Algorithms

EE4033; #901/39000

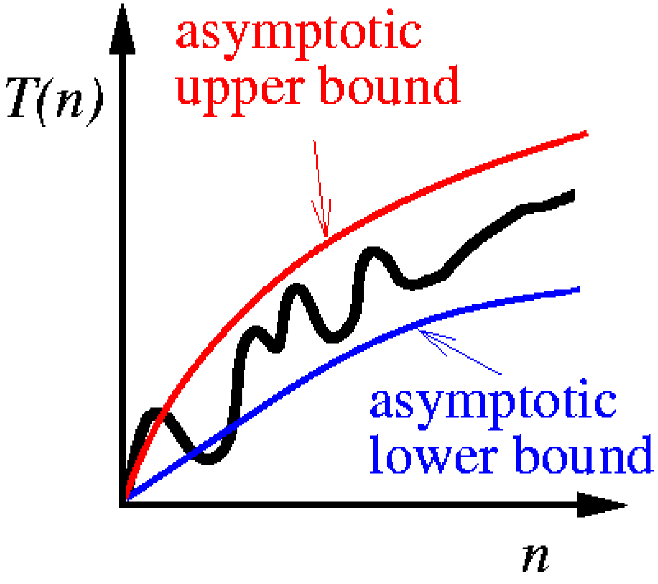
張耀文
Yao-Wen Chang
ywchang@ntu.edu.tw
http://cc.ee.ntu.edu.tw/~ywchang

Graduate Institute of Electronics Engineering
Department of Electrical Engineering
National Taiwan University

Fall 2018

Unit 1: Algorithmic Fundamentals

- **Course contents:**
  - On algorithms
  - Mathematical foundations
  - Asymptotic notation
  - Growth of functions
  - Recurrences
- **Readings:**
  - Chapters 1, 2, 3, 4
  - Appendix A



On Algorithms

- **Algorithm:** A well-defined procedure for transforming some **input** to a desired **output**.
- **Major concerns:**
  - **Correctness:** Does it **halt**? Is it **correct**? Is it **stable**?
  - **Efficiency:** **Time** complexity? **Space** complexity?
    - Worst case? Average case? (Best case?)

Resource usage

- **Better algorithms?**
  - **How: Faster** algorithms? Algorithms with **less space** requirement?
  - **Optimality:** Prove that an algorithm is **best possible/optimal**? Establish a **lower bound**?
- **Applications?**
  - **Everywhere in computing!**

Example: Sorting

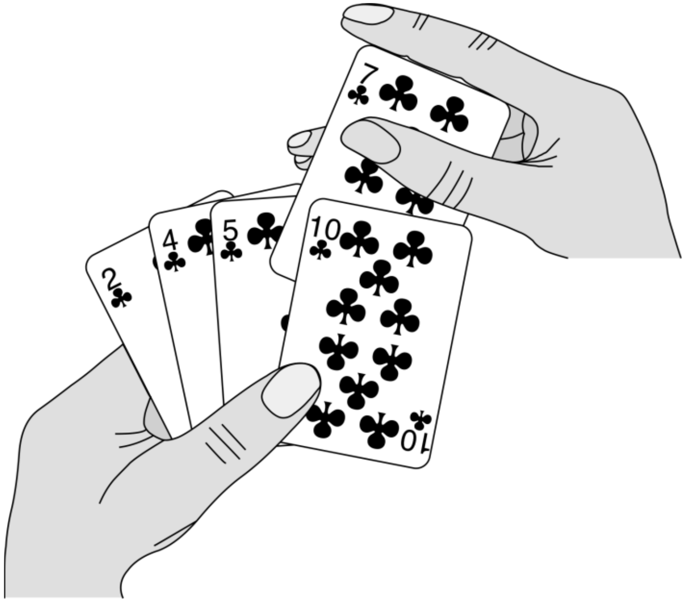
- **Input:** A sequence of n numbers $<a_1, a_2, \ldots, a_n>$.
- **Output:** A permutation $<a_1', a_2', \ldots, a_n'>$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

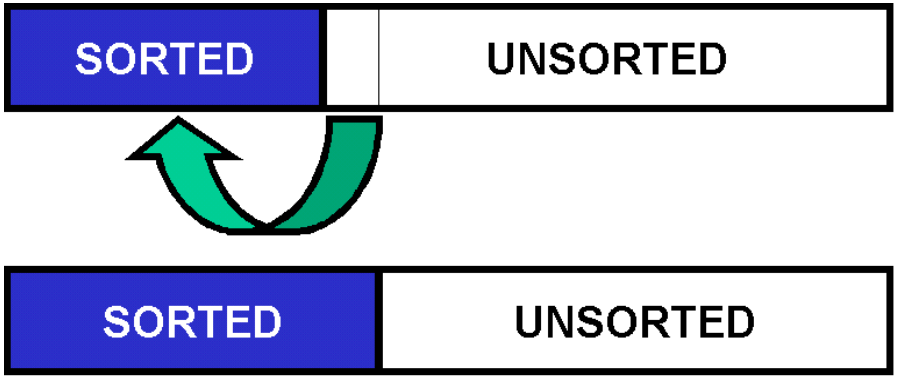
  Input: <8, 6, 9, 7, 5, 2, 3>

  Output: <2, 3, 5, 6, 7, 8, 9>

- Correct and efficient algorithms?

Incremental Approach: Insertion Sort





What is the invariant of this sort?

- **How do you sort cards?**

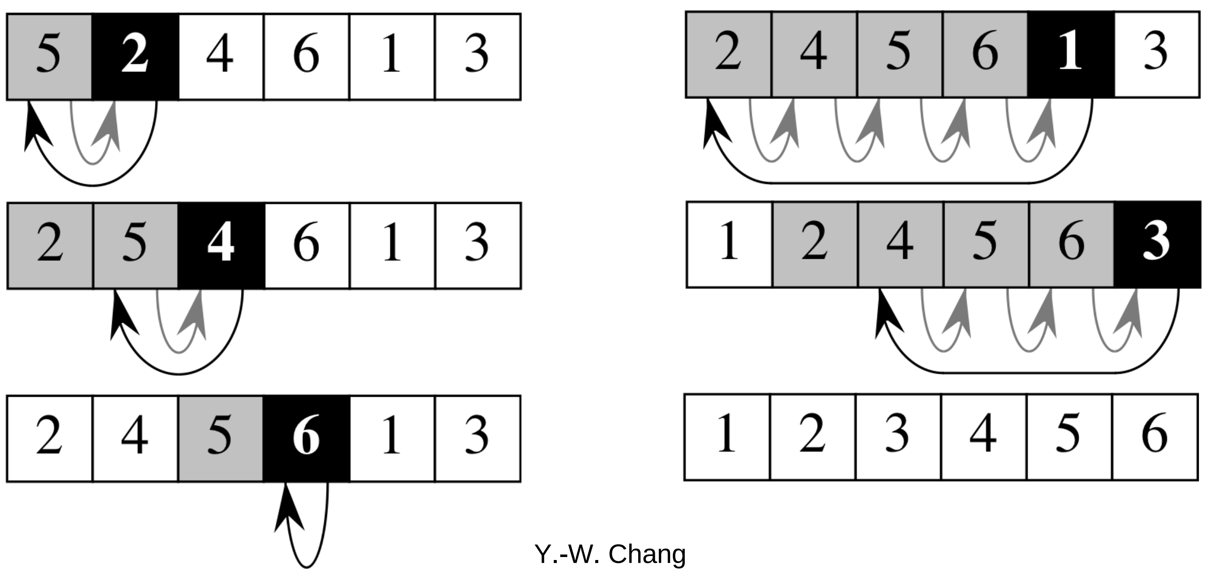
1. Keep left cards sorted, right cards unsorted
2. Each time insert a new card to left cards, in sorted order
3. Repeat until all cards inserted

Insertion Sort

6 5 3 1 8 7 2 4

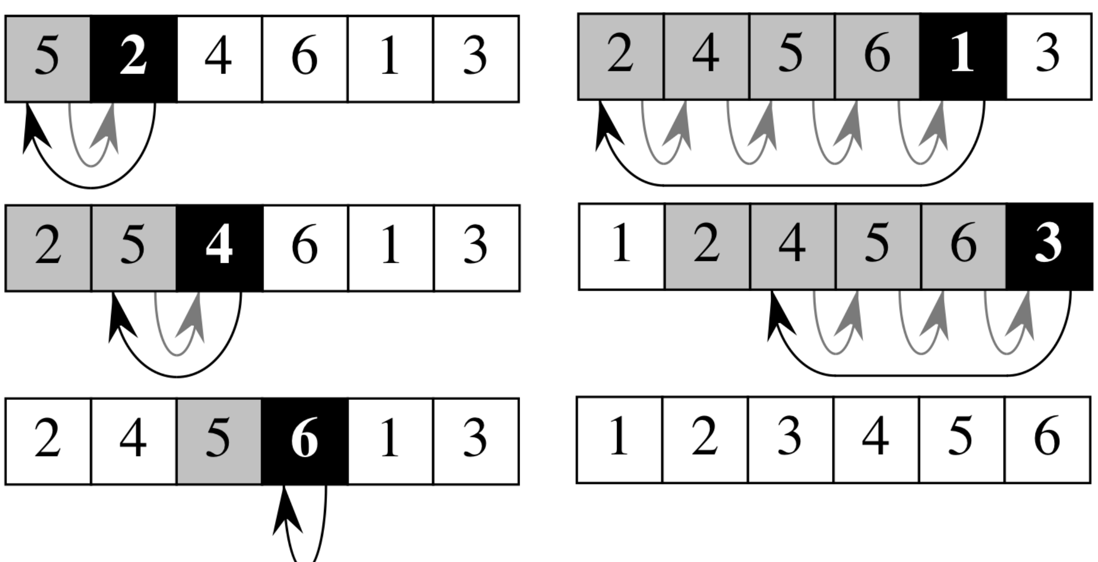
$j = 2$
$key = 2$
$i = 1$
$A[2] = 5$
$i = 0$
$A[1] = 2$

InsertionSort(A)
1. **for** $j = 2$ **to** $A.length$
2. $key = A[j]$
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. $A[i+1] = A[i]$ // Right shift
7. $i = i - 1$
8. $A[i+1] = key$



Correctness?

```
InsertionSort(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]      // Right shift
        i = i - 1
    A[i+1] = key
```



Loop invariant: subarray $A[1..j\text{-}1]$ consists of the elements originally in $A[1..j\text{-}1]$ but in sorted order.

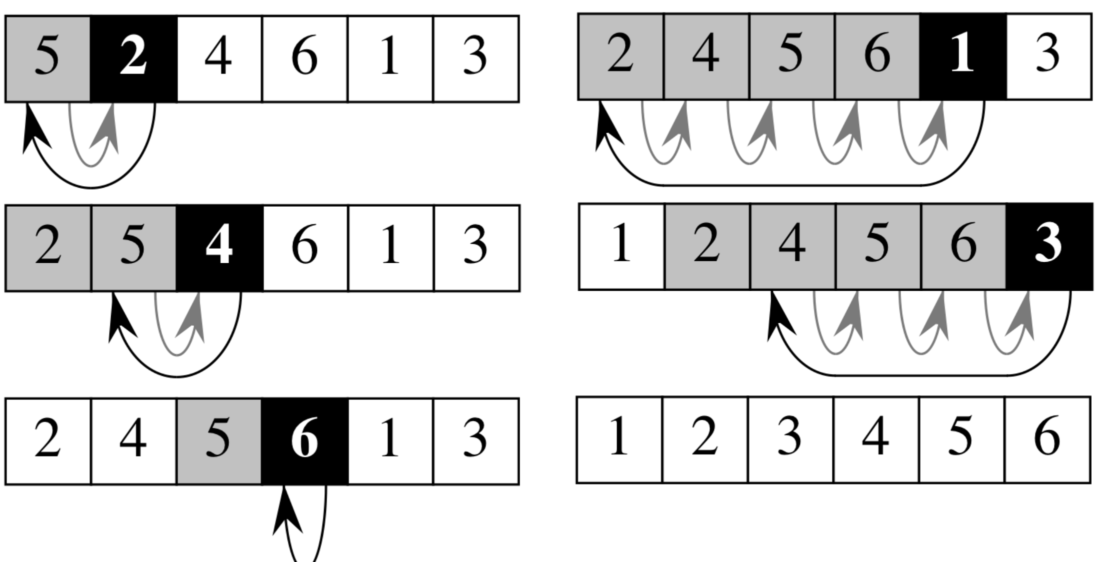
Loop Invariant for Proving Correctness

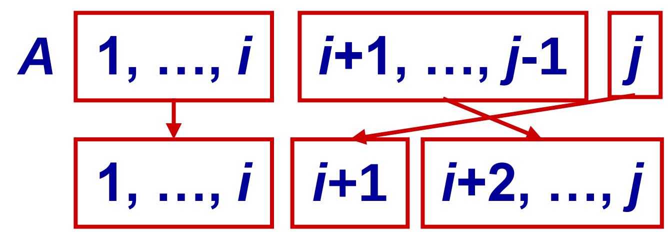
```
InsertionSort(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

- We may use **loop invariants** to prove the correctness.
  1. **Initialization:** True before the 1st iteration.
  2. **Maintenance:** If it is true before an iteration, it remains true before the next iteration.
  3. **Termination:** When the loop terminates, the invariant leads to the correctness of the algorithm.
  - **Mathematical induction!**

Loop Invariant of Insertion Sort

- **Loop invariant:** subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$ but in sorted order.
  - **Initialization:** $j = 2 \Rightarrow A[1]$ is sorted.
  - **Maintenance:** Move $A[j-1]$, $A[j-2]$,… one position to the right until the position for $A[j]$ is found.
  - **Termination:** $j = \boldsymbol{n+1} \Rightarrow A[1..n]$ is sorted. Hence the entire array is sorted!





Exact Analysis of Insertion Sort

InsertionSort(*A*)	cost	time
1. **for** *j* = 2 **to** *A.length*	c_1	n
2. *key* = *A*[*j*]	c_2	n-1
3. // Insert *A*[*j*] into the sorted sequence *A*[1..*j*-1]	0	n-1
4. *i* = *j* - 1	c_4	n-1
5. **while** *i* > 0 and *A*[*i*] > *key*	c_5	$\sum_{j=2}^{n} t_j$
6. *A*[*i*+1] = *A*[*i*]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7. *i* = *i* - 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8. *A*[*i*+1] = *key*	c_8	n-1

- Line 1 is executed (n-1) **+ 1** times. (why?)
- t_j: # of times the **while** loop test for value j (i.e., 1 + # of elements that have to be slid right to insert the j-th item).
- Step 5 is executed $t_2 + t_3 + \ldots + t_n$ times.
- Step 6 is executed $(t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)$ times.
- Run time $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$

Exact Analysis of Insertion Sort (cont'd)

InsertionSort(*A*)	cost	time
1. **for** *j* = 2 **to** *A.length*	c_1	n
2. *key* = *A*[*j*]	c_2	n-1
3. // Insert *A*[*j*] into the sorted sequence *A*[1..*j*-1]	0	n-1
4. *i* = *j* - 1	c_4	n-1
5. **while** *i* > 0 and *A*[*i*] > *key*	c_5	$\sum_{j=2}^{n} t_j$
6. *A*[*i*+1] = *A*[*i*]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7. *i* = *i* - 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8. *A*[*i*+1] = *key*	c_8	n-1

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$
- **Best case:** If the input is already sorted, all t_j's are 1.
  Linear: $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$
- **Worst case:** If the array is in reverse sorted order, $t_j = j$, $\forall j$.
  Quadratic: $T(n) = (c_5/2 + c_6/2 + c_7/2)\, n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)\, n - (c_2 + c_4 + c_5 + c_8)$
- **Exact analysis is often hard (and tedious)!**

Complexity

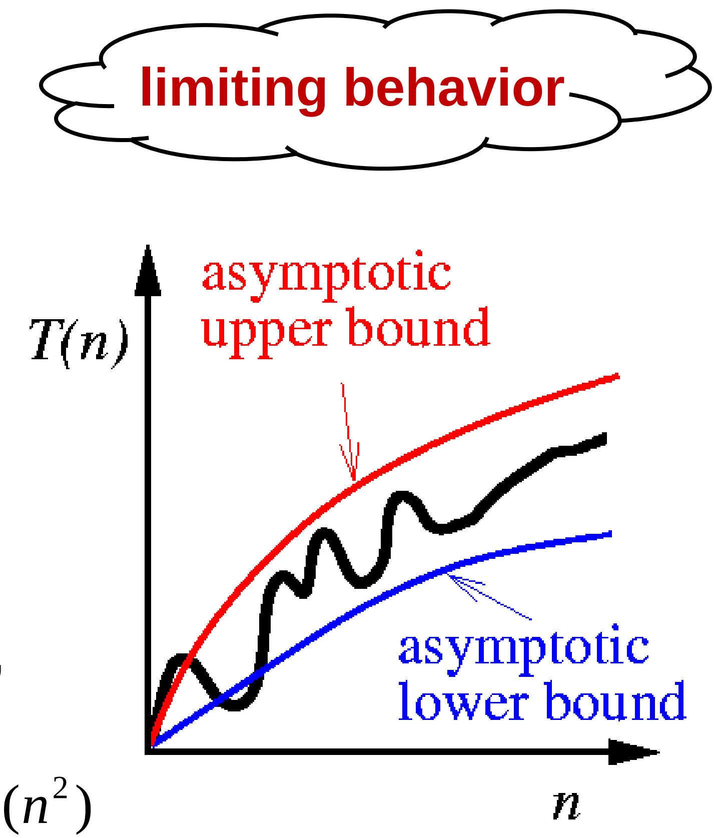
- During exact analysis, each line corresponds to several steps, each step requires a constant running time
- Computational complexity: an abstract measure of time and space necessary to execute an algorithm as functions of its "input size"
  - Time complexity $\Rightarrow$ running time (in terms of steps)
  - Space complexity $\Rightarrow$ memory requirement
- **Input size**: the number of alphabet symbols needed to encode the input
  - sort n integers $\Rightarrow$ input size: n
- **Step**: primitive operation
  - The time to execute a primitive operation must be constant: it must not increase as the input size grows

Asymptotic Analysis

- Asymptotic analysis looks at growth of $T(n)$ as $n \to \infty$
- Θ notation: Drop low-order terms and ignore the leading constant

  E.g., $8n^3 - 4n^2 + 5n - 2 = \Theta(n^3)$
- As n grows large, lower-order Θ algorithms outperform higher-order ones
- **Worst case:** input sorted in reverse, **while** loop is $\Theta(j)$

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(\sum_{j=2}^{n} j) = \Theta(n^2)$$



- **Average case:** all permutations equally likely, **while** loop is executed about $j/2$ times each iteration

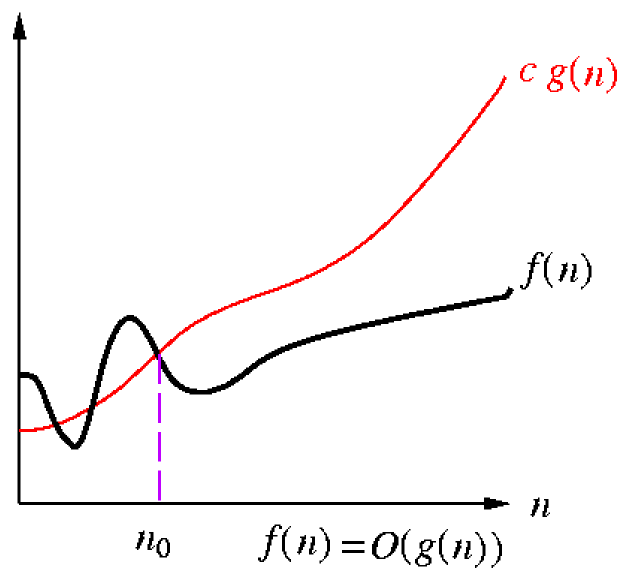
$$T(n) = \sum_{j=2}^{n} j/2 = \Theta(\sum_{j=2}^{n} j/2) = \Theta(n^2)$$

O: Upper Bounding Function

- **Def:** $f(n) = O(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \le \mathbf{f(n) \le cg(n)}$ for all $n \ge n_0$
- Intuition: $f(n)$ "$\le$" $g(n)$ when we ignore constant multiples and small values of n
- How to **verify** O (Big-Oh) relationships?
  - $f(n) = O(g(n))$ when $\lim_{n \to \infty} \frac{f(n)}{g(n)} \in [0,\infty)$, <u>if the limit exists</u>

Sufficient but not necessary condition

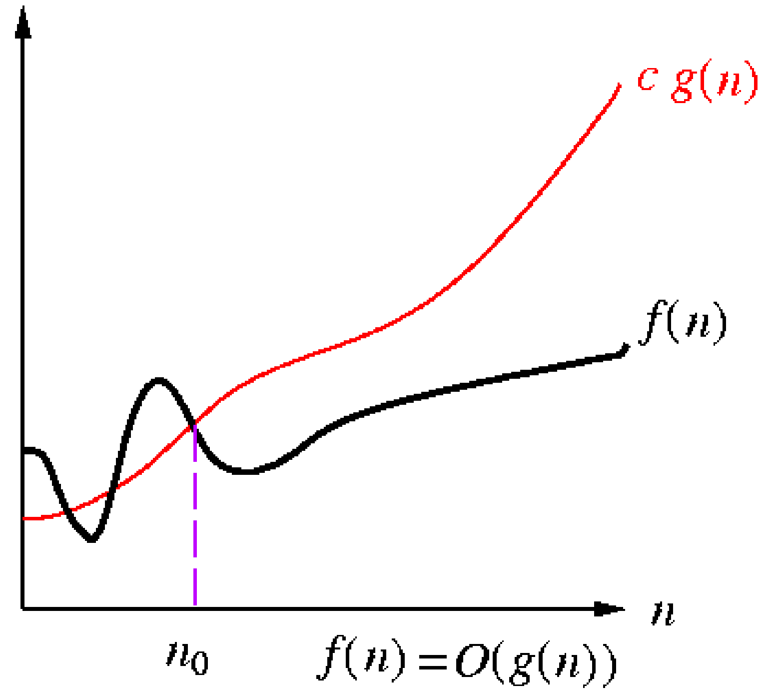
***f*(*n*): nonnegative function**
***n*: natural number**



Big-Oh Examples

- **Def:** $f(n) = O(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \leq \boldsymbol{f(n) \leq cg(n)}$ for all $n \geq n_0$.

1. $3n^2 + n = O(n^2)$? **Yes**
2. $3n^2 + n = O(n)$? **No**
3. $3n^2 + n = O(n^3)$? **Yes**



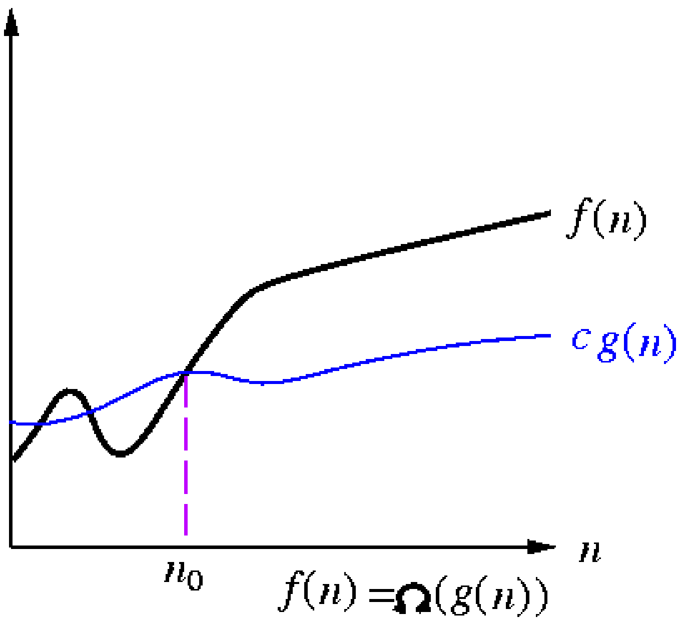
$$3n^2 + n \leq cn^2?$$

Take $c = 4$, $n_0 = 1$

$f(n) = O(g(n))$ when $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} \in [0, \infty)$, <u>if the limit exists</u>

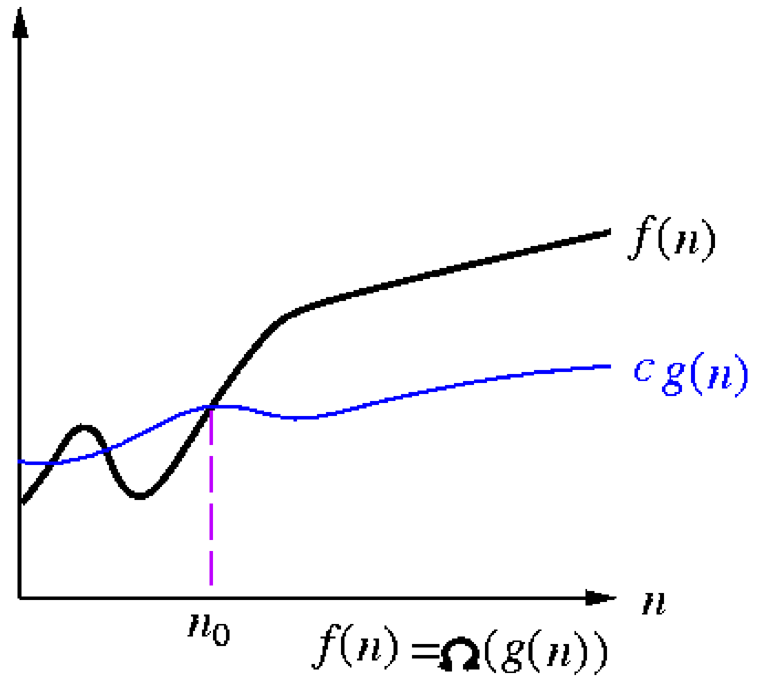
Ω : Lower Bounding Function

- **Def:** $f(n) = \Omega(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \le$ ***cg(n) ≤ f(n)*** for all $n \ge n_0$
- Intuition: $f(n)$ "≥" $g(n)$ when we ignore constant multiples and small values of n
- How to **verify** Ω (Big-Omega) relationships?
  - $f(n) = \Omega(g(n))$ when $\lim_{n \to \infty} \frac{f(n)}{g(n)} \in (0, \infty]$, if the limit exists



Big-Omega Examples

- **Def:** $f(n) = \Omega(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \le$ $\mathbf{cg(n) \le f(n)}$ for all $n \ge n_0$.

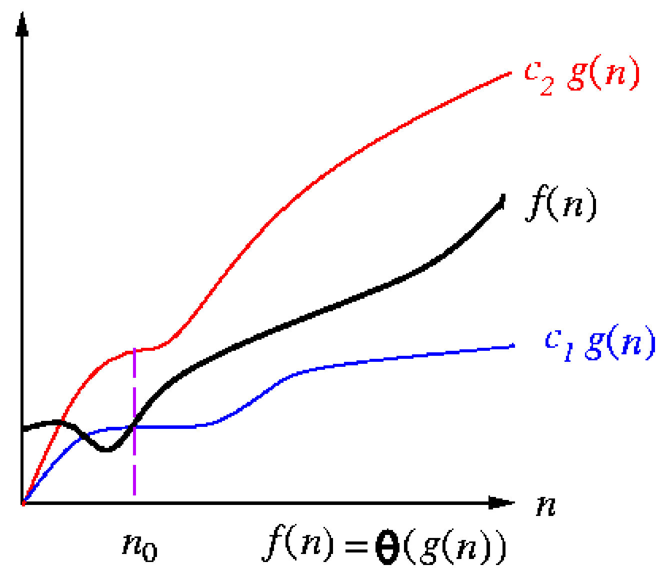


1. $\mathbf{3n^2 + n = \Omega(n^2)?}$ **Yes**
2. $\mathbf{3n^2 + n = \Omega(n)?}$ **Yes**
3. $\mathbf{3n^2 + n = \Omega(n^3)?}$ **No**

$f(n) = \Omega(g(n))$ when $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} \in (0, \infty]$, <u>if the limit exists</u>

Θ: Tightly Bounding Function

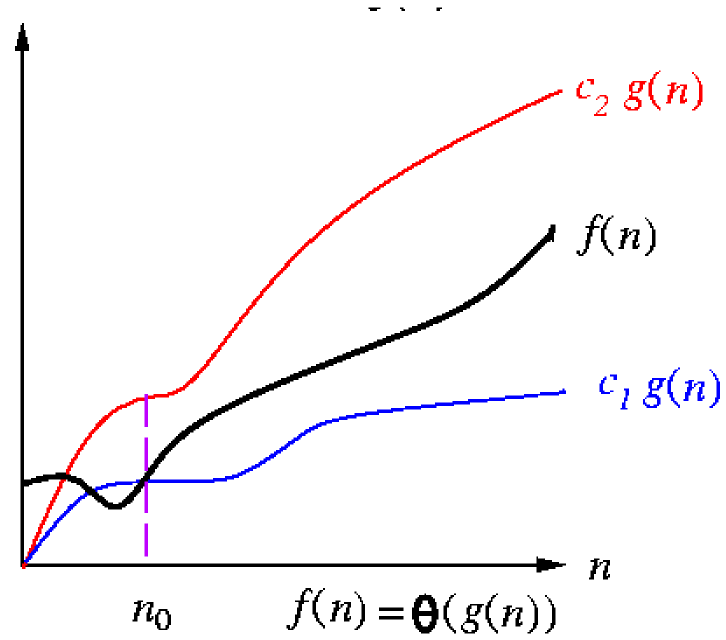
- **Def:** $f(n) = \Theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \le$ **$c_1 g(n) \le f(n) \le c_2 g(n)$** for all $n \ge n_0$
- Intuition: $f(n)$ "**=**" $g(n)$ when we ignore constant multiples and small values of n
- How to **verify** Θ relationships?
  - Show both "big Oh" (O) and "Big Omega" (Ω) relationships
  - $f(n) = \Theta(g(n))$ when $\lim_{n \to \infty} \frac{f(n)}{g(n)} \in (0,\infty)$**,** if the limit exists



Theta Examples

- **Def:** $f(n) = \Theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \le \mathbf{c_1 g(n) \le f(n) \le c_2 g(n)}$ for all $n \ge n_0$.

1. $3n^2 + n = \Theta(n^2)$? **Yes**
2. $3n^2 + n = \Theta(n)$? **No**
3. $3n^2 + n = \Theta(n^3)$? **No**



$f(n) = \Theta(g(n))$ when $\lim_{n \to \infty} \frac{f(n)}{g(n)} \in (0, \infty)$, if the limit exists

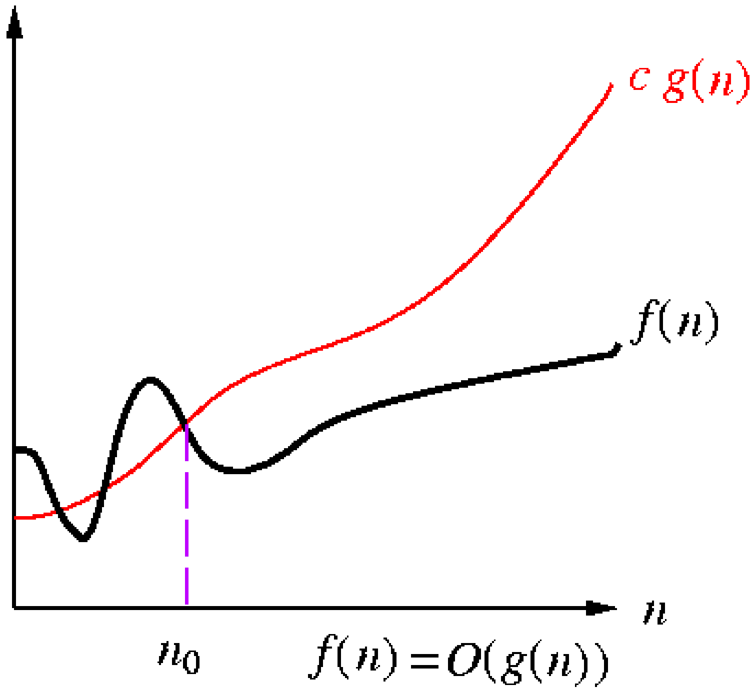
o, ω : Untightly Upper, Lower Bounding Functions

- **Little Oh *o*:** $f(n) = o(g(n))$ if $\forall$ **$c > 0$**, $\exists\, n_0 > 0$ such that $0 \le$ **$f(n) < cg(n)$** for all $n \ge n_0$.
- Intuition: $f(n)$ "**<**" **any constant multiple of** $g(n)$ when we ignore small values of n
- **Little Omega ω :** $f(n) = \omega(g(n))$ if $\forall$ **$c > 0$**, $\exists\, n_0 > 0$ such that $0 \le$ **$cg(n) < f(n)$** for all $n \ge n_0$.
- Intuition: $f(n)$ is "**>**" any constant multiple of $g(n)$ when we ignore small values of n
- How to **verify** o (Little-Oh) and ω (Little-Omega) relationships (if the limit exists)?
  - $f(n) = o(g(n))$ when $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
  - $f(n) = \omega(g(n))$ when $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

Little-Oh Examples

- **Little Oh *o*:** $f(n) = o(g(n))$ if $\forall$ **$c > 0$**, $\exists\ n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.

1. $3n^2 + n = o(n^2)$? **No**
2. $3n^2 + n = o(n)$? **No**
3. $3n^2 + n = o(n^3)$? **Yes**

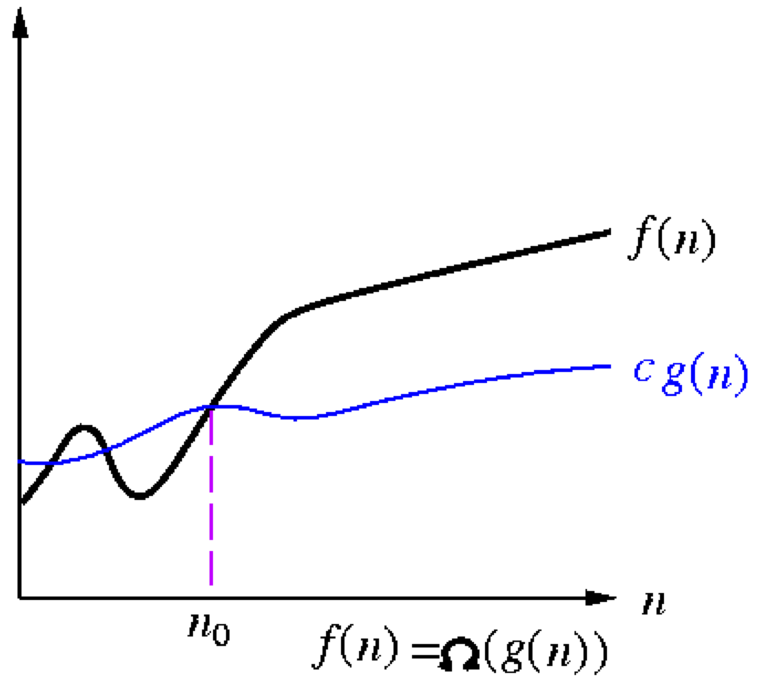


$f(n) = o(g(n))$ when $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = 0$, if the limit exists

Little-Omega Examples

- **Little Omega** ω **:** $f(n) = \omega(g(n))$ if $\forall$ **$c > 0$**, $\exists\, n_0 > 0$ such that $0 \le$ **$cg(n) < f(n)$** for all $n \ge n_0$.

1. $3n^2 + n = \omega(n^2)$? **No**
2. $3n^2 + n = \omega(n)$? **Yes**
3. $3n^2 + n = \omega(n^3)$? **No**



$f(n) = \omega(g(n))$ when $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = \infty$, if the limit exists

Properties for Asymptotic Analysis

- **Transitivity:** If $f(n) = \Pi(g(n))$ and $g(n) = \Pi(h(n))$, then $f(n) = \Pi(h(n))$, where $\Pi = O$, o, Ω, ω, or Θ
- **Rule of sums:** $f(n) + g(n) = \Pi(\max\{f(n), g(n)\})$, where $\Pi = O$, Ω, or Θ
- **Rule of products:** If $f_1(n) = \Pi(g_1(n))$ and $f_2(n) = \Pi(g_2(\mathrm{n}))$, then $f_1(n)\, f_2(n) = \Pi(g_1(n)\, g_2(n))$, where $\Pi = O$, o, Ω, ω, or Θ
- **Transpose symmetry:** $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
- **Transpose symmetry:** $f(n) = \mathrm{o}(g(n))$ iff $g(n) = \omega(f(n))$
- **Reflexivity**: $f(n) = \Pi(f(n))$, where $\Pi = O$, Ω , or Θ
- **Symmetry:** $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

Meaning of Asymptotic Notation

- "An algorithm has the worst-case running time $O(f(n))$": there is a constant c s.t. for every n big enough, **every execution** on an input of size n takes **at most** $cf(n)$ time.

- "An algorithm has the worst-case running time $\Omega(f(n))$": there is a constant c s.t. for every n big enough, **at least one execution** on an input of size n takes **at least** $cf(n)$ time.

Asymptotic Functions

- $\lg^{(i)} n = \underbrace{\lg \lg \ldots \lg}_{i} n.$ (cf. $\lg^{i} n = (\lg n)^{i}$)
- $\lg^{*} n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$
- **Polynomial-time complexity:** $O(p(n))$, where n is the **input size** and $p(n)$ is a polynomial function of n **($p(n) = n^{O(1)}$).**

1	constant
$\lg^{*} n$	iterated logarithm
$\lg^{(O(1))} n = \underbrace{\lg \lg \ldots \lg}_{O(1)} n$	–
$\lg n$	logarithmic
$\lg^{O(1)} n = (\lg n)^{O(1)}$	polylogarithmic
$\sqrt{n}$	sublinear
n	linear
$n \lg n$	loglinear
n^2	quadratic
n^3	cubic
n^4	quartic
$2^n, 3^n, \ldots$	exponential
$n!$	factorial
n^n	-

Polynomial -time complexity!!

Efficient!

Computational Complexity

- Computational complexity: an abstract measure of the **time** and **space** necessary to execute an algorithm as functions of its "input size".
  - Time complexity ⇒ running time (in terms of steps)
  - Space complexity ⇒ memory requirement
- Input size: size of encoded "binary" strings.
  - sort n words of bounded length ⇒ input size: n
  - **the input is the integer n ⇒ input size: lg n**
  - the input is the graph $G(V, E)$ ⇒ input size: $|V|$ and $|E|$
- Runtime comparison: assume 1 BIPS,1 instruction/op.

Time	Big-Oh	$n = 10$	$n = 100$	$n = 10^4$	$n = 10^6$	$n = 10^8$
500	O(1)	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec	$5*10^{-7}$ sec
$3n$	$O(n)$	$3*10^{-8}$ sec	$3*10^{-7}$ sec	$3*10^{-5}$ sec	0.003 sec	0.3 sec
$n \lg n$	$O(n \lg n)$	$3*10^{-8}$ sec	$6*10^{-7}$ sec	$1*10^{-4}$ sec	0.018 sec	2.5 sec
n^2	$O(n^2)$	$1*10^{-7}$ sec	$1*10^{-5}$ sec	0. 1 sec	16.7 min	116 days
n^3	$O(n^3)$	$1*10^{-6}$ sec	0.001 sec	16.7 min	31.7 yr	∞
2^n	$O(2^n)$	$1*10^{-6}$ sec	$4 *10^{11}$ cent.	∞	∞	∞
$n!$	$O(n!)$	0.003 sec	∞	∞	∞	∞