# Design and Analysis of Algorithms

CSE 5311

Lecture 1 Administration & Introduction

Song Jiang, Ph.D.

Department of Computer Science and Engineering

# Administration

- **Course CSE 5311**
  - What: Design and Analysis of Algorithms
  - When: 12:30-1:50pm Tu/Th
  - Where: NH 100
  - Who: Song Jiang (song.jiang@uta.edu)
  - Office Hours: 2:20-3:20pm Tu/Th at SIER 319 (or online by appointment)
  - TA: Mr. Sujit Maharjan (sxm5754@mavs.uta.edu)

    Office Hours: 9-10am Tu/Th at SEIR 322KK or online via Teams

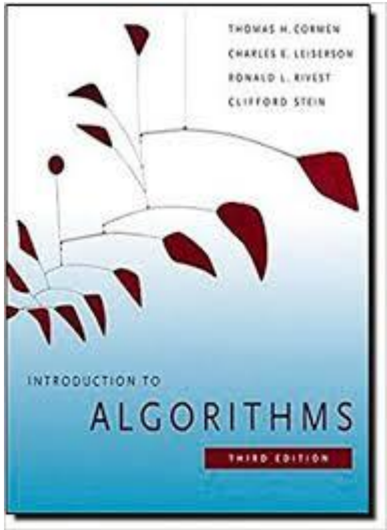
- **About your instructor**
  - Research areas: file and storage system, operating system, parallel and distributed computing, and high-performance computing,

# Study Materials

- **Prerequisites**
  - CSE 2320 Algorithms and Data Structures or its equivalents
  - Programming skills on a high-level language, such as C and Java.
  - Mathematical background on summations, sets, relations, probability, and matrix computation.

- **Textbook**

Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. **Introduction to Algorithms**. 3rd ed. MIT Press, 2009.



# Grading

- **Distribution**
  - 5% Class attendance
  - 30% Homework Assignments
  - 20% Quizzes
  - 20% Midterm Exam
  - 25% Final Exam

  ---
  100%
- Bonus credits may be offered for voluntarily and correctly answering in-class questions.
- **Attention**
  - Attendance is required.
  - The university makeup policy will be strictly adhered to. Generally, no make-up exams/quizzes except for university sanctioned reasons.
  - Late assignments are not accepted.

# Grading

- **Makeup Exams**

  No make-up exams will be given except for university sanctioned excused absences. If you will miss an exam (for a good reason), it is your responsibility to contact instructor before the exam.

-

- **Late Assignments**

  Late assignments are not accepted.

# Grading

- **Collaboration Policy**

Students are allowed and encouraged to collaborate on homework assignments. However, **You must write up each problem solution by yourself without assistance**, even if you collaborate with others to solve the problem. If you obtain a solution through research (e.g., on the Web), acknowledge your source, and write up the solution in your own words. **It is a violation of this policy to submit a problem solution that you cannot orally explain to the instructor or GTA** with a penalty of losing all credit points of the assignment.

# Final Grade

- **Final Letter Grade**
  - [90 100] --- A
  - [80 90) --- B
  - [70 80) --- C
  - [60 70) --- D
  - [00 60) --- F
- **Note**
  - [ ] denotes inclusion and ( ) denotes exclusion.
  - Your final weighted scores may be curved for assignment of your letter grade.

# What's the Course About?

- **The theoretical study of analysis and design of computer algorithms**
  - **Analysis:** predict the cost of an algorithm in terms of resources and performance
  - **Design:** design algorithms which minimize the cost
- **Basic goals for an algorithm**
  - Always correct
  - Always terminates
- **Our class: performance**

# Algorithms

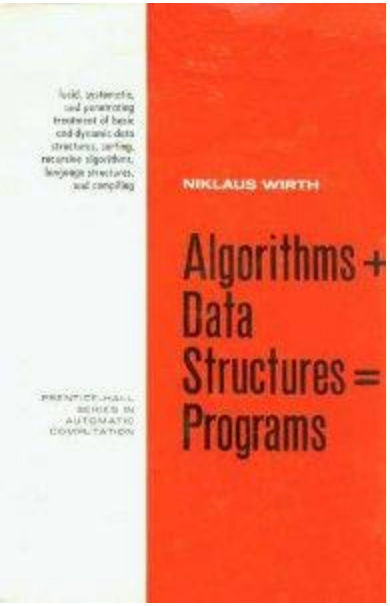
- An ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***.

- An example problem: sorting

  **Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

  **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

  **An instance of a problem:** <31; 41; 59; 26; 41; 58>



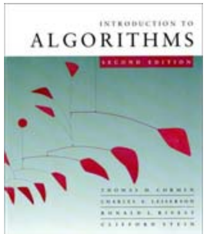

- Design algorithms for a problem:
  - Find **a** longest common subsequence of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$

# Why study algorithms and performance?

- 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.
- Performance is the ***currency*** of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

# Machine Model

- **Generic Random Access Machine (RAM)**
  - Executes operations sequentially
  - Set of primitive operations: Arithmetic. Logical, Comparisons, Function calls
- **Simplifying assumption**
  - All operations cost one unit
  - Eliminates dependence on the speed of our computer
  - Otherwise impossible to verify and to compare

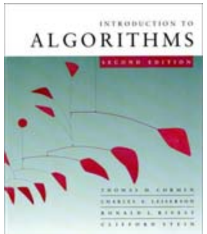



# Analysis of algorithms

*The theoretical study of computer-program performance and resource usage.*

What's more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability




# The problem of sorting

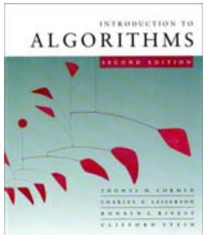
***Input:*** sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

***Output:*** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

**Example:**

***Input:*** 8 2 4 9 3 6

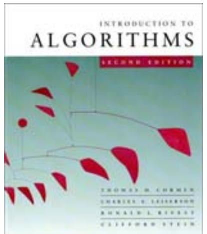
***Output:*** 2 3 4 6 8 9




# Insertion sort

"pseudocode"

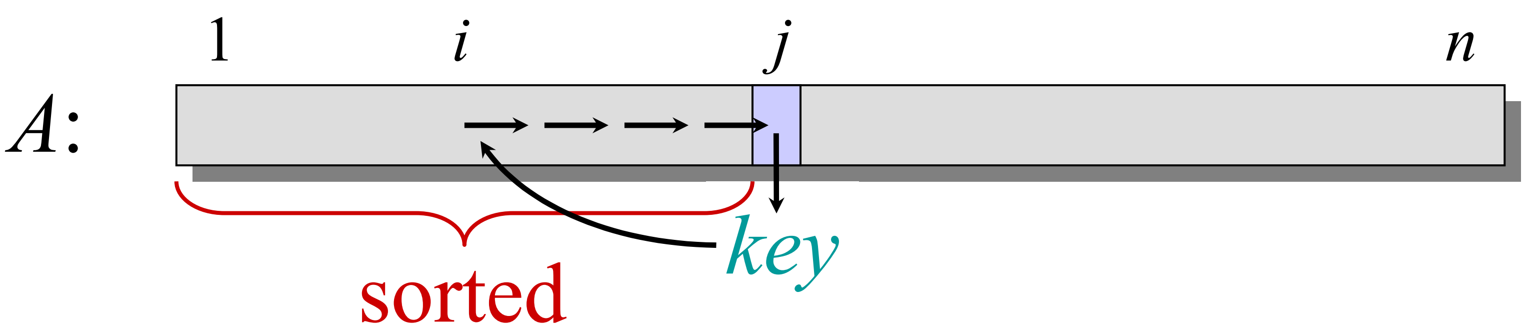
```
INSERTION-SORT (A, n)      ⊳ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```




# Insertion sort

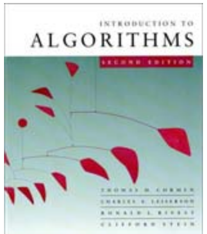
"pseudocode"

```
INSERTION-SORT (A, n)        ⊳ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



# Example of insertion sort

8 2 4 9 3 6




# Example of insertion sort

8 2 4 9 3 6

# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | 3 | 6 |

# Example of insertion sort

8 2 4 9 3 6

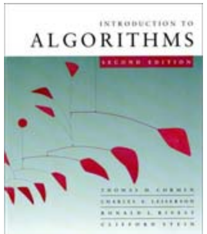
2 8 4 9 3 6

# Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

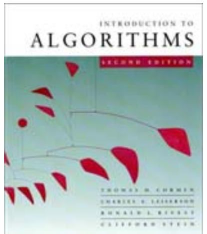



# Example of insertion sort

8 2 4 9 3 6

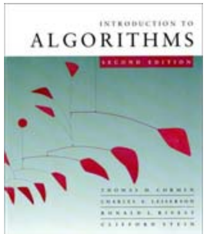
2 8 4 9 3 6

2 4 8 9 3 6



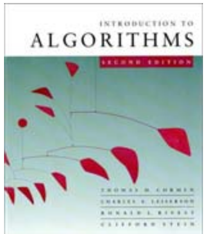

# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |
| 2 | 4 | 8 | 9 | 3 | 6 |




# 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 |




# Example of insertion sort

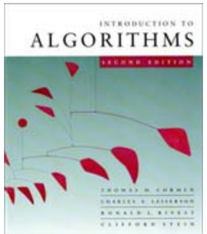
8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

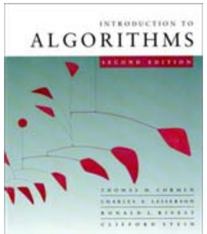
2 4 8 9 3 6

2 3 4 8 9 6




# Example of insertion sort

| | | | | | |
|---|---|---|---|---|---|
| 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 |

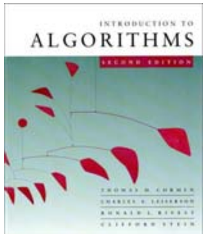



# Example of insertion sort

| | | | | | | |
|---|---|---|---|---|---|---|
| 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* |

# Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.




# Kinds of analyses

**Worst-case:** (usually)

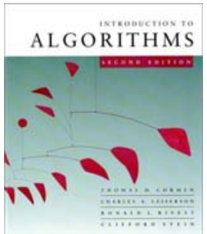
- $T(n)$ = maximum time of algorithm on any input of size $n$.

**Average-case:** (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size $n$.
- Need assumption of statistical distribution of inputs.

**Best-case:** (bogus)

- Cheat with a slow algorithm that works fast on *some* input.




# Machine-independent time

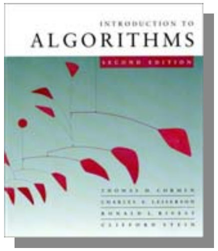
*What is insertion sort's worst-case time?*

- It depends on the speed of our computer:
    - relative speed (on the same machine),
    - absolute speed (on different machines).

**BIG IDEA:**

- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \to \infty$ .

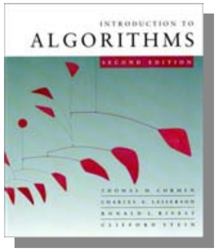
**"Asymptotic Analysis"**




# Asymptotic notation

$O$-notation (upper bounds):

> We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

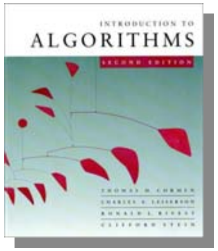



# Asymptotic notation

$O$-notation (upper bounds):

> We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

**Example:** $2n^2 = O(n^3)$ $\quad$ $(c = 1, n_0 = 2)$



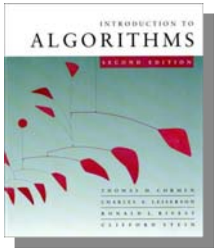

# Asymptotic notation

$O$-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

**EXAMPLE:** $2n^2 = O(n^3)$ $\quad (c = 1, n_0 = 2)$

*functions, not values*




# Asymptotic notation

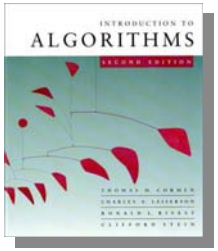
*O*-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

**EXAMPLE:** $2n^2 = O(n^3)$ $\quad (c = 1, n_0 = 2)$

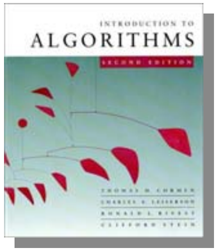
*functions, not values*

*funny, "one-way" equality*




# Set definition of O-notation

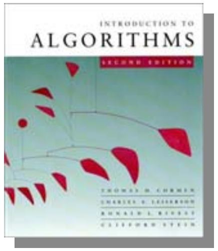
$O(g(n)) = \{ f(n)$ : there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$ $\}$




# Set definition of O-notation

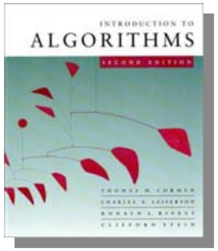
$O(g(n)) = \{ f(n) :$ there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$ $\}$

**EXAMPLE:** $2n^2 \in O(n^3)$




# Macro substitution

***Convention:*** A set in a formula represents an anonymous function in the set.




# Macro substitution

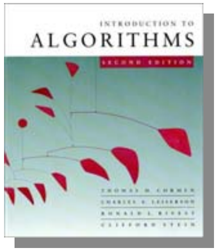
***Convention:*** A set in a formula represents an anonymous function in the set.

**EXAMPLE:** $f(n) = n^3 + O(n^2)$

means

$f(n) = n^3 + h(n)$

for some $h(n) \in O(n^2)$ .




# Macro substitution

***Convention:*** A set in a formula represents an anonymous function in the set.

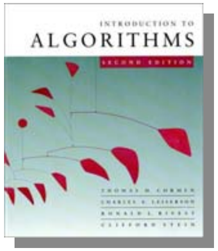
**EXAMPLE:** $n^2 + O(n) = O(n^2)$

means

for any $f(n) \in O(n)$:

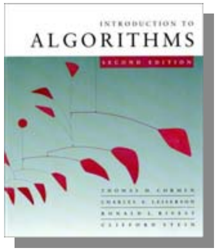
$$n^2 + f(n) = h(n)$$

for some $h(n) \in O(n^2)$ .




# Ω-notation (lower bounds)

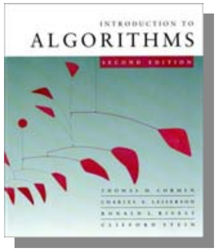
*O*-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.




# Ω-notation (lower bounds)

*O*-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.

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

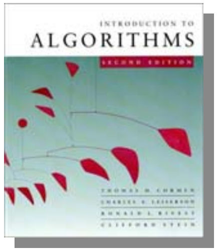



# Ω-notation (lower bounds)

*O*-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.

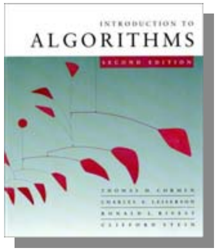
$$\Omega(g(n)) = \{ f(n) : \text{there exist constants } c > 0,\ n_0 > 0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \}$$

**EXAMPLE:** $\sqrt{n} = \Omega(\lg n)$ $(c = 1,\ n_0 = 16)$




# Θ-notation (tight bounds)

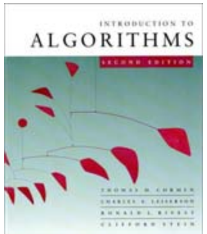
$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$




# Θ-notation (tight bounds)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

**EXAMPLE:** $\frac{1}{2}n^2 - 2n = \Theta(n^2)$



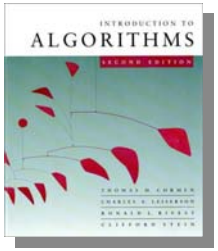

# Θ-notation

***Math:***

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

***Engineering:***

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$



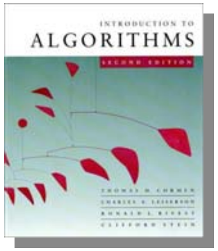

# $o$-notation and $\omega$-notation

$O$-notation and $\Omega$-notation are like $\leq$ and $\geq$.
$o$-notation and $\omega$-notation are like $<$ and $>$.

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

**EXAMPLE:** $2n^2 = o(n^3)$ $\quad (n_0 = 2/c)$

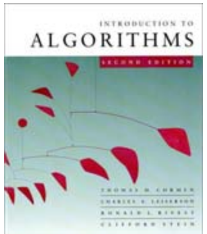
  




# $o$-notation and $\omega$-notation

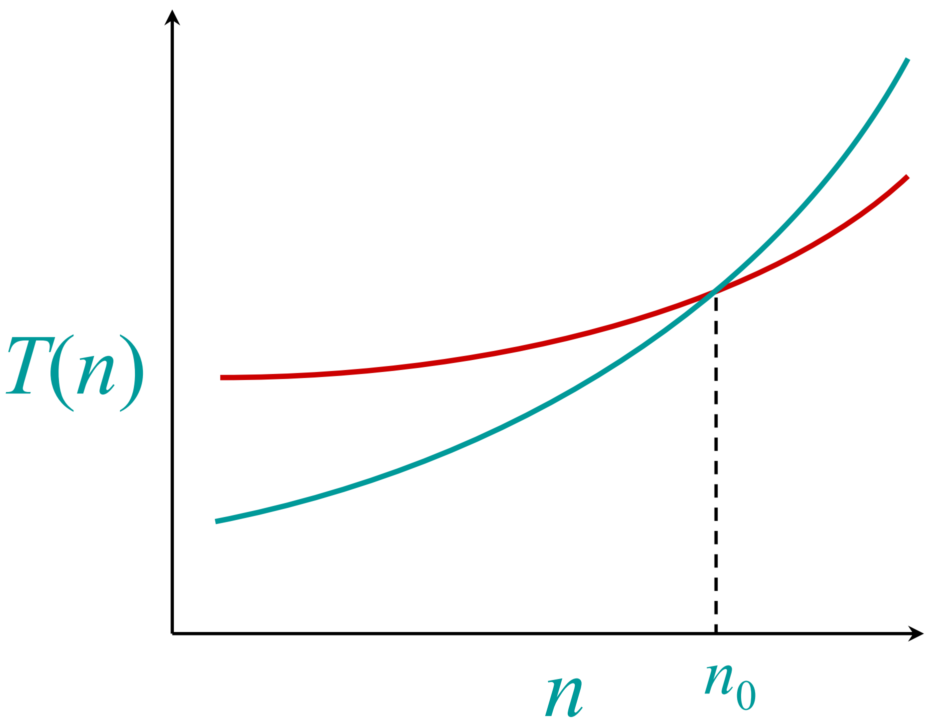
$O$-notation and $\Omega$-notation are like $\leq$ and $\geq$.
$o$-notation and $\omega$-notation are like $<$ and $>$.

$\omega(g(n)) = \{ f(n)$ : for any constant $c > 0$,
there is a constant $n_0 > 0$
such that $0 \leq cg(n) < f(n)$
for all $n \geq n_0$ $\}$

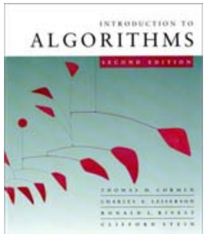



# Asymptotic performance

When $n$ gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.




- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.




# Insertion sort analysis

***Worst case:*** Input reverse sorted.

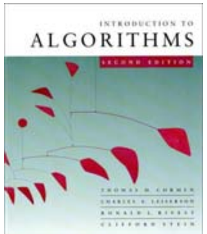
$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta\left(n^2\right) \qquad \text{[arithmetic series]}$$

***Average case:*** All permutations equally likely.

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

*Is insertion sort a fast sorting algorithm?*

- Moderately so, for small $n$.
- Not at all, for large $n$.

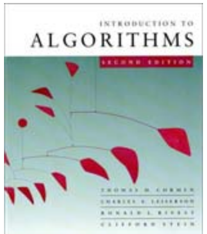



# Merge sort

MERGE-SORT $A[1 \mathinner{.\,.} n]$

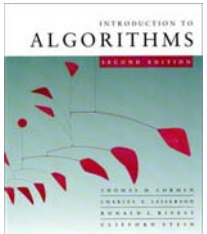
1. If $n = 1$, done.
2. Recursively sort $A[\,1 \mathinner{.\,.} \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \mathinner{.\,.} n\,]$ .
3. "***Merge***" the 2 sorted lists.

***Key subroutine:*** **MERGE**




# Merging two sorted arrays

| | |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |




# Merging two sorted arrays

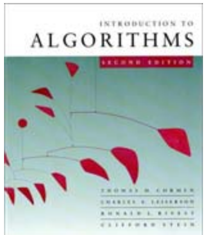
20 12

13 11

7 9

2 1

1




# Merging two sorted arrays

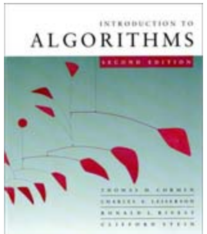
| | | | | |
|---|---|---|---|---|
| 20 | 12 | ‖ | 20 | 12 |
| 13 | 11 | ‖ | 13 | 11 |
| 7 | 9 | ‖ | 7 | 9 |
| 2 | 1 | ‖ | 2 | |

1

# Merging two sorted arrays

| | | | | |
|---|---|---|---|---|
| 20 | 12 | ‖ | 20 | 12 |
| 13 | 11 | ‖ | 13 | 11 |
| 7 | 9 | ‖ | 7 | 9 |
| 2 | 1 | ‖ | 2 | |

1          2

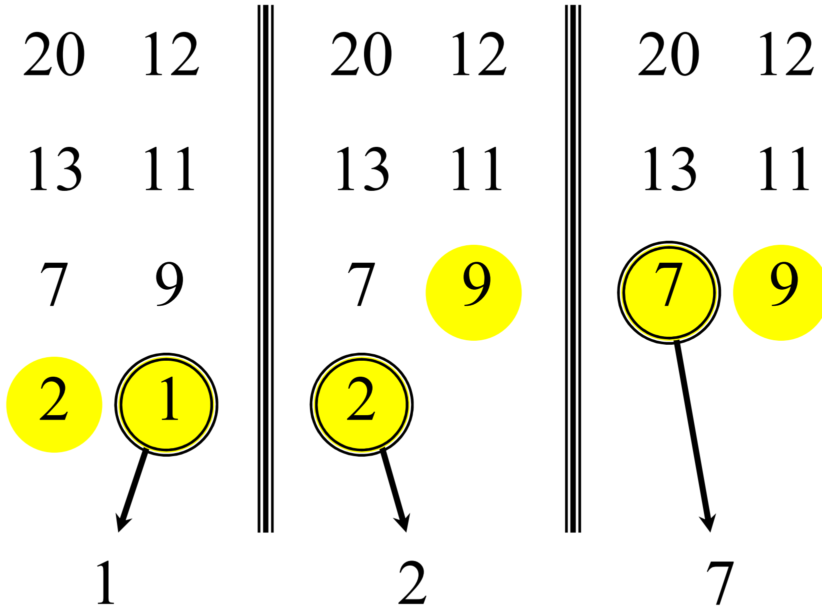


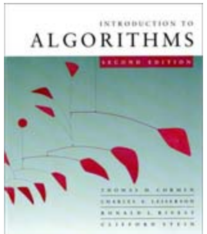

# Merging two sorted arrays

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 20 | 12 | ‖ | 20 | 12 | ‖ | 20 | 12 |
| 13 | 11 | ‖ | 13 | 11 | ‖ | 13 | 11 |
| 7 | 9 | ‖ | 7 | 9 | ‖ | 7 | 9 |
| 2 | 1 | ‖ | 2 | | ‖ | | |

1 2

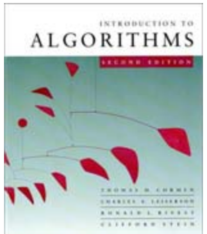
# Merging two sorted arrays



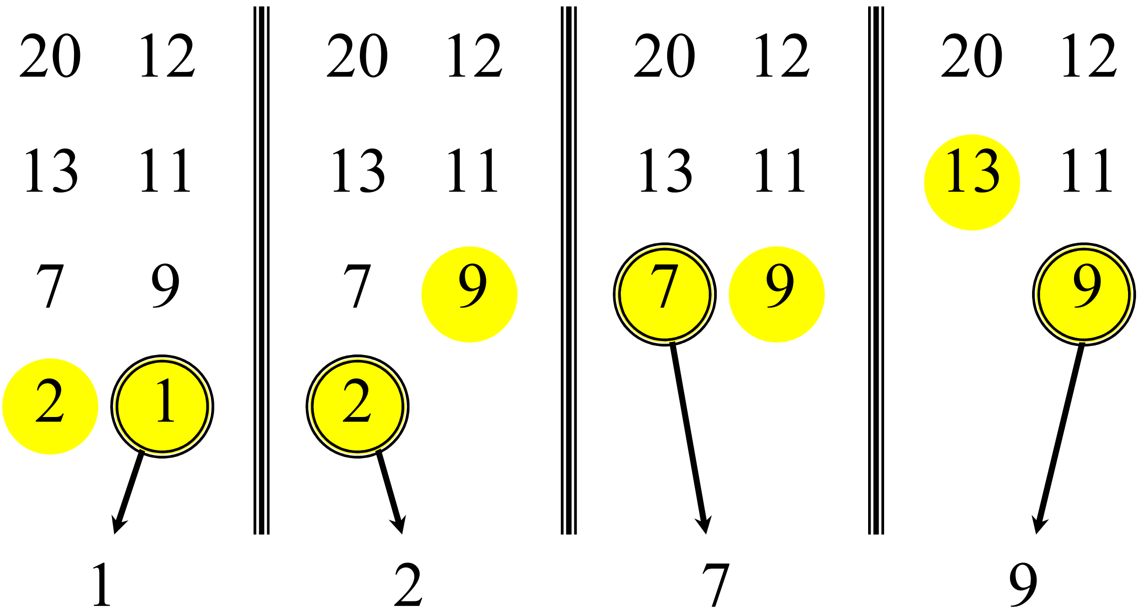



# Merging two sorted arrays

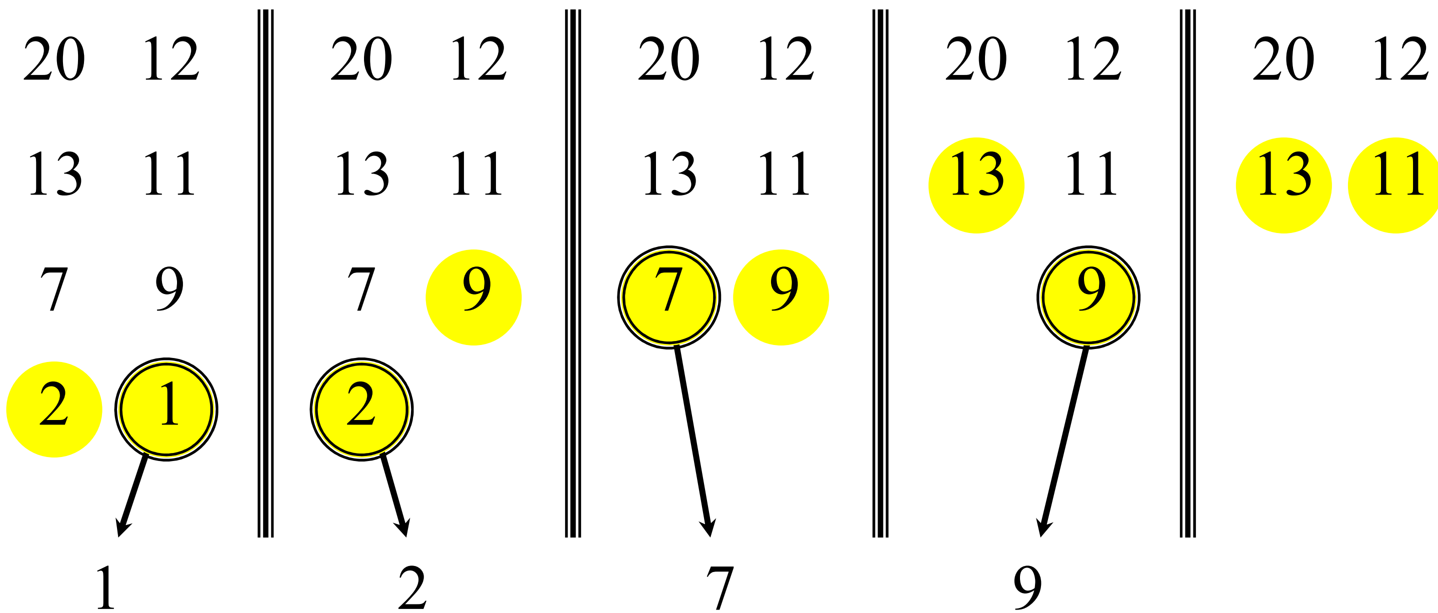
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 20 | 12 | 20 | 12 | 20 | 12 | 20 | 12 |
| 13 | 11 | 13 | 11 | 13 | 11 | 13 | 11 |
| 7 | 9 | 7 | 9 | 7 | 9 | | 9 |
| 2 | 1 | 2 | | | | | |
| | 1 | 2 | | | 7 | | |

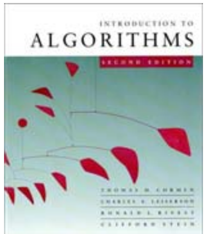



# Merging two sorted arrays

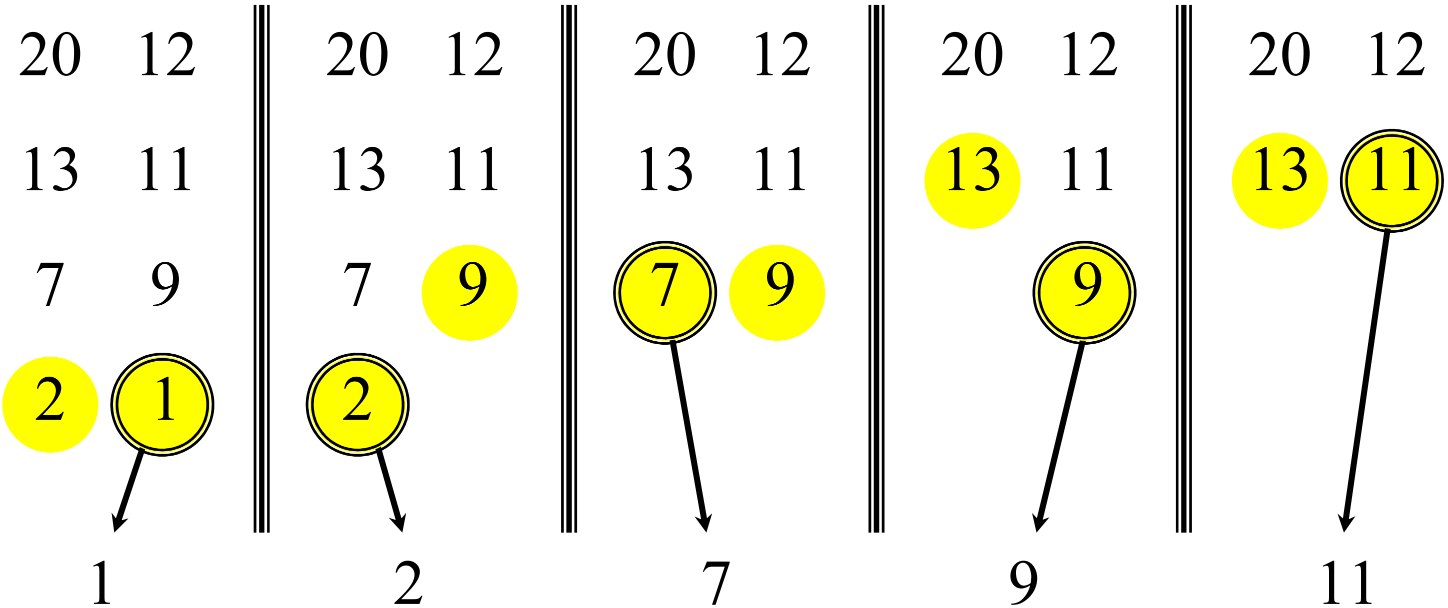


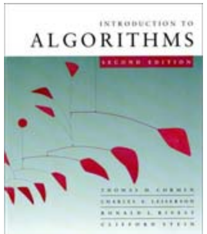
# Merging two sorted arrays



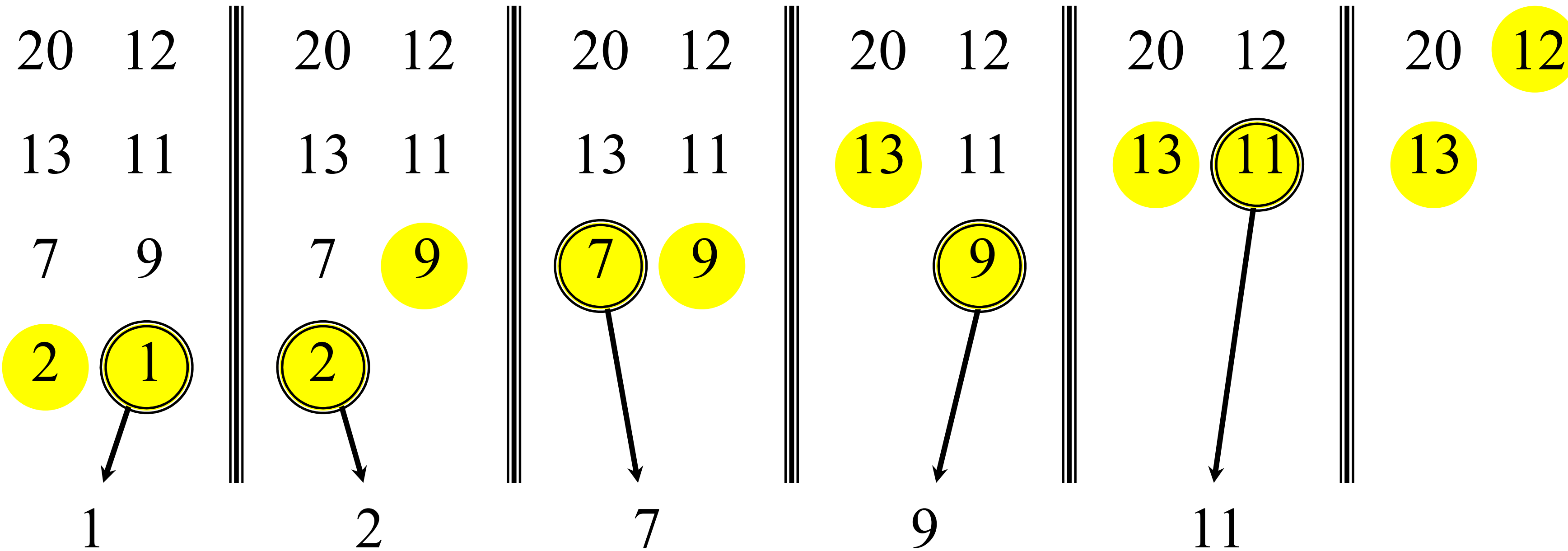



# Merging two sorted arrays

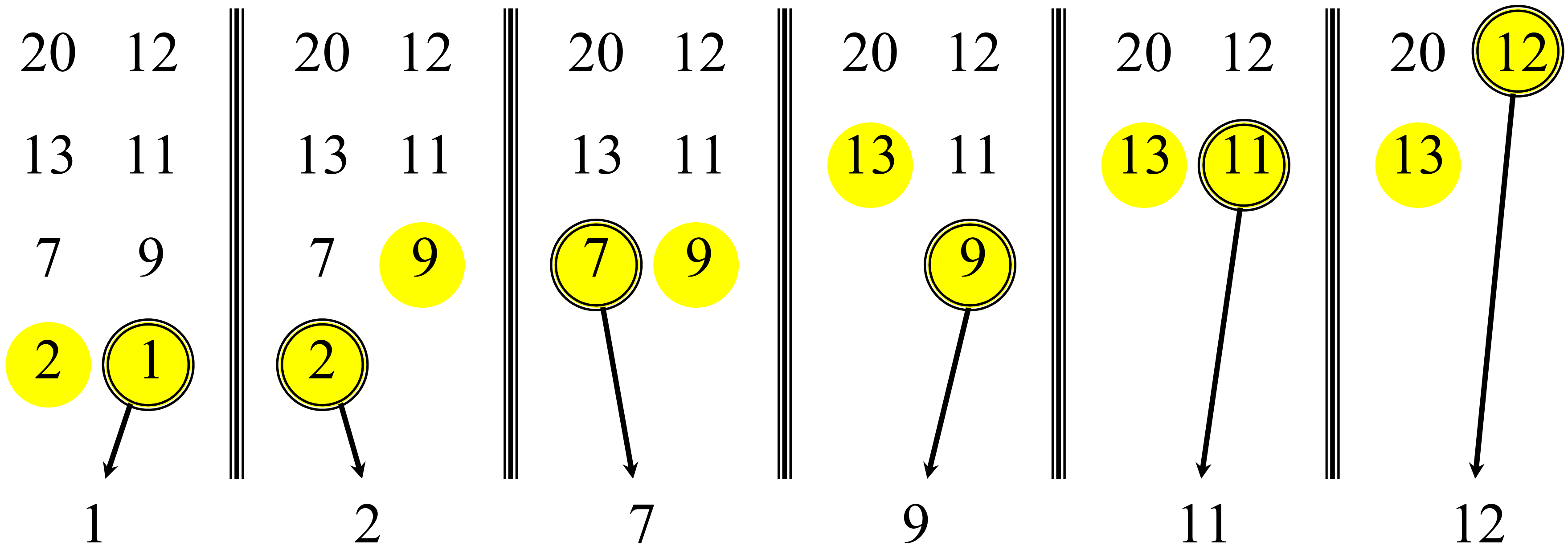


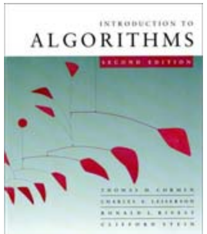



# Merging two sorted arrays

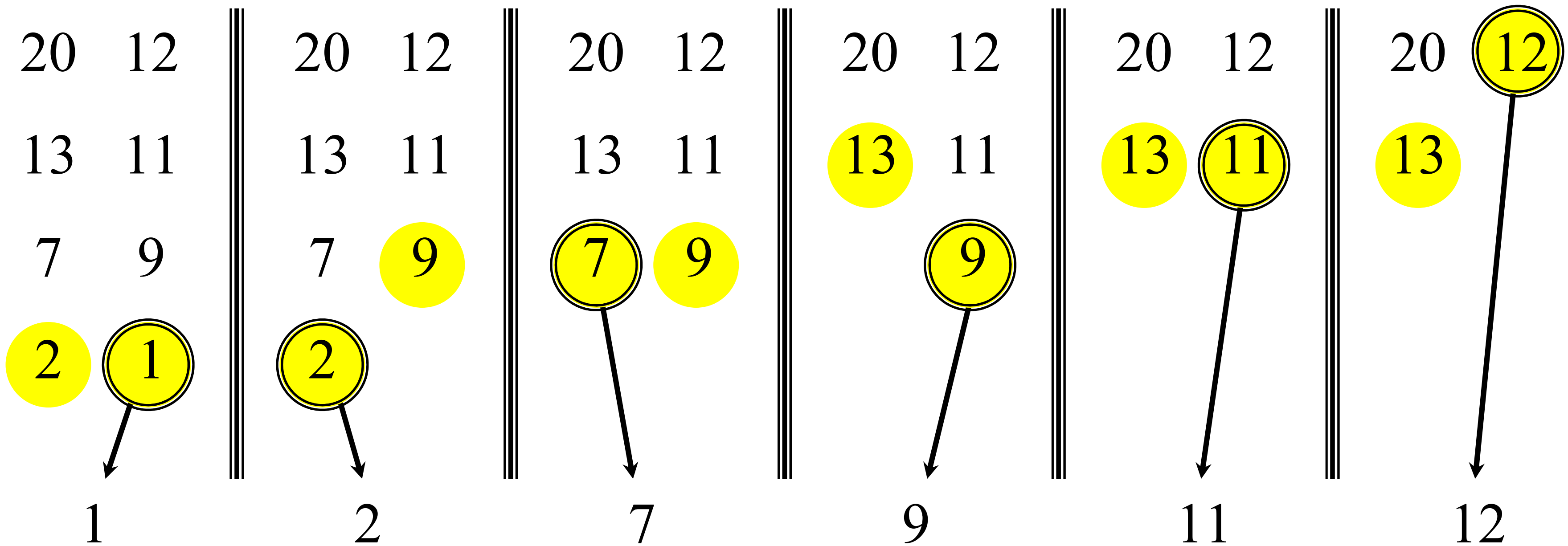


# Merging two sorted arrays

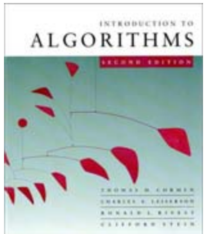





# Merging two sorted arrays




Time = Θ($n$) to merge a total of $n$ elements (linear time).

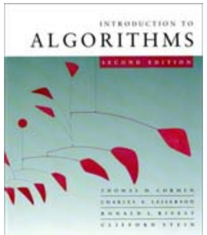



# Analyzing merge sort

| | |
|---|---|
| $T(n)$ | MERGE-SORT $A[1\,.\,.\,n]$ |
| $\Theta(1)$ | 1. If $n = 1$, done. |
| $2T(n/2)$ | 2. Recursively sort $A[\,1\,.\,.\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,.\,.\,n\,]$. |
| $\Theta(n)$ | 3. ***"Merge"*** the $2$ sorted lists |

***Abuse***

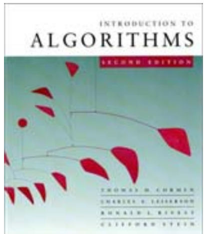
***Sloppiness:*** Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$ , but it turns out not to matter asymptotically.




# Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small $n$, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n) = \Theta(\mathrm{nlgn})$.




# Conclusions

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.
- Go test it out for yourself!