Algorithms 演算法

Foundations

— Introduction —

Professor James Chien-Mo Li 李建模
Electrical Engineering Department
National Taiwan University

Outline

- Introduction, CH1
- Getting Started, CH2
  - Insertion Sort
  - Merge Sort
- Growth of Functions, CH3
- Divide and Conquer, CH4

Introduction

- What is an *Algorithm*?
  - well-defined procedure to transform some *input* to desired *output*
- What is a *Problem*?
  - A statement specify the desired input/output relationship
- What is a *good* algorithm?
  - An algorithm is *correct*
    - For every input instance, it halts with a correct output
  - An algorithm is *efficient*
    - Runs very fast (low *time complexity*)
    - Needs little storage space (low *space complexity*)

Good Algorithm: Correct & Efficient

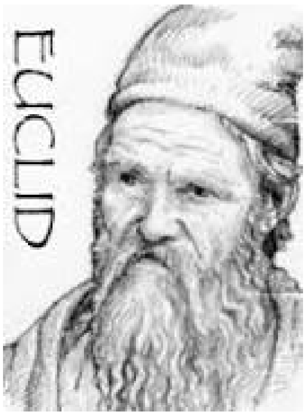
Al-Khwārizmī (780-850, Persian)

- Al-Khwārizmī, Persian astronomer and mathematician, wrote a treatise in 825 AD, On "*Calculation with Arabic Numerals*".
- It was translated into Latin in the 12th century as "*Algoritmi de numero Indorum*", whose title was likely intended to mean "*Algoritmi on the numbers of the Indians*",
  - where "Algoritmi" was the author's name
- But people misunderstanding the title treated Algoritmi as a Latin plural and this led to the word "algorithm" (Latin algorismus) coming to mean "calculation method"

Food For Thoughts:
Why Arabs are Good at Math?

History of Algorithms

- Euclid invented the first algorithm to find GCD (300 BC)
- Formalized by *Church-Turing Thesis* in 1936
- New Algorithms still being found recently, even by student like you



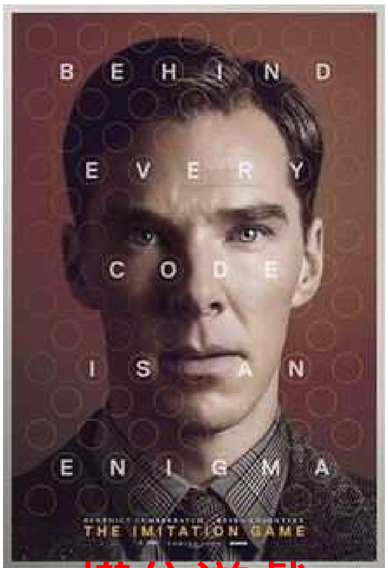
Euclid
(300BC)

Alonzo Church
(1903 – 1995)

Alan Turing
(1912 – 1954)

Why Study Algorithms?

- Top 3 reasons to study Algorithm:
-
-
-



模仿遊戲

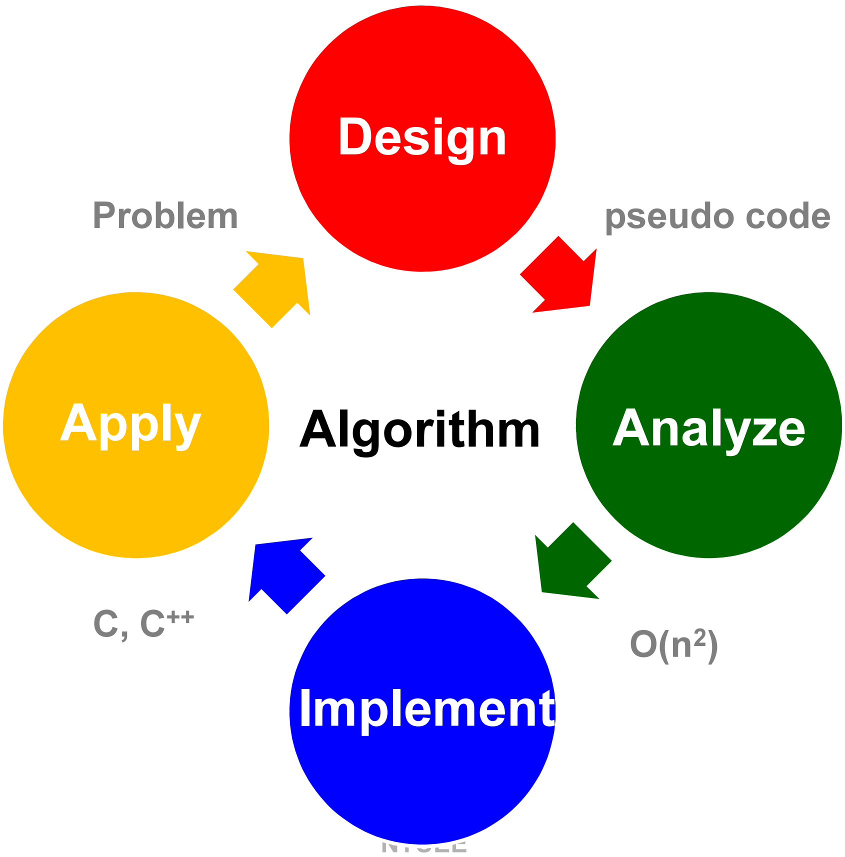
Complexity Comparison

- **Smart algorithm could make *huge* difference**
  - **n = input size; lg = $\log_2$**

Order	Name	n = 10	n = 100	$n = 10^3$	$n = 10^6$
1	constant	1×10^{-9} sec	1×10^{-9} sec	1×10^{-9} sec	1×10^{-9} sec
$\lg n$	logarithmic	3×10^{-9} sec	7×10^{-9} sec	1×10^{-8} sec	2×10^{-8} sec
$\sqrt{n}$	square root	3×10^{-9} sec	1×10^{-8} sec	3×10^{-8} sec	1×10^{-6} sec
n	Linear	1×10^{-8} sec	1×10^{-7} sec	1×10^{-6} sec	0.001 sec
$n \lg n$	linearithmic	3×10^{-8} sec	2×10^{-7} sec	3×10^{-6} sec	0.006 sec
n^2	quadratic	1×10^{-7} sec	1×10^{-5} sec	0.001 sec	16.7 min
n^3	cubic	1×10^{-6} sec	0.001 sec	1 sec	3×10^{5} cent.
2^n	exponential	1×10^{-6} sec	3×10^{17} cent.	∞	∞
$n!$	factorial	0.003 sec	∞	∞	∞

1 million instruction per second (MIPS)

What Do We Learn?



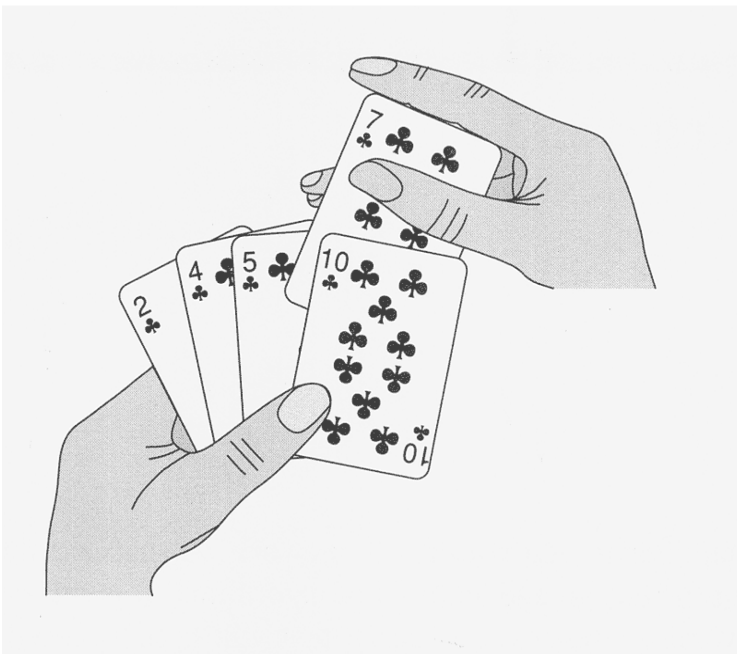
Sorting Problem

- Input: sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- Output: permutation (reordered) $\langle a_1', a_2', \ldots, a_n' \rangle$ such that
  - $a_1' \le a_2' \le \ldots \le a_n'$
- Example:
  - Input: <8, 6, 9, 7, 5, 2, 3>
  - Output: <2, 3, 5, 6, 7, 8, 9 >

- Any good algorithm?
  - *incremental* approach: *insertion sort*
  - *divide and conquer* approach: *merge sort*

How Do You Sort Cards?

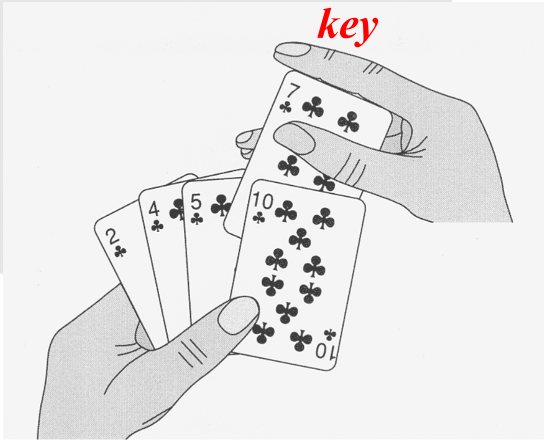
- Simple idea:
  - keep left cards sorted, right cards unsorted
  - each time insert a new card to left cards, in sorted order
  - repeat until all cards inserted



Insertion Sort

- *A.length* = number of elements in array A

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].      comment
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1                                   while loop
    A[i + 1] = key   // insert here
for loop
```

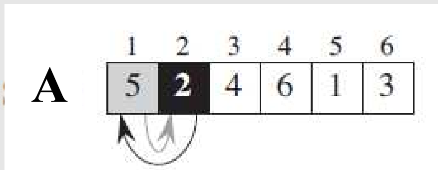


* see last page for more details about pseudo code convention

Operation of Insertion Sort (1)

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

j=2, *key*=2



i= 1

A[1+1]=*A*[1]

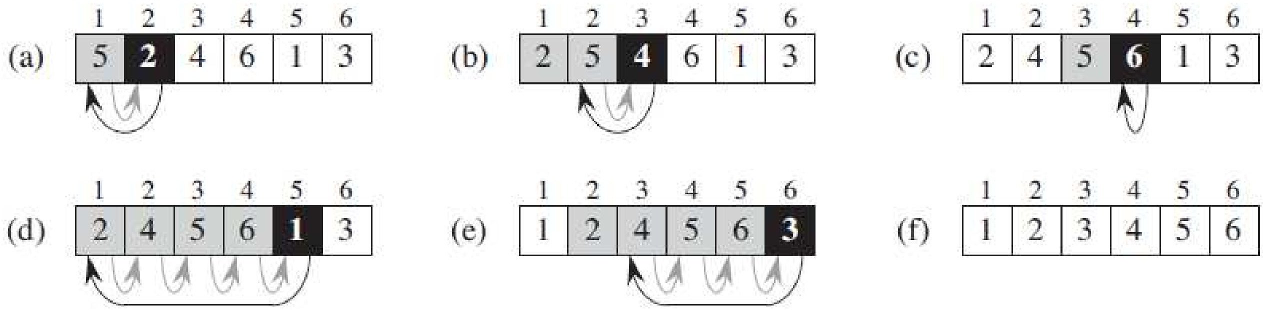
i= 0

A[0+1]=2

Operation of Insertion Sort (2)

- Fig 2.2

j=2, key=2 *j=3, key=4* *j=4, key=6*



j=5 *j=6*

Q1: Insertion Sort Correct ?

- Use ***Loop Invariant*** to prove following property always true,
  - Example:

At start of each iteration of for loop, subarray $A[1 \ldots j-1]$ consists of elements originally in $A[1 \ldots j-1]$ but in sorted order.

- To use loop invariant, we must show three things:
  - ***Initialization***:
    - Property is true before first iteration
  - ***Maintenance***:
    - Property remains true before every iteration
  - ***Termination***:
    - When loop terminates, invariant gives us a useful property to show that algorithm is correct

LI is like Math Induction

Prove Insertion Sort Correct

```
for j = 2 to A.length
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

- ***Loop Invariant property:***

> At start of each iteration of for loop, subarray $A[1 \dots j-1]$ consists of elements originally in $A[1 \dots j-1]$ but in sorted order.

- Initialization: $j = 2$
  - $A[1 \dots j-1]$ has only one element $A[1]$, which is trivially sorted
- Maintenance: $2<j<n+1$
  - moving $A[j-1], A[j-2], A[j-3], \dots$ by one position to the right until proper position for *key* is found
  - $A[1 \dots j]$ consists of original elements in sorted order
- Termination: $j=n+1$
  - $A[1 \dots n]$ consists of elements originally in $A[1 \,..\, n]$ in sorted order.
  - So entire array is sorted! QED

$A[1]$ $\quad$ $A[j\text{-}1]$	$A[j]$ $\quad$ $A[n]$

Q2: Insertion Sort Efficient?

- Given input size n, find running time of insertion sort
  - ***Running time*** = number of ***primitive operations*** executed
    - Primitive operations: arithmetic, compare …
  - ***Input size,*** n = number of items in input
    - *e.g.* n = size of array being sorted

- We will show 3 ***time complexity analysis*** for IS
  - ***Exact analysis***: very tedious
  - ***Worst-case/best-case/average-case analysis***: slow
  - ***Asymptotic analysis***: good for large *n*

> **Time Complexity Measures**
> **Algorithm Efficiency, esp. for Large *n***

Exact Analysis

- Let *T(n)* = running time of insertion sort given input size *n=A.length*

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

- t_j = number of times the "while" loop execution for value *j*

INSERTION-SORT(A)	*cost*	*times*
for $j=2$ **to** $A.length$	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	$n-1$

t_j

Why different?

BC/WC/AC Analysis

- Best-case: if array already sorted
  - $t_j=1$; *T(n)* is a linear function of *n*, or called *linear time*

$$T(n)=c_1n+c_2(n-1)+c_4(n-1)+c_5(n-1)+c_8(n-1)$$
$$=(c_1+c_2+c_4+c_5+c_8)n-(c_2+c_4+c_5+c_8)$$

- Worst-case: if array is in reverse sorted order
  - $t_j=j$; *T(n)* is a *quadratic function* of *n*

$$T(n)=c_1n+c_2(n-1)+c_4(n-1)+c_5\left(\frac{n(n+1)}{2}-1\right)$$
$$+c_6\left(\frac{n(n-1)}{2}\right)+c_7\left(\frac{n(n-1)}{2}\right)+c_8(n-1)$$
$$=\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n-(c_2+c_4+c_5+c_8)$$

$$\sum_{j=2}^{n}j=\frac{n(n+1)}{2}-1$$

- Average-case: random order
  - half elements are less than *A*[*j*]
  - $t_j\approx j/2$, *T*(*n*) is still a *quadratic function* of *n*

Asymptotic Analysis

- *Asymptotic analysis* looks at growth of $T(n)$ as $n \to \infty$
  - Easier than exact, BC/WC/AC analysis
- Θ *notation:* drop low-order terms and ignore coefficients
  - *e.g.* $5n^2+3n+4= \Theta(n^2)$
- Worst case: input reverse sorted, while loop is $t_j = \Theta(j)$

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

- Average case: all permutations equally likely, while loop is $t_j = \Theta(j / 2)$

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

- Both WC and AC are *asymptotically* $\Theta(n^2)$

$\boxed{\text{Insertion Sort is } \Theta(n^2)}$

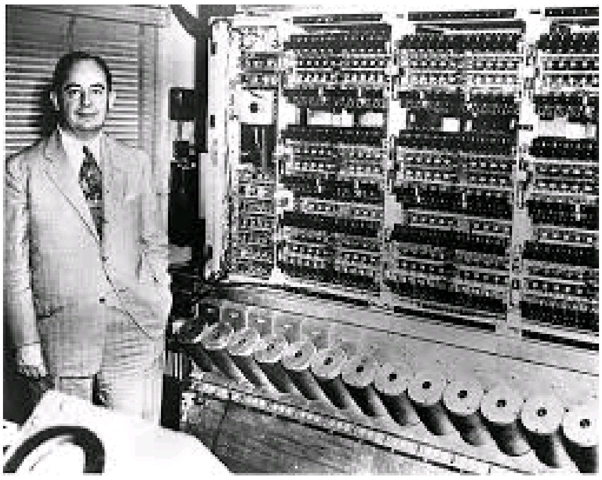
Outline

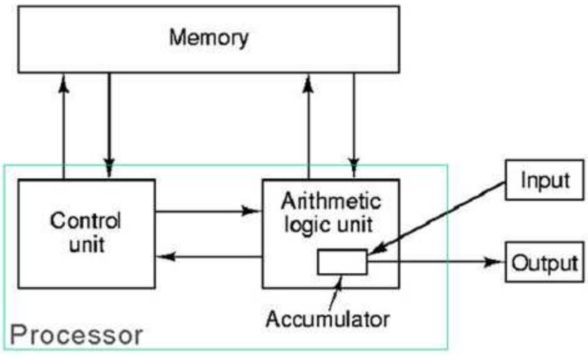
- Introduction, CH1
- Getting Started, CH2
  - Insertion Sort
  - Merge Sort
- Growth of Functions, CH3
- Divide and Conquer, CH4

John von Neumann (1903-1957, USA)

- Hungarian and American mathematician. He worked on one of earliest electronic computers (**EDVAC**), where he invented
  - **Merge Sort** (world's first non-trivial algorithm on computer)
  - **Von Neumann architecture** (still used by today's computers)

"If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is." *J. von Neumann*





First Draft of a Report on EDVAC, 1945

Divide and Conquer Approach

- Insertion sort uses ***incremental*** ***approach***
  - first sort subarray $A[1...j-1]$ then insert a single $A[j]$
  - too slow for large problems
  - how can we do better?

- ***Divide and conquer*** approach
  - ***Divide*** problem into a number of smaller subproblems
    - ***recursive case***: when subproblems are large, solve recursively
  - ***Conquer*** subproblems by solving them recursively
    - ***base case***: when subproblems are small, solve by brute force
  - ***Merge*** subproblem solutions to total solution

$$\boxed{\text{If } T(n)=n^2,\ n\rightarrow\tfrac{1}{2},\ T(n)\rightarrow\tfrac{1}{4}}$$

Merge Sort: Idea

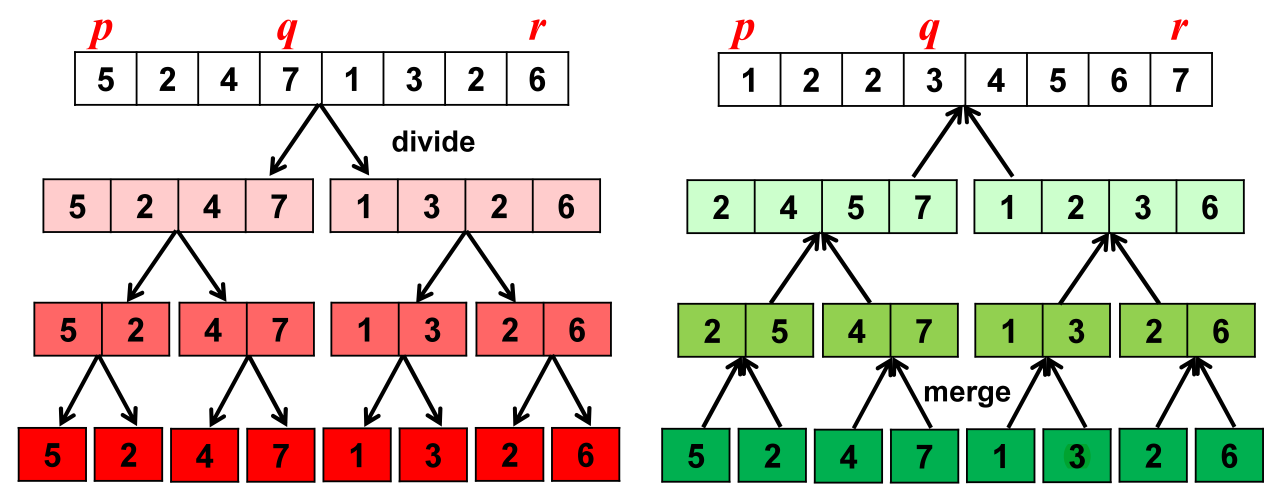
- Divide array into left and right halves
- Sort each halves
- Merge together

input	5	2	4	7	1	3	2	6
sort left	2	4	5	7	1	3	2	6
sort right	2	4	5	7	1	2	3	6
merge	1	2	2	3	4	5	6	7

But Each Halves Still Too Large…

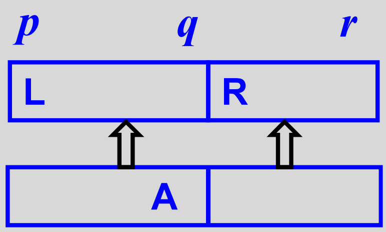
Merge Sort: Top-down Recursion

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```



MERGE (1)

```
MERGE(A, p, q, r)
   n1 = q − p + 1
   n2 = r − q
   let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
   for i = 1 to n1
       L[i] = A[p + i − 1]
   for j = 1 to n2
       R[j] = A[q + j]
   L[n1 + 1] = ∞
   R[n2 + 1] = ∞
   i = 1
   j = 1
   for k = p to r
       if L[i] ≤ R[j]
           A[k] = L[i]
           i = i + 1
       else
           A[k] = R[j]
           j = j + 1
```



MERGE (2)

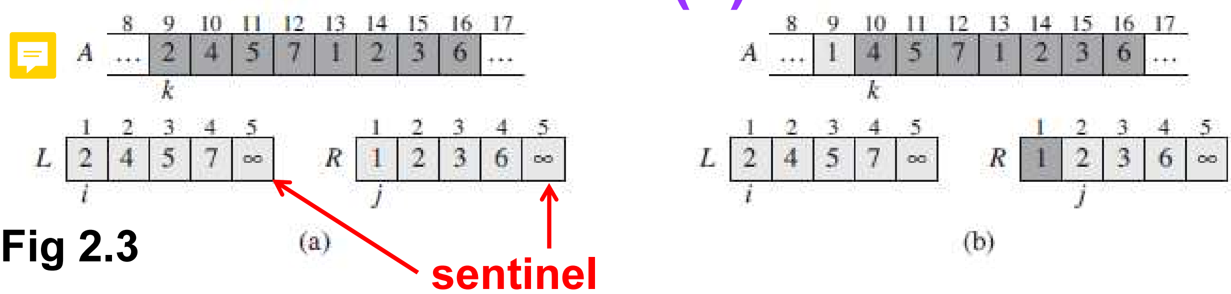
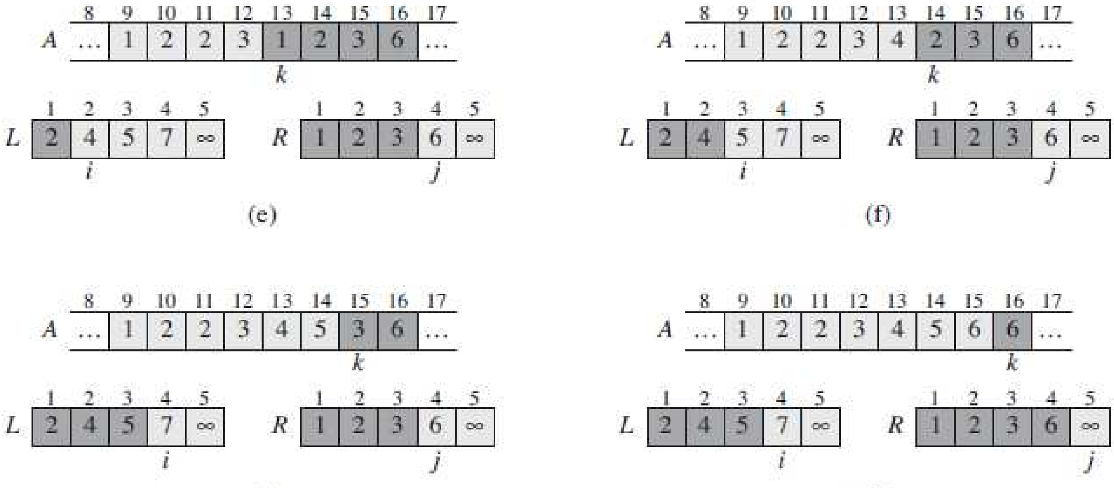


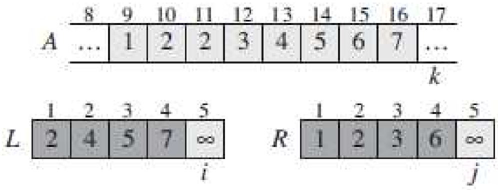
Fig 2.3

```
   L[n1 + 1] = ∞     // add sentinel at end
   R[n2 + 1] = ∞
   i = 1
   j = 1
   for k = p to r
       if L[i] ≤ R[j]       // left is smaller
           A[k] = L[i]      // copy to A
           i = i + 1
       else
           A[k] = R[j]      // right is smaller
           j = j + 1        // copy to A
```

MERGE (3)



(e) (f) (g) (h)



Q1: What is sentinel for?

Q2: What is time complexity?

$\Theta(1)$? $\Theta(n)$? $\Theta(n^2)$?

Time Complexity of MERGE

- *n* cards into two piles
  - Each pile is sorted and placed face-up
  - We will merge them into a single sorted pile
- Repeat following basic steps: (at most *n* iterations)
  - Choose smaller of the two top cards, remove it from its pile
  - Place the chosen card face-down onto output pile
  - Repeat until one input pile is empty
- Each basic step should take constant time
- Each card is removed only once

Merge is Linear Time

Time Complexity of Merge-Sort

- Merge-Sort is a ***recursive function***
  - describe function in terms of itself

MERGE-SORT(A, p, r)	$T(n)$
if $p < r$	$\Theta(1)$
then $q \leftarrow \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
MERGE-SORT(A, p, q)	$T(n/2)$
MERGE-SORT$(A, q+1, r)$	$T(n/2)$
MERGE(A, p, q, r)	$\Theta(n)$

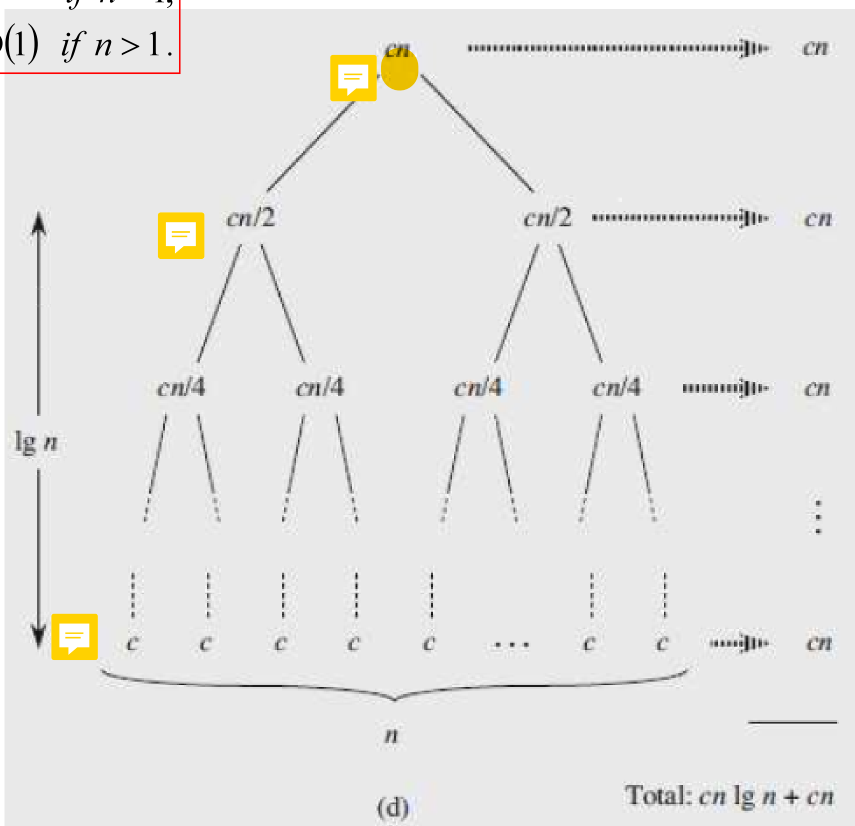
- *T(n)* can be calculated ***recursively***

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

Recursion Tree

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

- c = constant
- Each level is cn
- Height is $\lg n$
- $(\lg n) + 1$ levels of cn
- $T(n) = cn[(\lg n) + 1]$
  $= \Theta(n \lg n)$
  - *lg* means $\log_2$
- More details
  - See 1.4



Food for Thoughts (FFT)

- $\Theta(n \lg n)$ is smaller than $\Theta(n^2)$
  - merge sort is faster than insertion sort
- Q: merge sort is weaker than insertion sort in one thing. Can you point it out?
  - hint: why don’t we use merge sort when we sort card
  - no free lunch ☺