



# Algorithms

#901/39000

張耀文

Yao-Wen Chang

[ywchang@cc.ee.ntu.edu.tw](mailto:ywchang@cc.ee.ntu.edu.tw)

<http://cc.ee.ntu.edu.tw/~ywchang>

Graduate Institute of Electronics Engineering

Department of Electrical Engineering

National Taiwan University

Fall 2009



# Administrative Matters

---

- **Course#:** 901/39000
- **Time/Location:** Thursdays 2:20--5:20pm (**with potential extensions to 5:40pm**); BL-113
- **Instructor:** Yao-Wen Chang (張耀文)
- **E-mail:** ywchang@cc.ee.ntu.edu.tw
- **URL:** <http://cc.ee.ntu.edu.tw/~ywchang>
- **Office:** BL-428. (Tel) 3366-3556; (Fax) 2364-1972
- **Office Hours:** Tuesdays 5--6pm or by appointment
- **Teaching Assistants**
  - Xin-Wei Shih 施信瑋 ([raistlin@eda.ee.ntu.edu.tw](mailto:raistlin@eda.ee.ntu.edu.tw))
  - Shih-Lun Huang 黃士倫 ([aaron@eda.ee.ntu.edu.tw](mailto:aaron@eda.ee.ntu.edu.tw))
  - Office: BL-406; Tel: 23635251 # 6406
  - Office Hours: 12:20-1:20 pm Wednesdays.
- **Textbook:** Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*, 2<sup>nd</sup> Ed., McGraw Hill/MIT Press, 2001

# Teaching Assistants

---



- Xin-Wei Shih 施信璋
- [raistlin@eda.ee.ntu.edu.tw](mailto:raistlin@eda.ee.ntu.edu.tw)
- Office: BL-406; Tel: 23635251 # 6406
- Office hours: 12:20-1:20 pm  
Wednesdays
- 2<sup>nd</sup>-year Ph.D. student



- Shih-Lun Huang 黃士倫
- [aaron@eda.ee.ntu.edu.tw](mailto:aaron@eda.ee.ntu.edu.tw)
- Office: BL-406; Tel: 23635251 # 6406
- 1<sup>st</sup>-year Ph.D. student

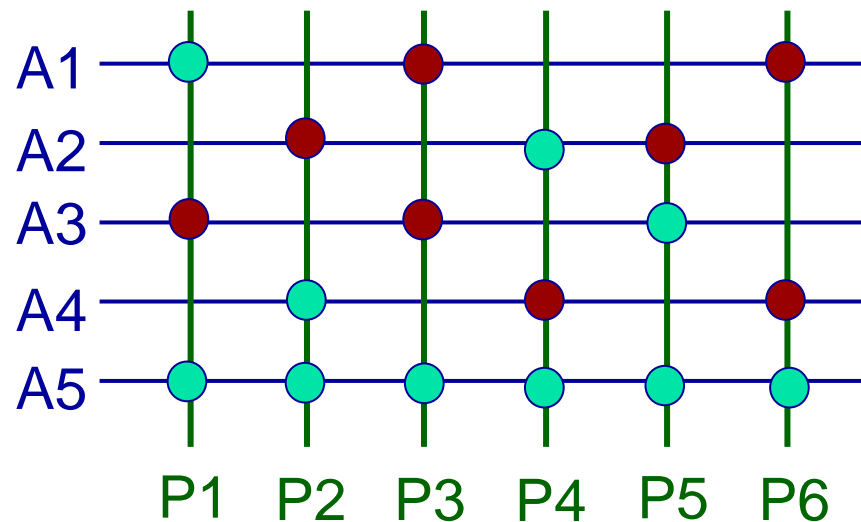
# Course Objectives

---

- Focuses on the ***design and analysis*** of algorithms and their applications



- ***Develops problem-solving techniques!!***



# Prerequisites & Course Contents

---

- **Prerequisites:** data structures (or discrete mathematics)
- **Algorithmic fundamentals:** mathematical foundations, growth of functions, recurrences (5 hrs)
- **Sorting and order statistics (5)**
- **Data structures:** heaps, red-black trees, disjoint sets (4)
- **Advanced design and analysis techniques:** dynamic programming, greedy algorithms, amortized analysis (11)
- **Graph algorithms:** representations, searching, spanning trees, shortest paths, network flow, matching (14)
- **NP-completeness and approximation algorithms (6)**
- **Other algorithms:** computational geometry, branch and bound, and simulated annealing, as time permits

# Grading Policy

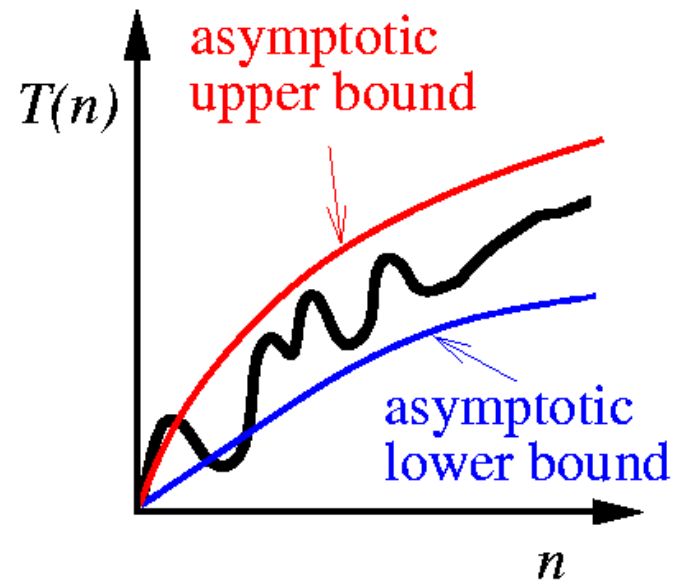
---

- **Grading:**
  - Six homework assignments + quizzes: **30%**
  - Three programming assignments (sorting, dynamic programming, graph algorithm): **20%**
    - All submissions will be checked for duplication & those with  $\geq 50\%$  similarity will be penalized
  - Two in-class tests (**November 5: 18% + January 14: 32%**)
  - Bonuses for class participation
- **Homework/programming assignments:**
  - Penalty for late submission: **20% per day**
- **On-line resources:**
  - <http://cc.ee.ntu.edu.tw/~ywchang/Courses/Alg/alg.html>
- **Academic Honesty: Avoid cheating at all cost**

# 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

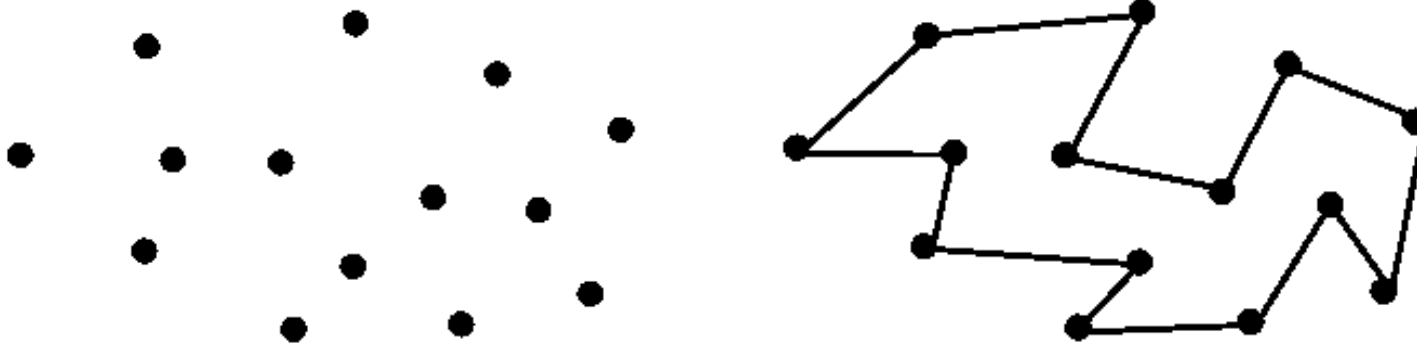
---

- **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?)
- **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: Traveling Salesman Problem (TSP)

---

- **Input:** A set  $P$  of points (cities) together with a distance  $d(p, q)$  between any pair  $p, q \in P$ .
- **Output:** The shortest circular route that starts and ends at a given point and visits all the points.

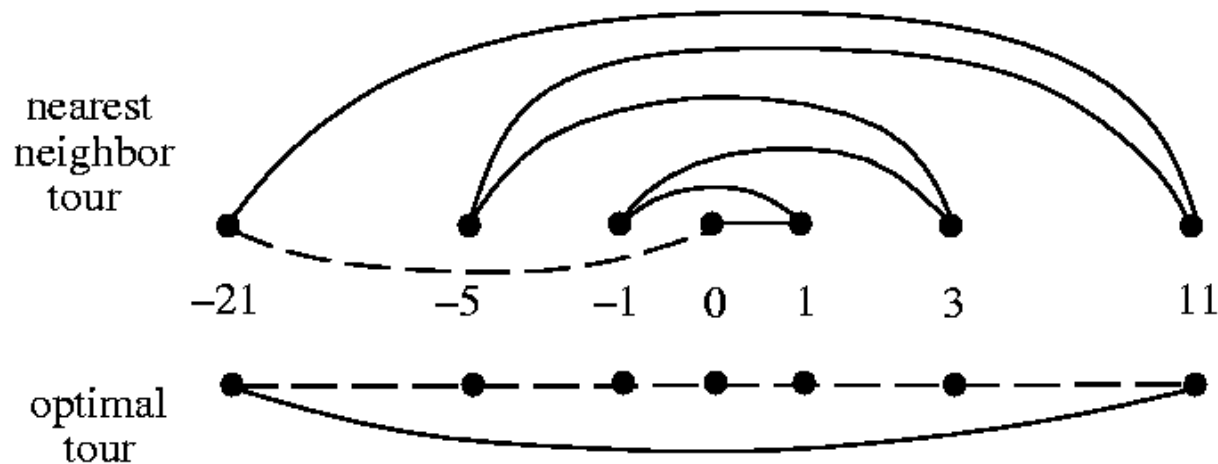


- Correct and efficient algorithms?

# Nearest Neighbor Tour

1. pick and visit an initial point  $p_0$ ;
2.  $i \leftarrow 0$ ;
3. **while** there are unvisited points **do**
4.     visit  $p_i$ 's nearest unvisited point  $p_{i+1}$ ;
5.      $i \leftarrow i + 1$ ;
6. return to  $p_0$  from  $p_i$ .

- Simple to implement and very efficient, but **incorrect!**



# A Correct, But Inefficient Algorithm

---

```
1.  $d \leftarrow \infty$  ;
2. for each of the  $n!$  permutations  $\pi_i$  of the  $n$  points
3.   if ( $\text{cost}(\pi_i) \leq d$ ) then
4.      $d \leftarrow \text{cost}(\pi_i)$ ;
5.      $T_{min} \leftarrow \pi_i$ ;
6. return  $T_{min}$ .
```

- **Correctness?** Tries all possible orderings of the points  $\Rightarrow$  Guarantees to end up with the shortest possible tour.
- **Efficiency?** Tries  $n!$  possible routes!
  - 120 routes for 5 points, 3,628,800 routes for 10 points, 20 points?
- No known efficient, correct algorithm for TSP!
  - TSP is **“NP-complete”**.

## Example: Sorting

---

- **Input:** A sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$ .
- **Output:** A permutation  $\langle a_1', a_2', \dots, a_n' \rangle$  such that  $a_1' \leq a_2' \leq \dots \leq a_n'$ .

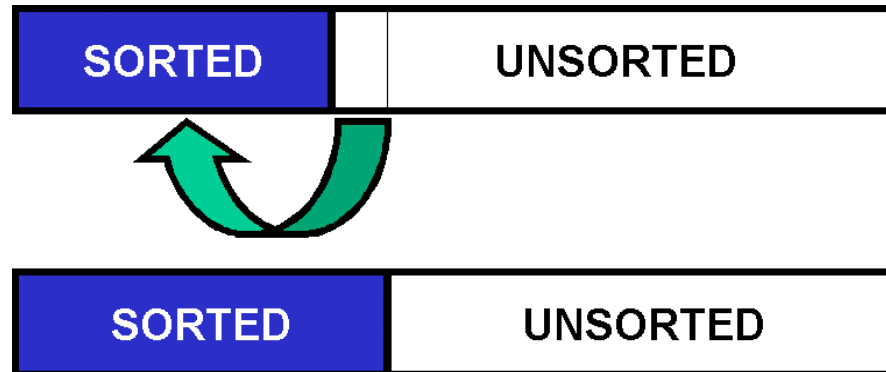
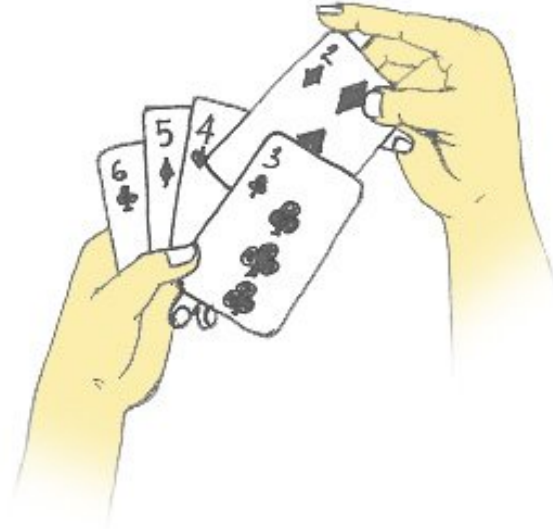
Input:  $\langle 8, 6, 9, 7, 5, 2, 3 \rangle$

Output:  $\langle 2, 3, 5, 6, 7, 8, 9 \rangle$

- Correct and efficient algorithms?

# Insertion Sort Illustration

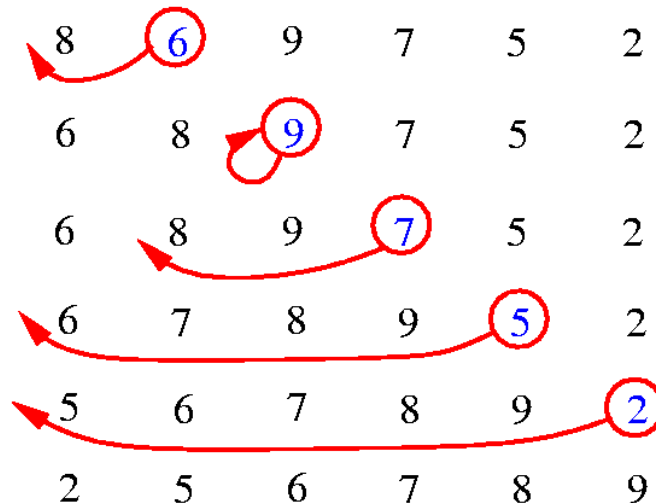
---



**What is the invariant of this sort?**

# Insertion Sort

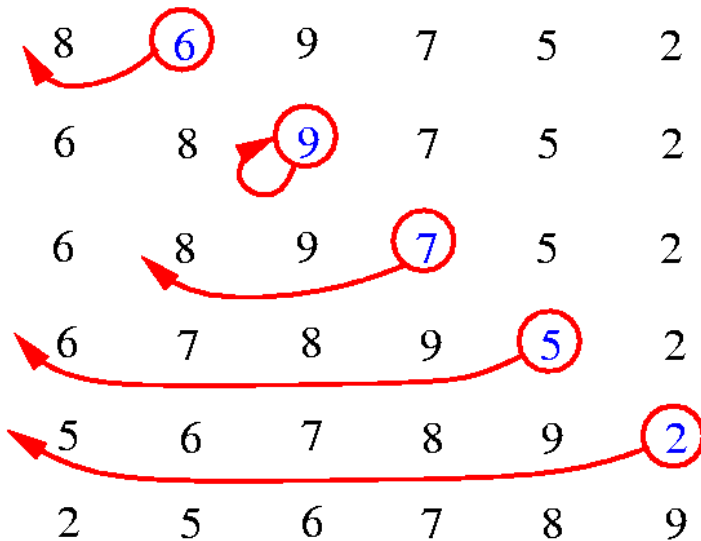
```
InsertionSort(A)
  1. for  $j \leftarrow 2$  to  $\text{length}[A]$  do
    2.    $\text{key} \leftarrow A[j]$ ;
    3.   /* Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ . */
    4.    $i \leftarrow j - 1$ ;
    5.   while  $i > 0$  and  $A[i] > \text{key}$  do
    6.      $A[i+1] \leftarrow A[i]$ ;
    7.      $i \leftarrow i - 1$ ;
    8.    $A[i+1] \leftarrow \text{key}$ ;
```



# Correctness?

InsertionSort( $A$ )

1. **for**  $j \leftarrow 2$  **to**  $\text{length}[A]$  **do**
2.    $\text{key} \leftarrow A[j]$ ;
3.   */\* Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ . \*/*
4.    $i \leftarrow j - 1$ ;
5.   **while**  $i > 0$  and  $A[i] > \text{key}$  **do**
6.        $A[i+1] \leftarrow A[i]$ ;
7.        $i \leftarrow i - 1$ ;
8.    $A[i+1] \leftarrow \text{key}$ ;



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

# Loop Invariant for Proving Correctness

```
InsertionSort(A)
1. for  $j \leftarrow 2$  to  $length[A]$  do
2.    $key \leftarrow A[j]$ ;
3.   /* Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ . */
4.    $i \leftarrow j - 1$ ;
5.   while  $i > 0$  and  $A[i] > key$  do
6.      $A[i+1] \leftarrow A[i]$ ;
7.      $i \leftarrow i - 1$ ;
8.    $A[i+1] \leftarrow key$ ;
```

- We may use **loop invariants** to prove the correctness.
  - **Initialization:** True before the 1<sup>st</sup> iteration.
  - **Maintenance:** If it is true before an iteration, it remains true before the next iteration.
  - **Termination:** When the loop terminates, the invariant leads to the correctness of the algorithm.

# Loop Invariant of Insertion Sort

```
InsertionSort(A)
1. for  $j \leftarrow 2$  to  $\text{length}[A]$  do
2.    $\text{key} \leftarrow A[j]$ ;
3.   /* Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ . */
4.    $i \leftarrow j - 1$ ;
5.   while  $i > 0$  and  $A[i] > \text{key}$  do
6.      $A[i+1] \leftarrow A[i]$ ;
7.      $i \leftarrow i - 1$ ;
8.    $A[i+1] \leftarrow \text{key}$ ;
```

- **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  $A[j-1], A[j-2], \dots$  one position to the right until the position for  $A[j]$  is found.
  - **Termination:**  $j = n+1 \Rightarrow A[1..n]$  is sorted. Hence the entire array is sorted!

# Exact Analysis of Insertion Sort

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

- The **for** loop 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  $t_2 + t_3 + \dots + t_n$  times.
- Step 6 is executed  $(t_2 - 1) + (t_3 - 1) + \dots + (t_n - 1)$  times.
- Run time  $T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1)$

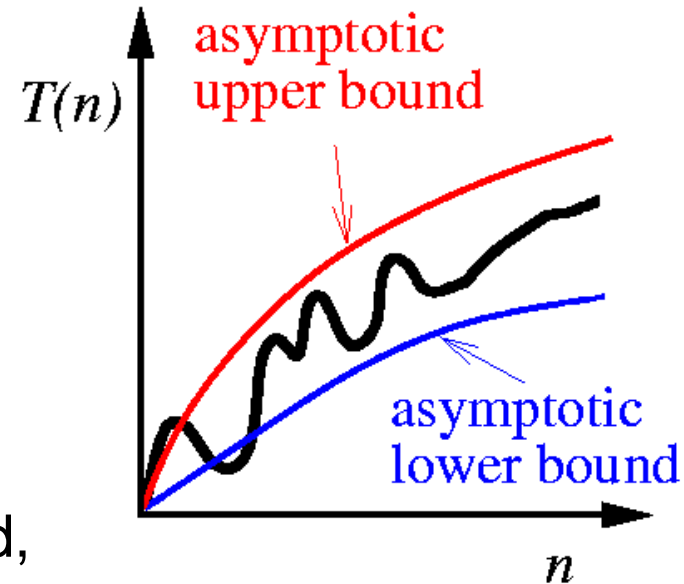
## Exact Analysis of Insertion Sort (cont'd)

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

- $T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1)$
- **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)!**

# Asymptotic Analysis

- Asymptotic analysis looks at growth of  $T(n)$  as  $n \rightarrow \infty$ .
- $\theta$  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  $\theta$  algorithms outperform higher-order ones.



- **Worst case:** input reverse sorted, **while** loop is  $\theta(j)$

$$T(n) = \sum_{j=2}^n \theta(j) = \theta\left(\sum_{j=2}^n j\right) = \theta(n^2)$$

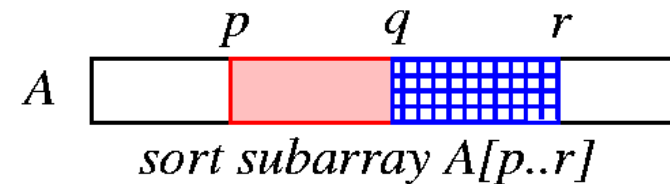
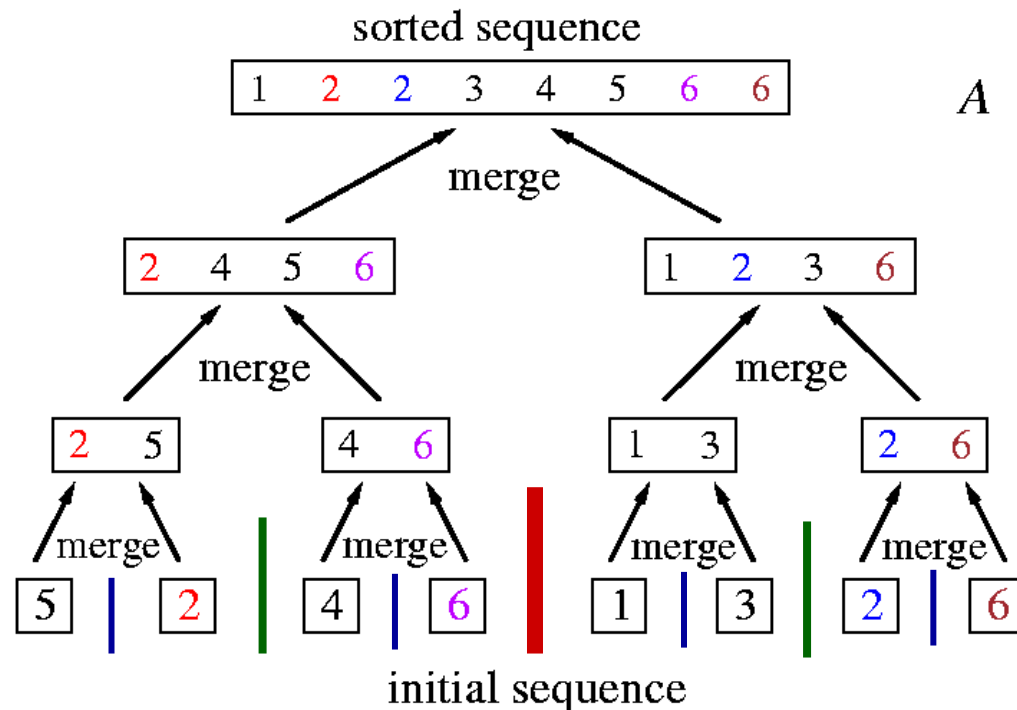
- **Average case:** all permutations equally likely, **while** loop is  $\theta(j/2)$

**Will fix the notation later!**

$$T(n) = \sum_{j=2}^n \theta(j/2) = \theta\left(\sum_{j=2}^n j/2\right) = \theta(n^2)$$

# Merge Sort: A Divide-and-Conquer Algorithm

MergeSort( $A, p, r$ )	$T(n)$
1. <b>if</b> $p < r$ <b>then</b>	$\theta(1)$
2. $q \leftarrow \lfloor (p+r)/2 \rfloor$	$\theta(1)$
3.   MergeSort ( $A, p, q$ )	$T(n/2)$
4.   MergeSort ( $A, q+1, r$ )	$T(n/2)$
5.   Merge( $A, p, q, r$ )	$\theta(n)$



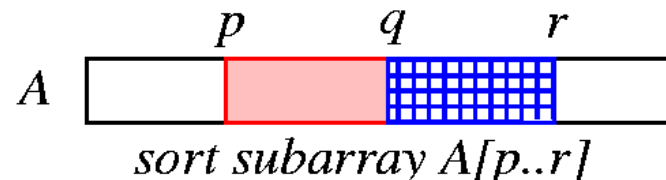
Merge( $A, p, q, r$ ) merges two **sorted** subarrays  $A[p..q]$  and  $A[q+1..r]$  into **sorted**  $A[p..r]$ .

# Recurrence

- Describes a function recursively in terms of itself.
- Describes performance of recursive algorithms.
- Recurrence for merge sort

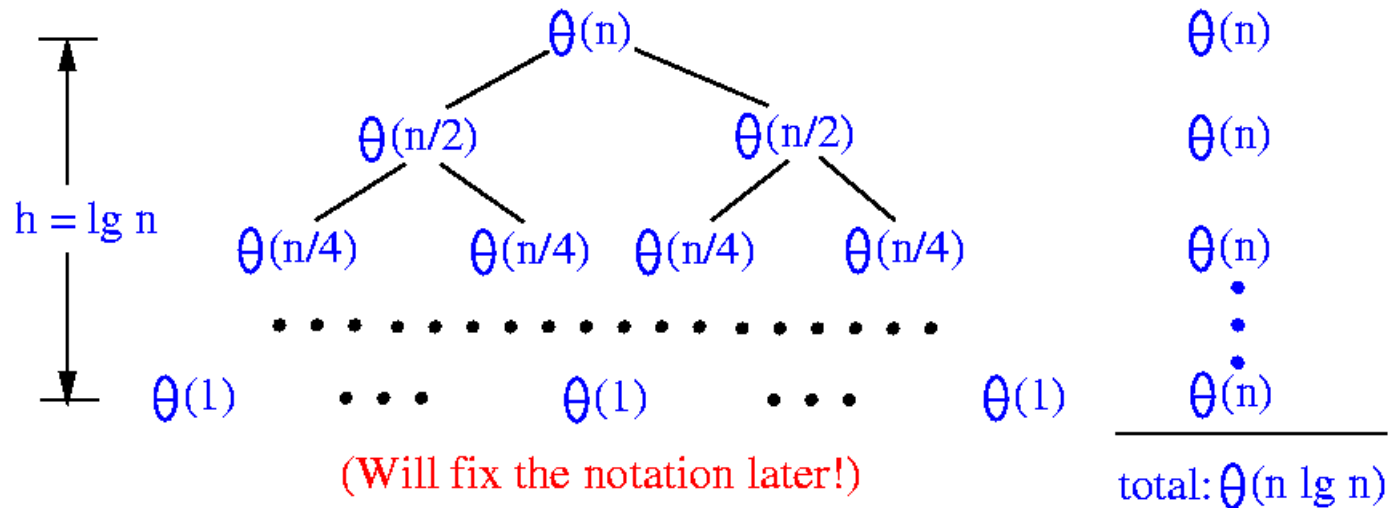
$$T(n) = \begin{cases} \theta(1), & \text{if } n=1 \\ 2 T(n/2) + \theta(n), & \text{if } n>1 \end{cases}$$

MergeSort( $A, p, r$ )	$T(n)$
1. <b>If</b> $p < r$ <b>then</b>	$\theta(1)$
2. $q \leftarrow \lfloor (p+r)/2 \rfloor$	$\theta(1)$
3.   MergeSort ( $A, p, q$ )	$T(n/2)$
4.   MergeSort ( $A, q+1, r$ )	$T(n/2)$
5.   Merge( $A, p, q, r$ )	$\theta(n)$



# Recursion Tree for Asymptotic Analysis

$$T(n) = \begin{cases} \theta(1), & \text{if } n=1 \\ 2 T(n/2) + \theta(n), & \text{if } n>1 \end{cases}$$

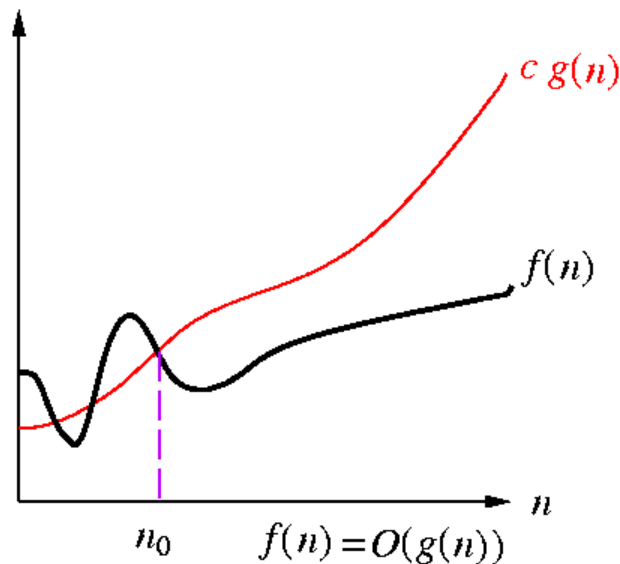


- $\theta(n \lg n)$  grows more slowly than  $\theta(n^2)$ .
- Thus merge sort asymptotically beats insertion sort in the **worst case**.
  - insertion sort: stable, in-place
  - merge sort: stable, not in-place

# O: Upper Bounding Function

---

- **Def:**  $f(n) = O(g(n))$  if  $\exists c > 0$  and  $n_0 > 0$  such that  $0 \leq \mathbf{f(n)} \leq \mathbf{cg(n)}$  for all  $n \geq n_0$ .
- Intuition:  $f(n)$  “ $\leq$ ”  $g(n)$  when we ignore constant multiples and small values of  $n$ .
- How to show O (Big-Oh) relationships?
  - $f(n) = O(g(n))$  implies that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  for some  $c \geq 0$ , if the limit exists.



# Infinity

---

After explaining to a student through various lessons and examples that:

$$\lim_{x \rightarrow 8} \frac{1}{x-8} = \infty$$

I tried to check if she really understood that, so I gave her a different example. This was the result:

$$\lim_{x \rightarrow 5} \frac{1}{x-5} = 5$$

# Big-Oh Examples

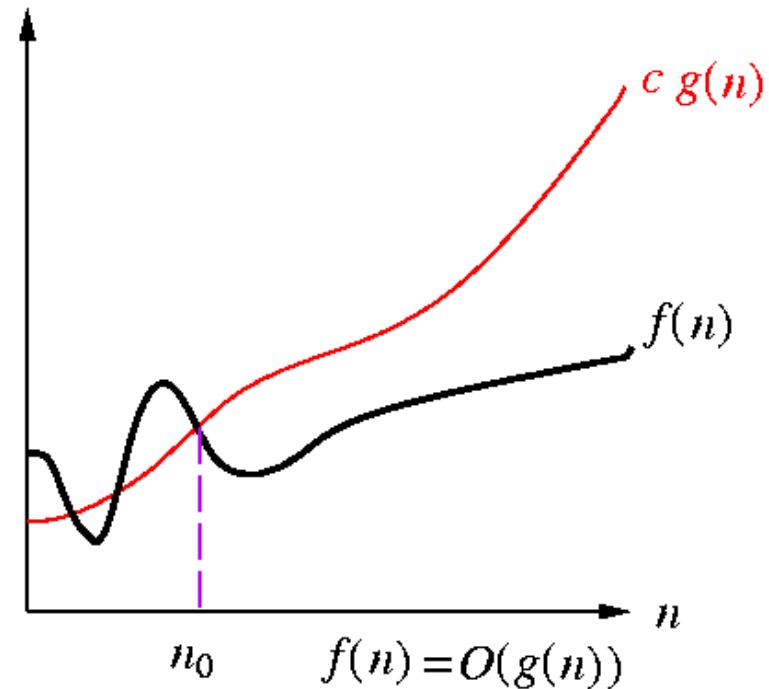
- **Def:**  $f(n) = O(g(n))$  if  $\exists c > 0$  and  $n_0 > 0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$ .

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

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

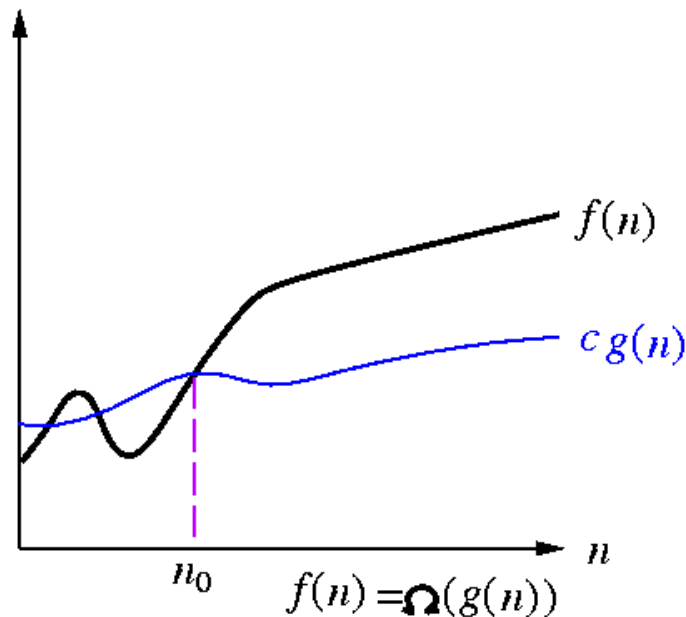
$$\text{Take } c = 4, n_0 = 1$$

$f(n) = O(g(n))$  implies that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  for some  $c \geq 0$ , if the limit exists.



# $\Omega$ : Lower Bounding Function

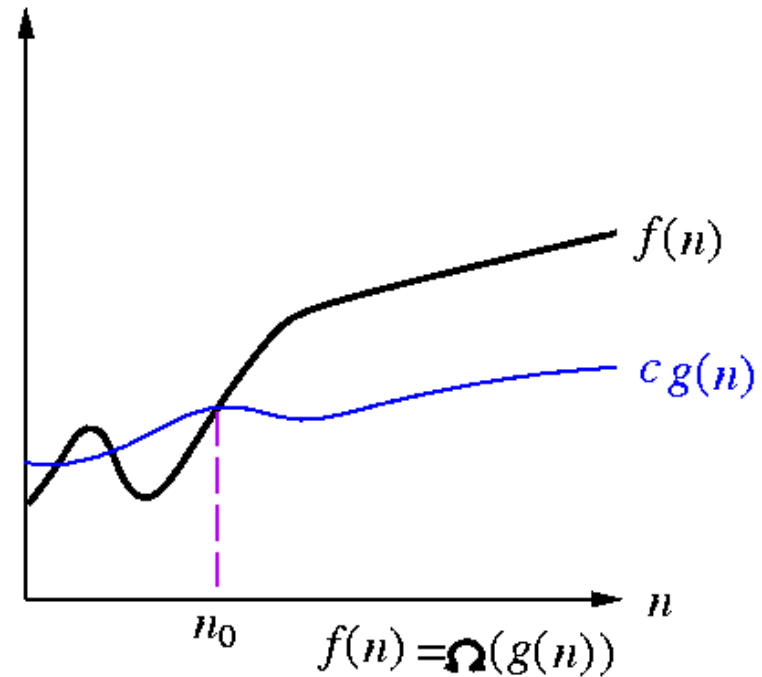
- **Def:**  $f(n) = \Omega(g(n))$  if  $\exists c > 0$  and  $n_0 > 0$  such that  $0 \leq cg(n) \leq f(n)$  for all  $n \geq n_0$ .
- Intuition:  $f(n)$  “ $\geq$ ”  $g(n)$  when we ignore constant multiples and small values of  $n$ .
- How to show  $\Omega$  (Big-Omega) relationships?
  - $f(n) = \Omega(g(n))$  implies that  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = c$  for some  $c \geq 0$ , if the limit exists.



# Big-Omega Examples

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

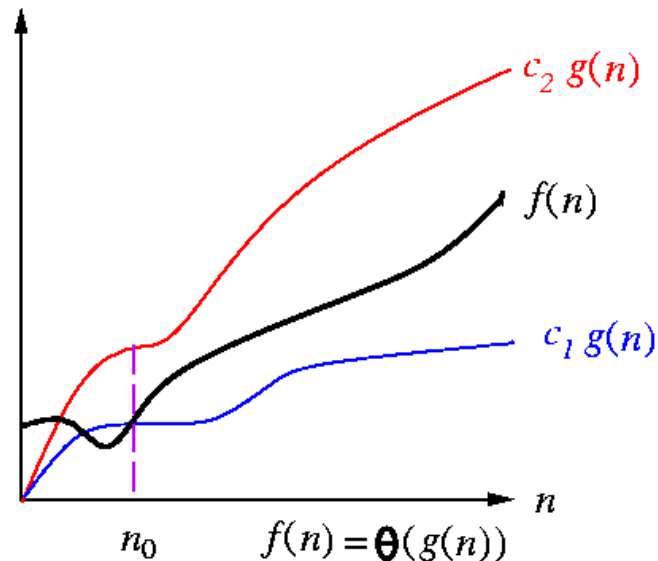
1.  $3n^2 + n = \Omega(n^2)$ ? **Yes**
2.  $3n^2 + n = \Omega(n)$ ? **Yes**
3.  $3n^2 + n = \Omega(n^3)$ ? **No**



$f(n) = \Omega(g(n))$  implies that  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = c$  for some  $c \geq 0$ , if the limit exists.

# $\theta$ : Tightly Bounding Function

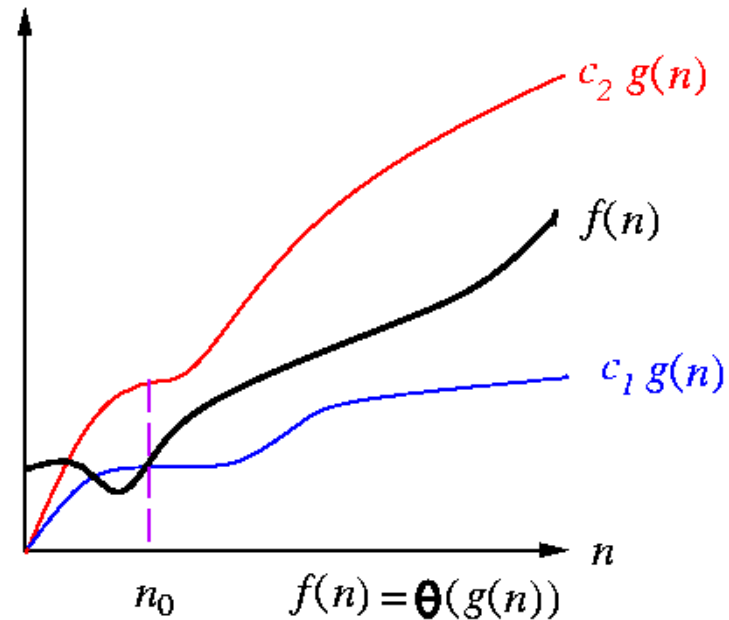
- **Def:**  $f(n) = \theta(g(n))$  if  $\exists c_1, c_2 > 0$  and  $n_0 > 0$  such that  $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$  for all  $n \geq n_0$ .
- Intuition:  $f(n)$  “=”  $g(n)$  when we ignore constant multiples and small values of  $n$ .
- How to show  $\theta$  relationships?
  - Show both “big Oh” ( $O$ ) and “Big Omega” ( $\Omega$ ) relationships.
  - $f(n) = \theta(g(n))$  implies that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  for some  $c > 0$ , 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  $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$  for all  $n \geq n_0$ .

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



$f(n) = \theta(g(n))$  implies that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$  for some  $c > 0$ , if the limit exists.

## **$o, \omega$ : Untightly Upper, Lower Bounding Functions**

---

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

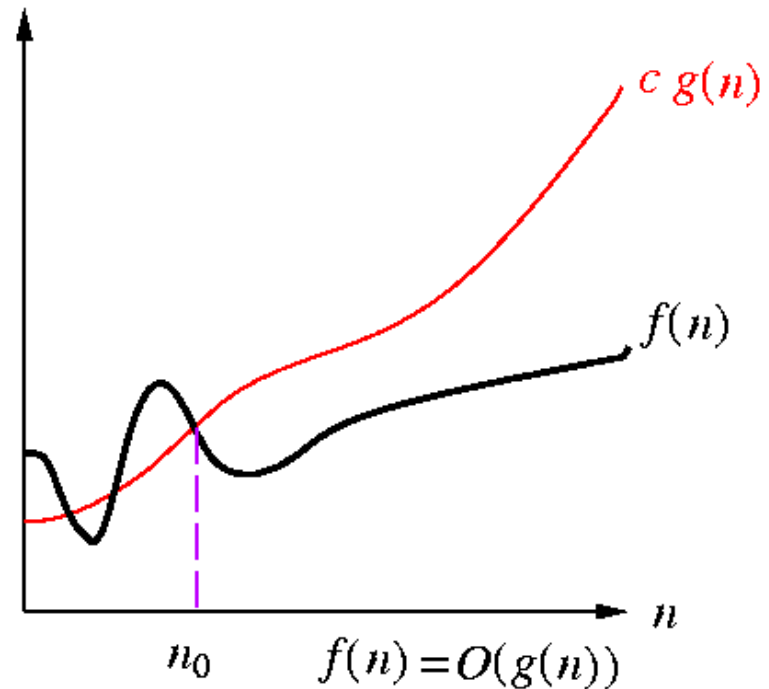
# Little-Oh Examples

- **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**

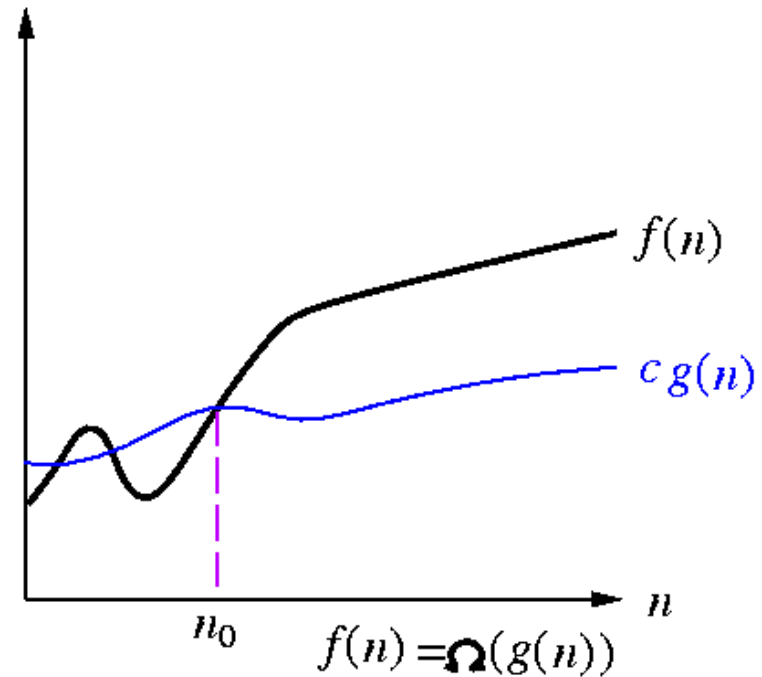


$f(n) = o(g(n))$  implies that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , if the limit exists

# Little-Omega Examples

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

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



$f(n) = \omega(g(n))$  implies that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ , if the limit exists

# 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.

# Properties for Asymptotic Analysis

---

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

# Asymptotic Functions

- $\lg^{(i)} n = \underbrace{\lg \lg \dots \lg n}_i$ . (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
$\lg^{O(1)} n = \underbrace{\lg \lg \dots \lg n}_{O(1)}$	–
$\lg n$	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
$2^n, 3^n, \dots$	exponential
$n!$	factorial
$n^n$	–

**Polynomial  
-time  
complexity!!**

# Computational Complexity

- **Computational complexity**: an abstract measure of the **time** and **space** necessary to execute an algorithm as functions of its “input size”.
- Input size: size of encoded “binary” strings.
  - sort  $n$  words of bounded length  $\Rightarrow$  input size:  $n$
  - **the input is the integer  $n \Rightarrow$  input size:  $\lg n$**
  - the input is the graph  $G(V, E) \Rightarrow$  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	$O(1)$	$5 \cdot 10^{-7}$ sec	$5 \cdot 10^{-7}$ sec	$5 \cdot 10^{-7}$ sec	$5 \cdot 10^{-7}$ sec	$5 \cdot 10^{-7}$ sec
$3n$	$O(n)$	$3 \cdot 10^{-8}$ sec	$3 \cdot 10^{-7}$ sec	$3 \cdot 10^{-5}$ sec	0.003 sec	0.3 sec
$n \lg n$	$O(n \lg n)$	$3 \cdot 10^{-8}$ sec	$6 \cdot 10^{-7}$ sec	$1 \cdot 10^{-4}$ sec	0.018 sec	2.5 sec
$n^2$	$O(n^2)$	$1 \cdot 10^{-7}$ sec	$1 \cdot 10^{-5}$ sec	0.1 sec	16.7 min	116 days
$n^3$	$O(n^3)$	$1 \cdot 10^{-6}$ sec	0.001 sec	16.7 min	31.7 yr	$\infty$
$2^n$	$O(2^n)$	$1 \cdot 10^{-6}$ sec	$4 \cdot 10^{11}$ cent.	$\infty$	$\infty$	$\infty$
$n!$	$O(n!)$	0.003 sec	$\infty$	$\infty$	$\infty$	$\infty$

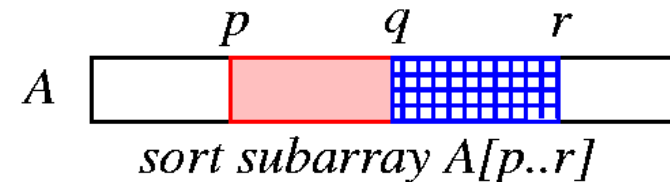
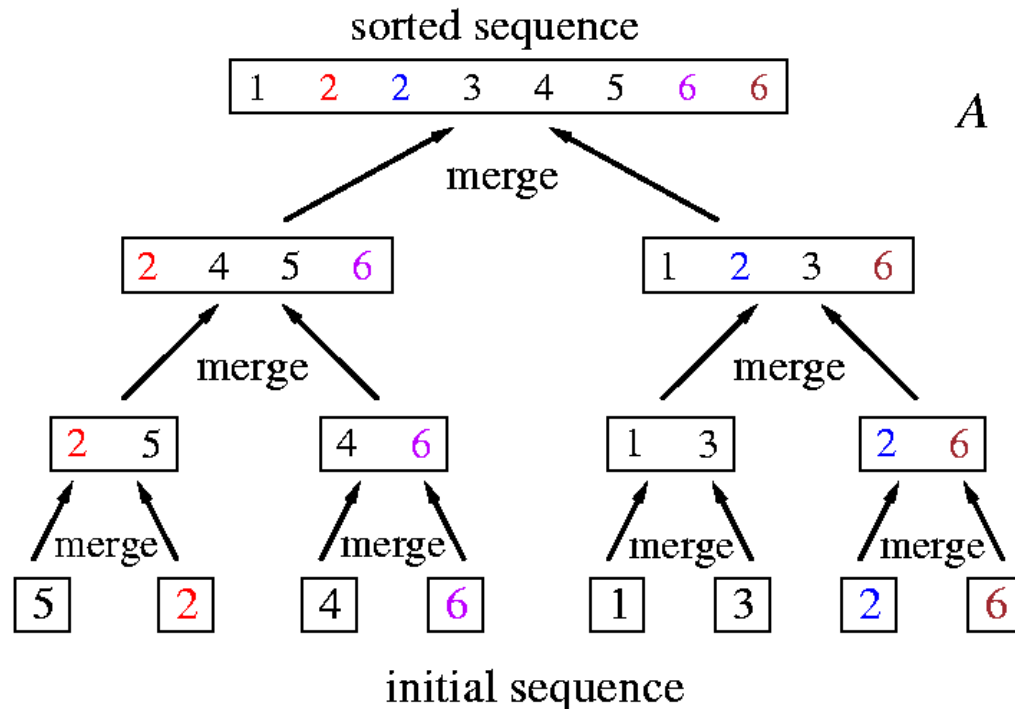
# Divide-and-Conquer Algorithms Revisited

---

- The divide-and-conquer paradigm
  - **Divide** the problem into a number of subproblems.
  - **Conquer** the subproblems (solve them).
  - **Combine** the subproblem solutions to get the solution to the original problem.
- Merge sort:  $T(n) = 2T(n/2) + \theta(n) = \theta(n \lg n)$ .
  - **Divide** the  $n$ -element sequence to be sorted into two  $n/2$ -element sequences.
  - **Conquer**: sort the subproblems, recursively using merge sort.
  - **Combine**: merge the resulting two sorted  $n/2$ -element sequences.

# Merge Sort Revisited

MergeSort( $A, p, r$ )	$T(n)$
1. <b>if</b> $p < r$ <b>then</b>	$\theta(1)$
2. $q \leftarrow \lfloor (p+r)/2 \rfloor$	$\theta(1)$
3.   MergeSort ( $A, p, q$ )	$T(n/2)$
4.   MergeSort ( $A, q+1, r$ )	$T(n/2)$
5.   Merge( $A, p, q, r$ )	$\theta(n)$



Merge( $A, p, q, r$ )  
 merges two **sorted**  
 subarrays  $A[p..q]$   
 and  $A[q+1..r]$  into  
 $A[p..r]$ .

# Analyzing Divide-and-Conquer Algorithms

---

- Recurrence for a divide-and-conquer algorithms

$$T(n) = \begin{cases} \theta(1), & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n), & \text{otherwise} \end{cases}$$

- $a$ : # of subproblems
- $n/b$ : size of the subproblems
- $D(n)$ : time to divide the problem of size  $n$  into subproblems
- $C(n)$ : time to combine the subproblem solutions to get the answer for the problem of size  $n$

- Merge sort:

$$T(n) = \begin{cases} \theta(1), & \text{if } n \leq c \\ 2T(n/2) + \theta(n), & \text{otherwise} \end{cases}$$

- $a = 2$ : two subproblems
- $n/b = n/2$ : each subproblem has size  $\approx n/2$
- $D(n) = \theta(1)$ : compute midpoint of array
- $C(n) = \theta(n)$ : merging by scanning sorted subarrays

# Divide-and-Conquer: Binary Search

---

- Binary search on a **sorted** array:
  - **Divide:** Check middle element.
  - **Conquer:** Search the subarray.
  - **Combine:** Trivial.
- Recurrence:  $T(n) = T(n/2) + \theta(1) = \theta(\lg n)$ .

$$T(n) = \begin{cases} \theta(1), & \text{if } n \leq c \\ T(n/2) + \theta(1), & \text{otherwise} \end{cases}$$

- $a = 1$ : search one subarray
- $n/b = n/2$ : each subproblem has size  $\approx n/2$
- $D(n) = \theta(1)$ : compute midpoint of array
- $C(n) = \theta(1)$ : trivial

# Solving Recurrences

---

- Three general methods for solving recurrences
  - **Iteration:** Convert the recurrence into a summation by expanding some terms and then bound the summation
  - **Substitution:** Guess a solution and verify it by induction.
  - **Master Theorem:** if the recurrence has the form
$$T(n) = aT(n/b) + f(n),$$
then **most likely** there is a formula that can be applied.
- Two **simplifications** that won't affect asymptotic analysis
  - Ignore floors and ceilings.
  - Assume base cases are constant, i.e.,  $T(n) = \theta(1)$  for small  $n$ .

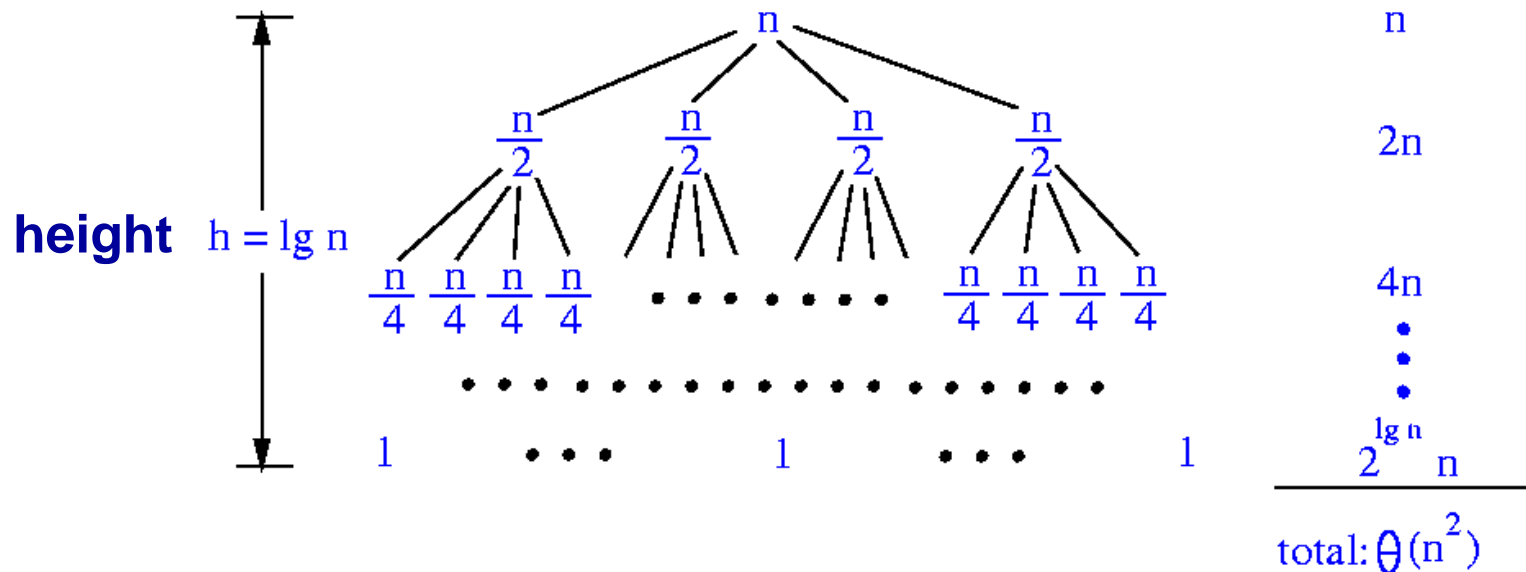
# Solving Recurrences: Iteration

- **Example:**  $T(n) = 4T(n/2) + n$ .

$$\begin{aligned} T(n) &= 4T(n/2) + n && /* expand */ \\ &= 4(4T(n/4) + n/2) + n && /* simplify */ \\ &= 16T(n/4) + 2n + n && /* expand */ \\ &= 16(4T(n/8) + n/4) + 2n + n && /* simplify */ \\ &= 64T(n/8) + 4n + 2n + n && /* #level = lg n */ \\ &= 4^{\lg n} T(1) + \dots + 4n + 2n + n && /* convert to summation */ \\ &= 4^{\lg n} c + n \sum_{k=0}^{\lg n - 1} 2^k && /* a^{\lg b} = b^{\lg a} */ \\ &= cn^{\lg 4} + n \left( \frac{2^{\lg n} - 1}{2 - 1} \right) && /* 2^{\lg n} = n^{\lg 2} */ \\ &= cn^2 + n(n^{\lg 2} - 1) \\ &= (c + 1)n^2 - n \\ &= \Theta(n^2) \end{aligned}$$

# Iteration by Using Recursion Trees

- Root: computation ( $D(n) + C(n)$ ) at top level of recursion.
- Node at level  $i$ : Subproblem at level  $i$  in the recursion.
- Height of tree: #level in the recursion.
- $T(n)$  = sum of all nodes in the tree.
- $T(1) = 1 \Rightarrow T(n) = 4T(n/2) + n = n + 2n + 4n + \dots + 2^{\lg n}n = \theta(n^2)$ .



# Solving Recurrences: Substitution (Guess & Verify)

---

1. Guess form of a solution.
2. Apply math. induction to find the constant and verify the solution.
3. Use to find an upper or a lower bound.

- **Example:** Guess  $T(n) = 4T(n/2) + n = O(n^3)$  ( $T(1) = 1$ )

- Show  $T(n) \leq cn^3$  for some  $c > 0$  (**we must find c**).

1. Basis:  $T(2) = 4T(1) + 2 = 6 \leq 2^3c$  (pick  $c = 1$ )

2. Assume  $T(k) \leq ck^3$  for  $k < n$ , and prove  $T(n) \leq cn^3$

$$\begin{aligned}T(n) &= 4 T(n/2) + n \\ &\leq 4 (c (n/2)^3) + n \\ &= cn^3/2 + n \\ &= cn^3 - (cn^3/2 - n) \\ &\leq cn^3,\end{aligned}$$

where  $c \geq 2$  and  $n \geq 1$ . (**Pick  $c \geq 2$  for Steps 1 & 2!**)

- **Useful tricks:** subtract a lower order term, change variables (e.g.,  $T(n) = T(\sqrt{n}) + \lg n$ )

## Pitfall in Substitution

---

- **Example:** Guess  $T(n) = 2T(n/2) + n = O(n)$  (wrong guess!)

– Show  $T(n) \leq cn$  for some  $c > 0$  (we must find  $c$ ).

1. Basis:  $T(2) = 2T(1) + 2 = 4 \leq 2c$  (pick  $c = 2$ )

2. Assume  $T(k) \leq ck$  for  $k < n$ , and prove  $T(n) \leq cn$

$$T(n) = 2T(n/2) + n$$

$$\leq 2(cn/2) + n$$

$$= cn + n$$

$$= O(n)? \quad /* \text{Wrong!!} */$$

- What's wrong?
- How to fix? Subtracting a lower-order term may help!

# Fixing Wrong Substitution

---

- Guess  $T(n) = 4T(n/2) + n = O(n^2)$  (right guess!)
  - Assume  $T(k) \leq ck^2$  for  $k < n$ , and prove  $T(n) \leq cn^2$ .

$$\begin{aligned}T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^2 + n \\ &= \mathbf{cn^2 + n} \\ &= \mathbf{O(n^2)} \quad \mathbf{/* Wrong!! */}\end{aligned}$$

- Fix by subtracting a lower-order term.
  - Assume  $T(k) \leq c_1k^2 - c_2k$  for  $k < n$ , and prove  $T(n) \leq c_1n^2 - c_2n$ .

$$\begin{aligned}T(n) &= 4T(n/2) + n \\ &\leq 4(c_1(n/2)^2 - c_2(n/2)) + n \\ &= \mathbf{c_1n^2 - 2c_2n + n} \\ &\leq \mathbf{c_1n^2 - c_2n} \quad \mathbf{(if\ c_2 \geq 1)}\end{aligned}$$

- Pick  $c_1$  big enough to handle initial conditions.

# Master Theorem

---

- Let  $a \geq 1$  and  $b > 1$  be constants,  $f(n)$  be a function, and  $T(n)$  be defined on nonnegative integers as

$$T(n) = aT(n/b) + f(n).$$

- Then,  $T(n)$  can be bounded asymptotically as follows:
  1.  $T(n) = \Theta(n^{\log_b a})$  if  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ .
  2.  $T(n) = \Theta(n^{\log_b a} \lg n)$  if  $f(n) = \Theta(n^{\log_b a})$ .
  3.  $T(n) = \Theta(f(n))$  if  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$  and  $af(n/b) \leq cf(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ .
- Intuition: compare  $f(n)$  with  $\Theta(n^{\log_b a})$ .
  - Case 1:  $f(n)$  is **polynomially smaller** than  $\Theta(n^{\log_b a})$ .
  - Case 2:  $f(n)$  is **asymptotically equal** to  $\Theta(n^{\log_b a})$ .
  - Case 3:  $f(n)$  is **polynomially larger** than  $\Theta(n^{\log_b a})$ .

# Solving Recurrences Using the Master Method

---

- **Ex. 1:**  $T(n) = 4T\left(\frac{n}{2}\right) + n$ 
  - Compare  $f(n) = n$  with  $n^{\log_b a} = n^2$ :  $f(n) = n = O(n^{2-\epsilon})$ .
  - Case 1 applies:  $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$ .
- **Ex. 2:**  $T(n) = 2T\left(\frac{n}{2}\right) + n$ 
  - Compare  $f(n) = n$  with  $n^{\log_b a} = n$ :  $f(n) = n = \Theta(n)$ .
  - Case 2 applies:  $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n)$ .

# Solving Recurrences Using the Master Method

---

- **Ex. 3:**  $T(n) = 3T(\frac{n}{4}) + n \lg n$ 
  - Compare  $f(n) = n \lg n$  with  $n^{\log_b a} = n^{0.79}$ :  $f(n) = n \lg n = \Omega(n^{0.79+\epsilon})$ .
  - Case 3 **could** apply: Need to check for “regularity” condition that  $af(n/b) \leq cf(n)$ .
    - \* Find  $c < 1$  s.t.  $af(n/b) \leq cf(n)$  for large enough  $n$ .
    - \*  $3\frac{n}{4} \lg \frac{n}{4} \leq cn \lg n$  which is true for  $c = \frac{3}{4}$ .
  - Case 3 applies:  $T(n) = \Theta(f(n)) = \Theta(n \lg n)$ .
- **Ex. 4:**  $T(n) = 4T(\frac{n}{2}) + \frac{n^2}{\lg n}$ ?? (Ans:  $T(n) = \Theta(n^2 \lg \lg n)$ ; Hint: harmonic series  $\sum_{i=1}^n 1/i = \Theta(\lg n)$ .)