - The total number of bits in the two integers.
    - Like multiplying two integers.
  - Could be described by more than one number.
    - E.g., graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.



Running Time

- On the particular input, running time is the number of primitive operations (steps) executed.
  - Define steps to be machine-independent.
  - Figure that each line of pseudocode requires a constant amount of time.
  - Each execution of the line *i* takes the same amount of time c_i.
  - Assume that the line consists of only primitive operations.
    - If the line is a subroutine call, then the actual call takes constant time.
      - But the execution of the subroutine being called might not.
    - If the line specifies operations other than primitive ones, then it might take more than constant time.
      - E.g., “sort the points by x-coordinate.”



Analysis of Insertion Sort

- The running time of an algorithm:

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

- Let *T(n)* = running time of INSERTION-SORT

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$



Analysis of Insertion Sort (Cont.)

- The best case (the array is already sorted)

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

$T(n) = an + b$ for constants a and b(that depend on the statement costs c_i)

➔A *linear function of n*

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

Analysis of Insertion Sort (Cont.)

- The worst case (the array is in reverse sorted order)

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j \text{ and } \sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1).$$

Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

arithmetic series
(the parentheses is not necessary)

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

$$T(n) = an^2 + bn + c$$

for constants a, b, c

➔A *quadratic function of n*

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
key $= A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] >$ *key*	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] =$ *key*	c_8	$n-1$



Worst Case and Average Case Analysis

- We usually concentrate on finding the ***worst-case running time***: the longest running time for *any* input of size n.
  - The worst-case running time gives a guaranteed upper bound on the running time for any input.
  - For some algorithms, the worst case occurs often.
    - Searches for absent items may be frequent.
  - Why not analyze the average case? Because it's often about as bad as the worst case.
    - Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of n.
    - E.g., in the insertion sort, the average value of $t_j \approx j/2$



Order of Growth

- Another abstraction to ease analysis and focus on the important features.
- Look only at the leading term of the formula for running time.
  - Drop lower-order terms.
  - Ignore the constant coefficient in the leading term.
- ***Example:*** For insertion sort, the worst-case running time is $an^2 + bn + c = \Theta(n^2) = O(n^2)$.
  - Drop lower-order terms ➔ an^2.
  - Ignore constant coefficient ➔ n^2 *(the order of growth).*



Growth of Functions



Asymptotic Notations

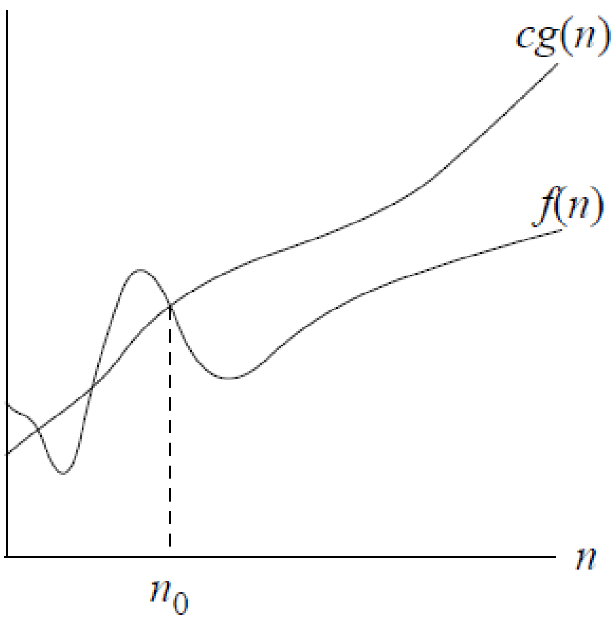
- Describe behavior of functions *in the limit*.
- Describe *growth* of functions.
- Focus on what's important by abstracting away *low-order terms* and *constant factors*.
- Compare "sizes" of functions:

$$
\begin{array}{ccc}
O & \approx & \leq \\
\Omega & \approx & \geq \\
\Theta & \approx & = \\
o & \approx & < \\
\omega & \approx & >
\end{array}
$$



O-Notation

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.



$2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2

$n^2 + n$

$n^2 + 1000n$

$1000n^2 + 1000n$

Also,

n

$n/1000$

$n^{1.99999}$

$n^2/\lg\lg\lg n$

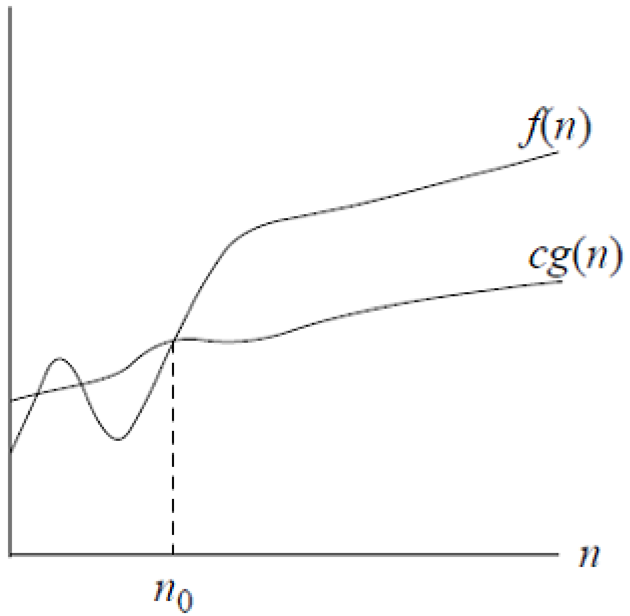
$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$



Ω-Notation

$\Omega(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$.



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

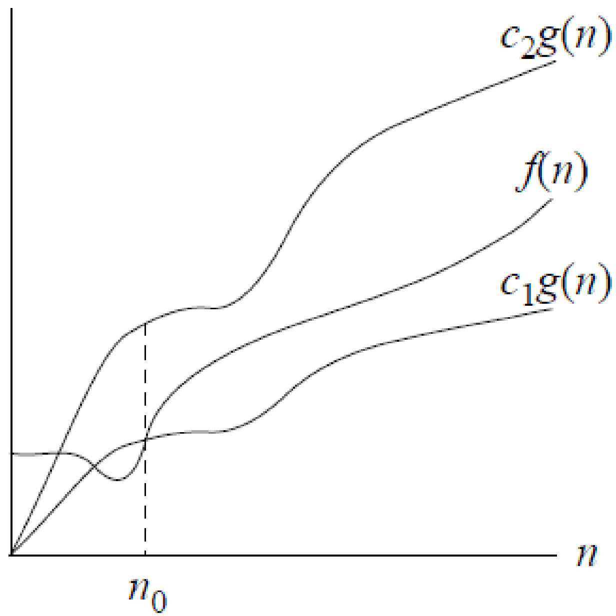
$\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg \lg \lg n$
2^{2^n}

Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$



$n^2/2 - 2n = \Theta(n^2)$,
with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Theorem:
f(n) = Θ(g(n)) iff *f = O(g(n)) and f = Ω(g(n))*
Leading constants and *low-order terms* don't matter.

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.



o-Notation

$o(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Another view, probably easier to use: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

$f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$.



ω-Notation

$$\omega(g(n)) = \{f(n) : \text{ for all constants } c > 0 \text{, there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\} .$$

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$$n^{2.0001} = \omega(n^2)$$
$$n^2 \lg n = \omega(n^2)$$
$$n^2 \ne \omega(n^2)$$

$f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.



Asymptotic Notation in Equations

- ***When on right-hand side***
  - $\Theta(n)$ stands for some anonymous function in the set $\Theta(n)$.
  - E.g., $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

$$2n^2 + 3n + 1 = 2n^2 + f(n) \text{ for some } f(n) \in \Theta(n). \text{ In particular, } f(n) = 3n + 1.$$

- ***When on left-hand side***
  - $2n^2 + \Theta(n) = \Theta(n^2)$

$$\text{for all functions } f(n) \in \Theta(n), \text{ there exists a function } g(n) \in \Theta(n^2) \text{ such that } 2n^2 + f(n) = g(n).$$

Monotonicity

- *f(n)* is ***monotonically increasing (non-decreasing)*** if $m \le n \Rightarrow f(m) \le f(n)$
- *f(n)* is ***monotonically decreasing (non-increasing)*** if $m \ge n \Rightarrow f(m) \ge f(n)$
- *f(n)* is ***strictly increasing*** if $m < n \Rightarrow f(m) < f(n)$
- *f(n)* is ***strictly decreasing*** if $m > n \Rightarrow f(m) > f(n)$



Project 1

- Use C language to implement the insertion sort (suggested tool: Dev C++).
  - Use *fscanf()* to get integers from the input file.
  - Sort the input integers and output the sorted integers in the ***monotonically increasing*** order on the screen.
- Deadline: 24:00, 2010.09.20
  - Email the .c or .cpp program to me: johnsonchang@ntut.edu.tw
  - Email title: Algo_P1_學號_姓名