Ch. 2: Getting Started

About this lecture

- Study a few simple algorithms for sorting
  - Insertion Sort
  - Selection Sort, Bubble Sort (Exercises)
  - Merge Sort
- Show why these algorithms are correct
- Try to analyze the efficiency of these algorithms (how fast they run)

The Sorting Problem

Input: A list of *n* numbers

Output: Arrange the numbers in increasing order

Remark: Sorting has many applications.

If the list is already sorted, we can search a number in the list faster.

Insertion Sort

- A good algorithm for sorting a small number of elements
- It works the way you might sort a hand of playing cards:
  - 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
  - Finally, the cards held in the left hand are sorted

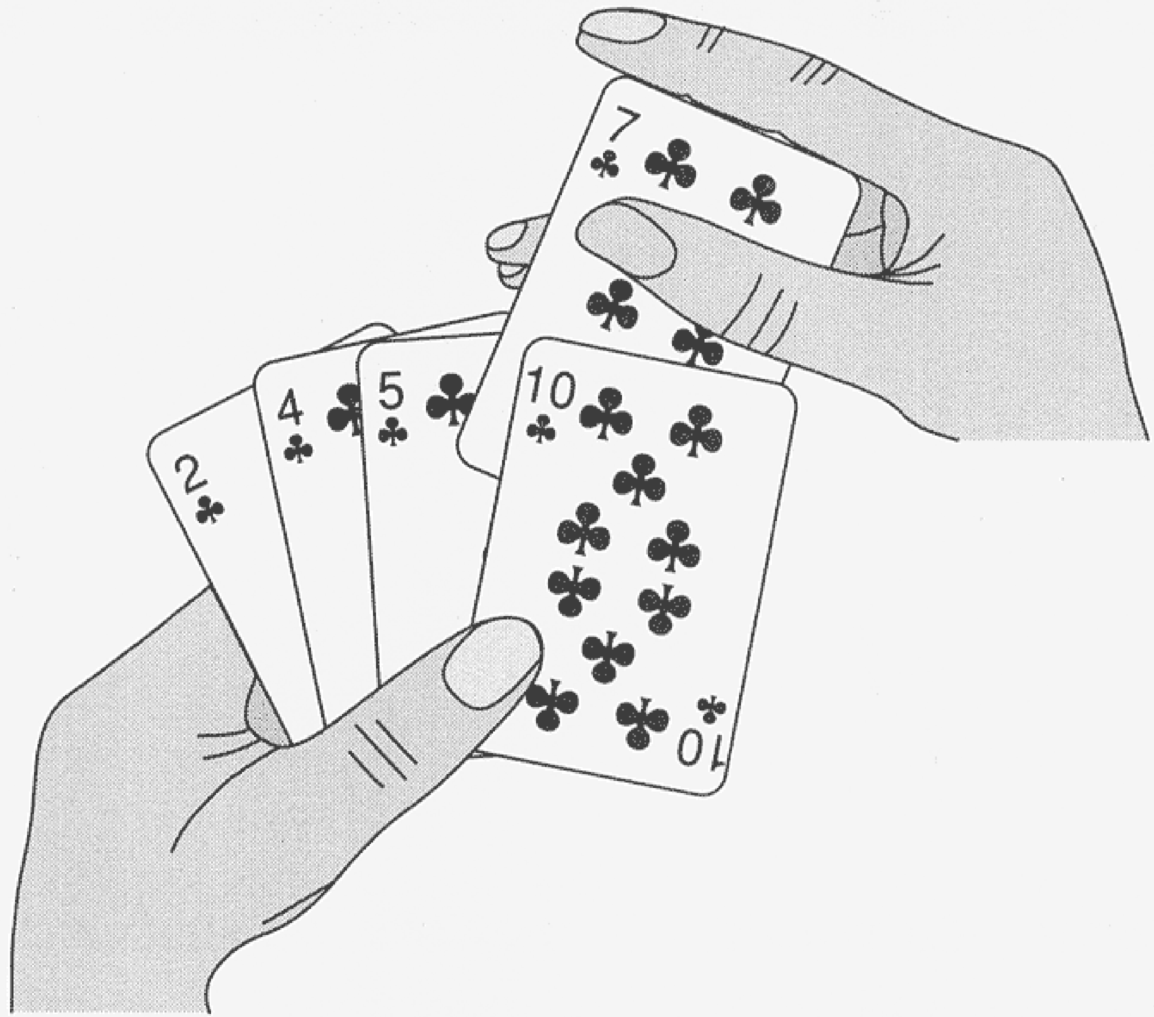
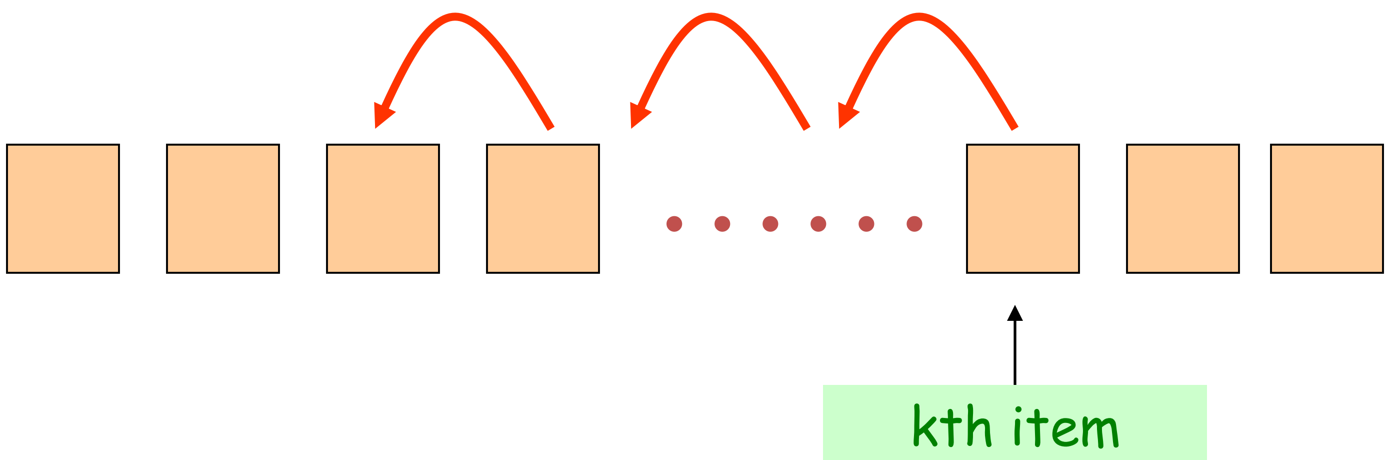


Figure 2.1 Sorting a hand of cards using insertion sort.

Insertion Sort

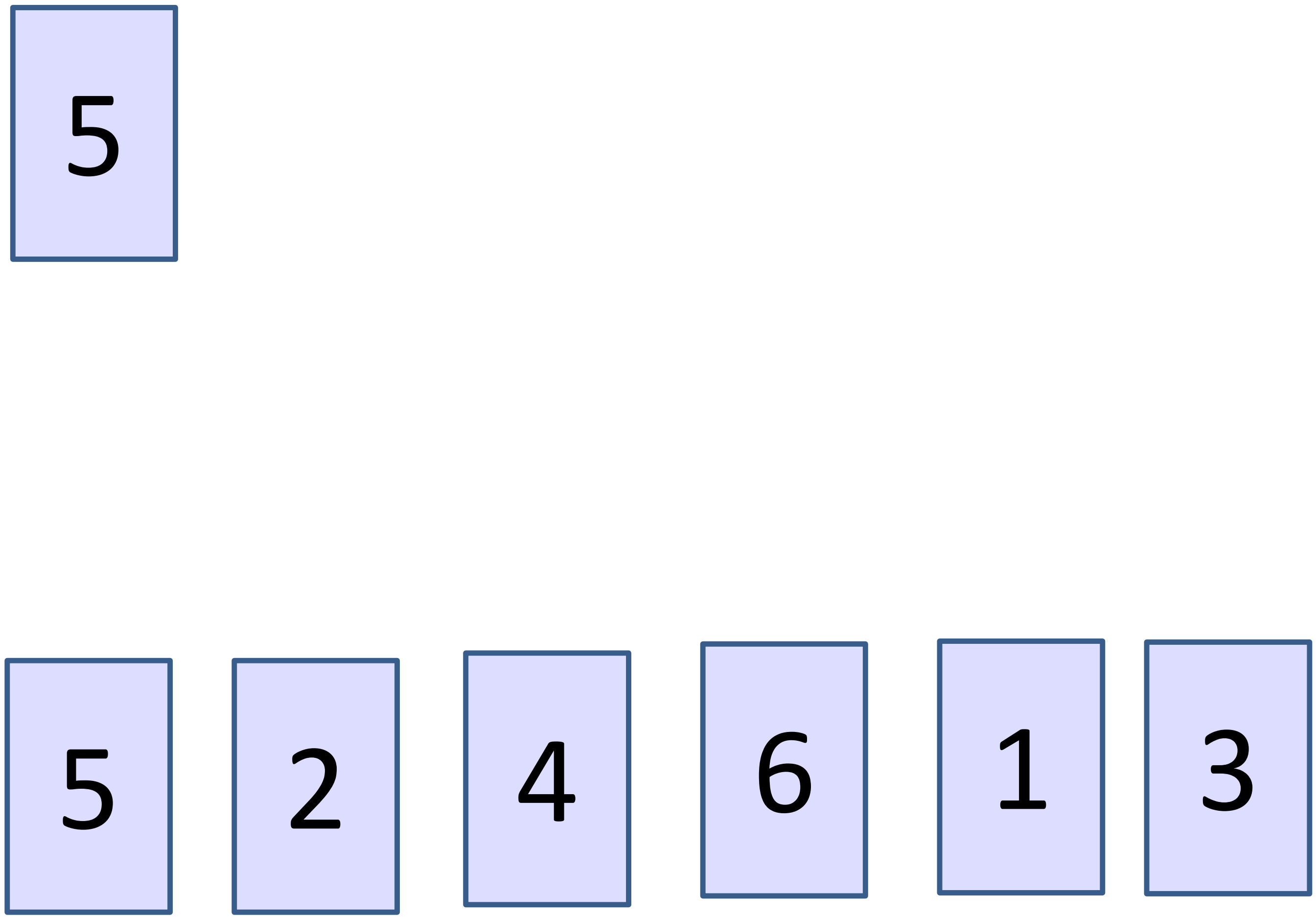
- Operates in *n* rounds
- At the k^{th} round,

Swap towards left side ;
Stop until seeing an item
with a smaller value.

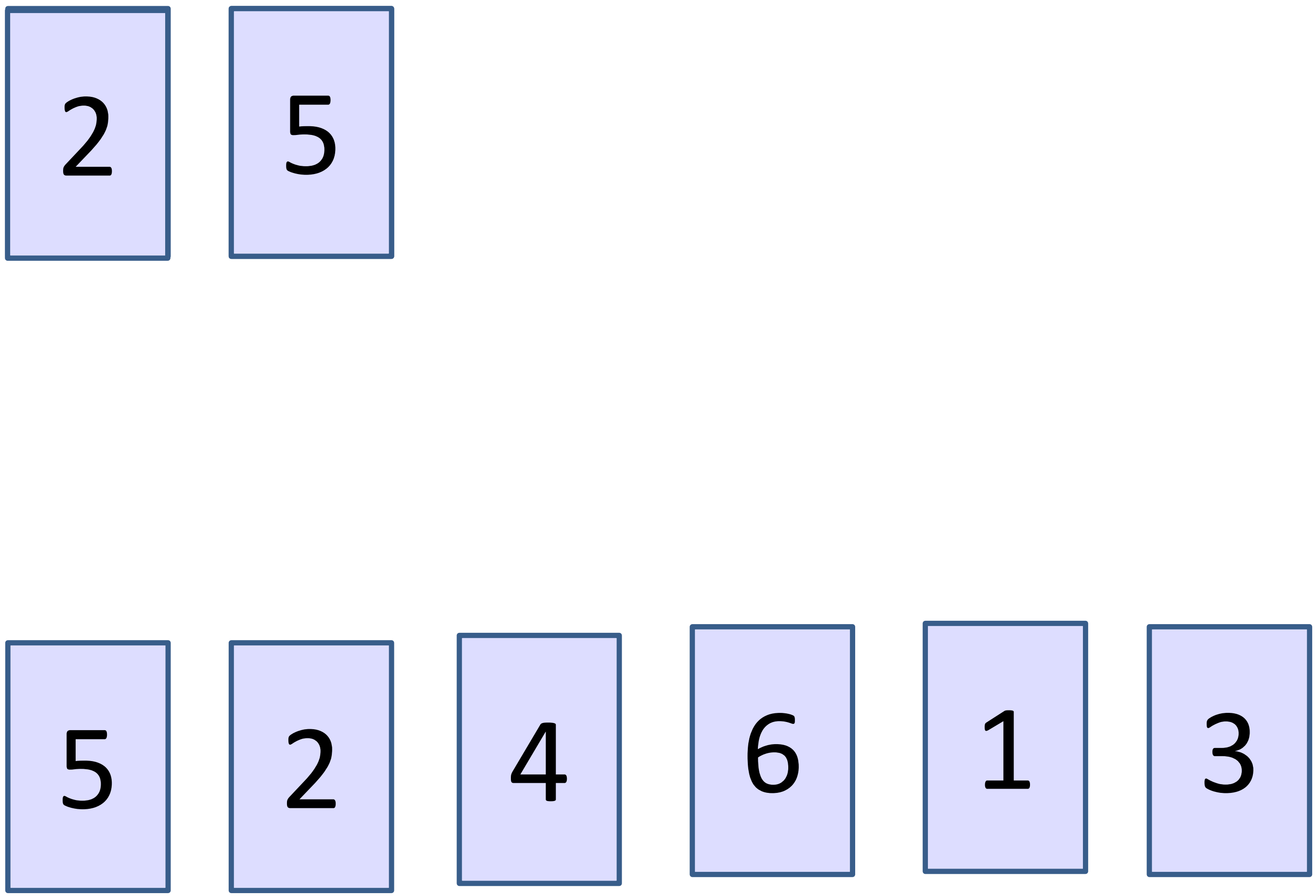


Question: Why is this algorithm correct?

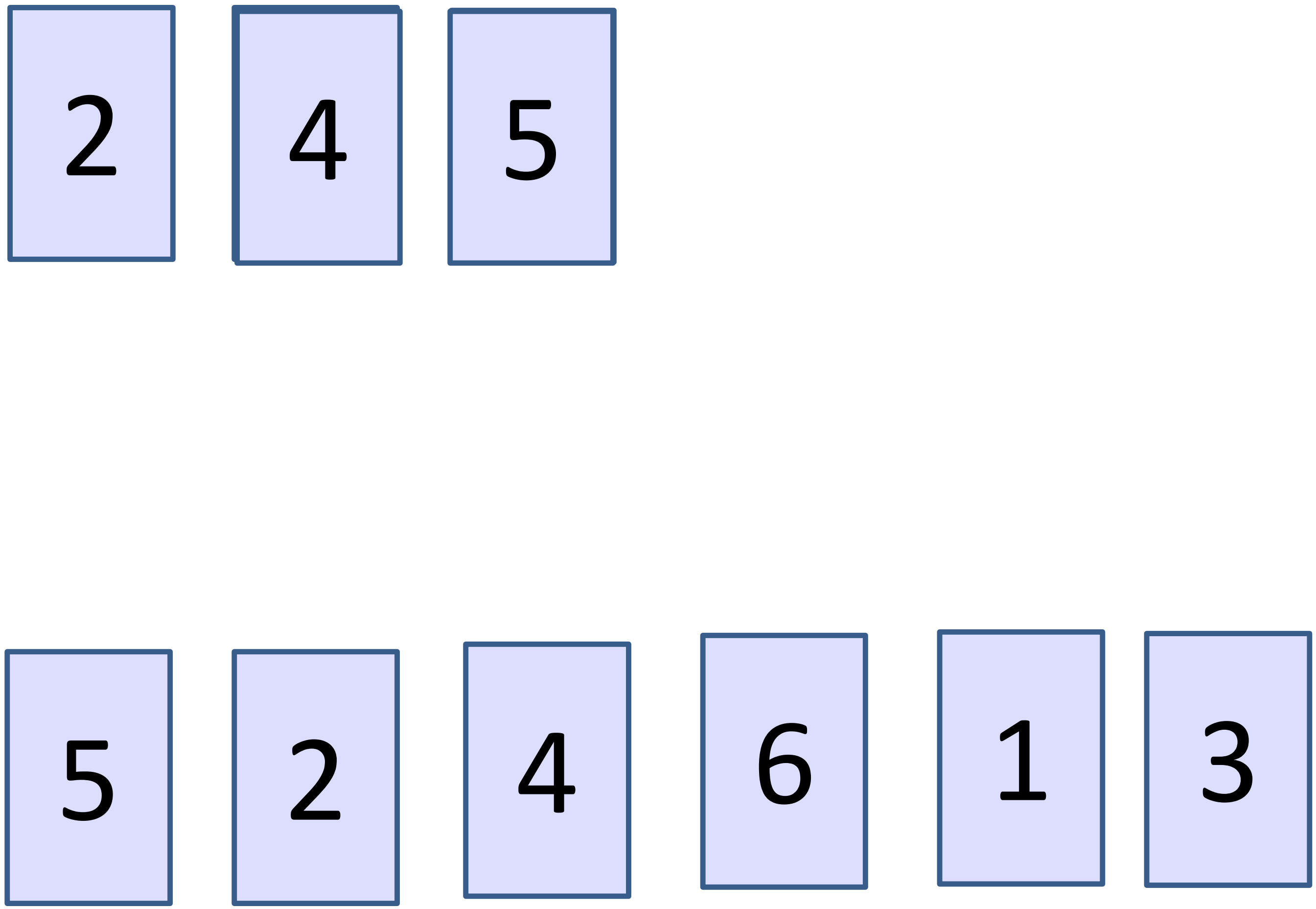
Insertion Sort Example



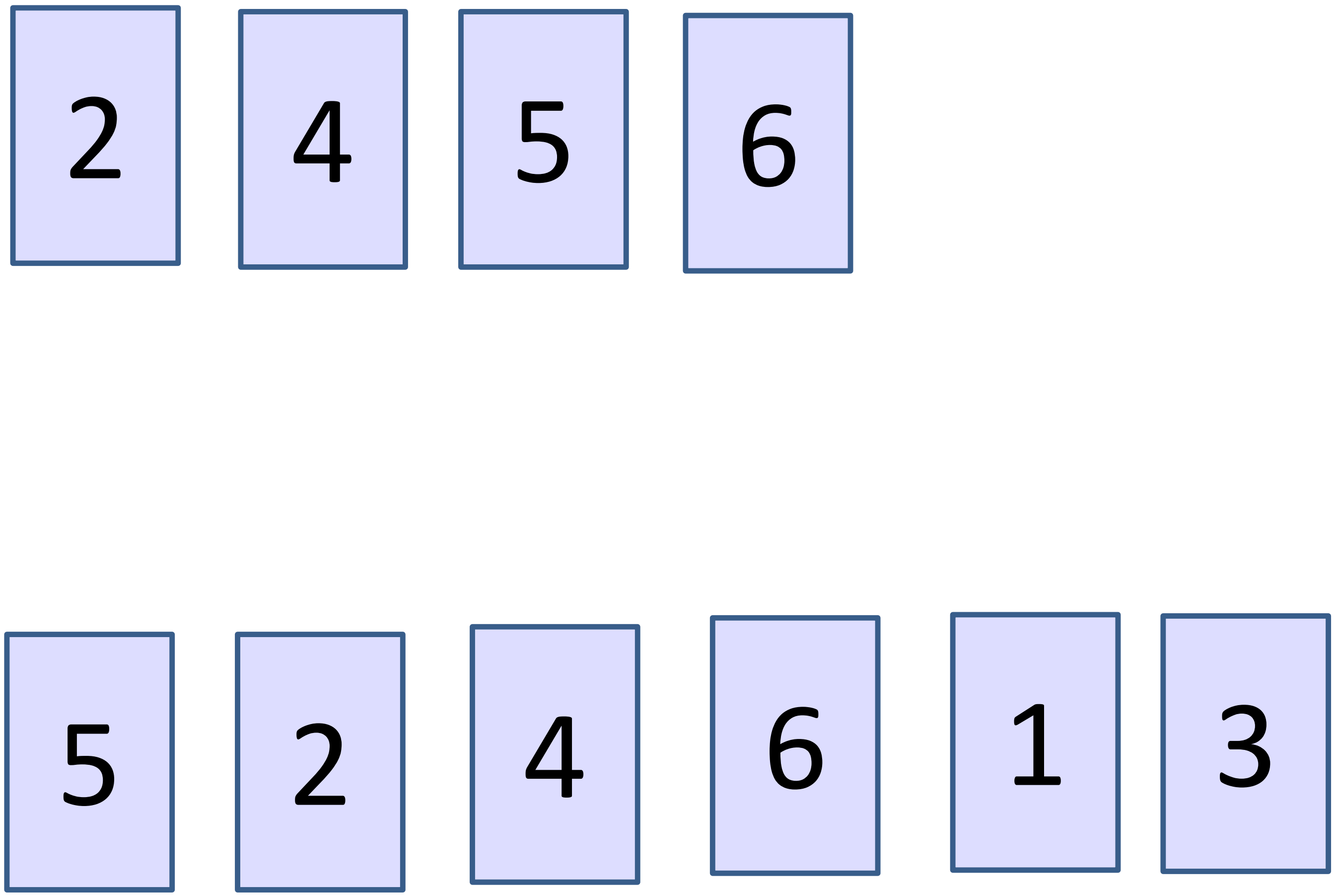
Insertion Sort Example



Insertion Sort Example



Insertion Sort Example

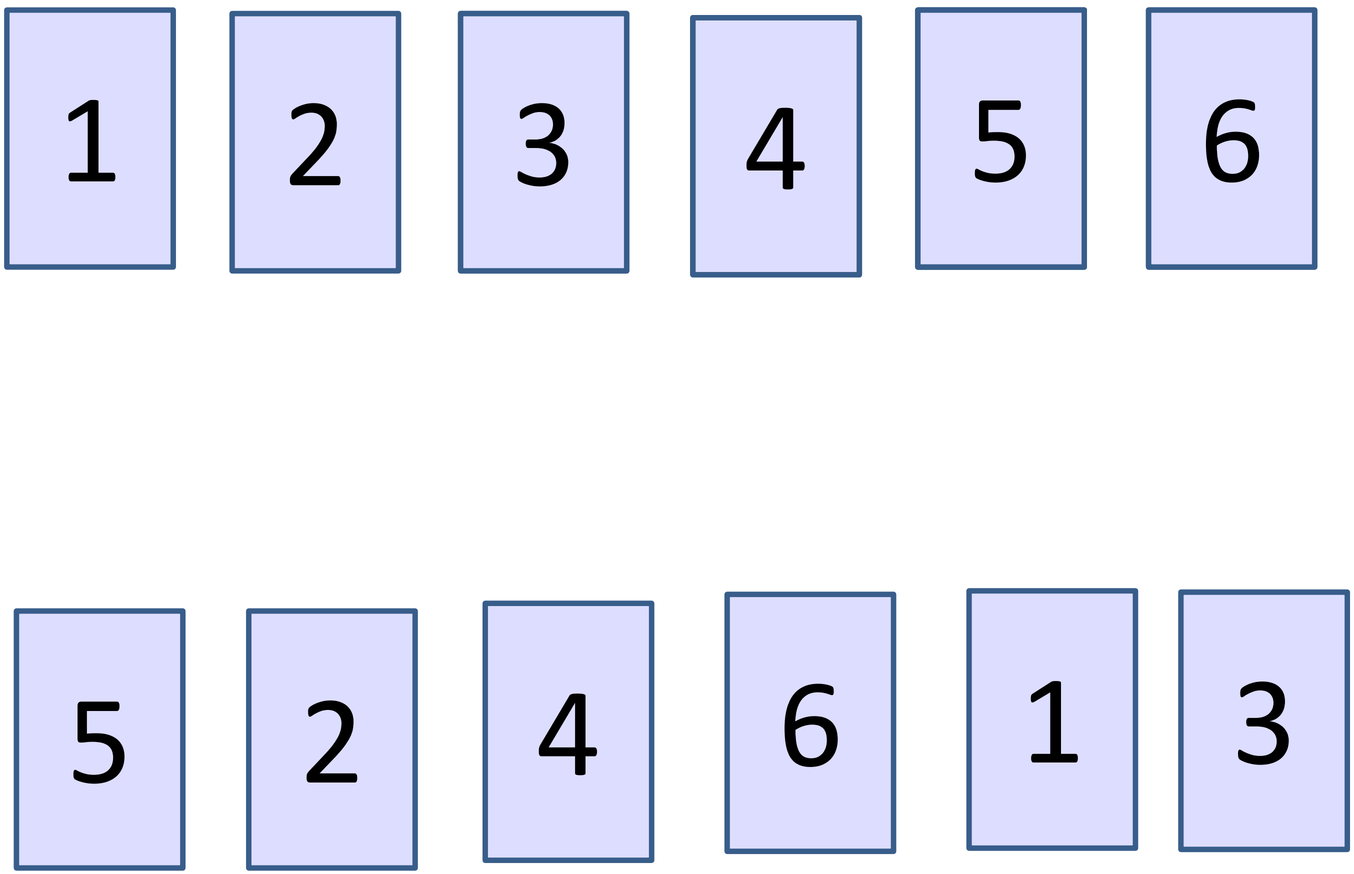


Insertion Sort Example

1	2	4	5	6

5	2	4	6	1	3

Insertion Sort Example



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

Loop invariants and the correctness of insertion sort

Correctness of Insertion Sort

- **Three properties for Loop Invariant:**
  - **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, array is sorted.
- **Loop invariant for Insertion sort:** At the start of each iteration of the for loop of lines 1-8, the subarray A[1..j-1] consists of the elements originally subarray A[1..j-1], but in sorted order

Analyzing the Running Times

- Which of candidate algorithms is the best?
- Compare their running time on a computer
  - But there are many kinds of computers !!!

Standard assumption: Our computer is a RAM (Random Access Machine), so that

- each arithmetic (such as $+$, $-$, $\times$, $\div$), memory access, and control (such as conditional jump, subroutine call, return) takes constant amount of time

Analyzing the Running Times

- Suppose that our algorithms are now described in terms of RAM operations

  ➔ we can count # of each operation used

  ➔ we can measure the running time!

- Running time is usually measured as a function of the input size
  - E.g., *n* in our sorting problem

Insertion Sort (Running Time)

The following is a pseudo-code for Insertion Sort.

Each line requires constant RAM operations.

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

t_j = # of times *key* is compared at round j

Insertion Sort (Running Time)

- Let $T(n)$ denote the running time of insertion sort, on an input of size n
- By combining terms, we have

$$T(n) = c_1 n + (c_2+c_4+c_8)(n-1) + c_5 \sum t_j +$$
$$(c_6+c_7) \sum (t_j - 1)$$

- The values of t_j are dependent on the input

Insertion Sort (Running Time)

- Best Case:

The input list is sorted, so that all $t_j = 1$

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

$= Kn + c$ ➔ linear function of n

- Worst Case:

The input list is sorted in decreasing order, so that all $t_j = j-1$

Then, $T(n) = K_1n^2 + K_2n + K_3$

➔ quadratic function of n

Worst-Case Running Time

- In our course (and in most CS research), we concentrate on worst-case time
- Some reasons for this:
  1. Gives an upper bound of running time
  2. Worst case occurs fairly often
- Remark: Some people also study average-case running time (they assume input is drawn randomly)

Divide and Conquer

→ Divide a big problem into smaller subproblems

→ Solve (Conquer) smaller subproblems recursively

→ Combine the results to solve original one

- The above idea is called Divide-and-Conquer
- Smart idea to solve complex problems
- Can we apply this idea for sorting ?

Divide-and-Conquer for Sorting

- What is a smaller subproblem ?
  - ➔ e.g., sorting fewer numbers
  - ➔ Let’s divide the list to two shorter lists
- Next, solve smaller subproblems (how?)
- Finally, combine the results
  - ➔ “merging” two sorted lists into a single sorted list (how?)

Merge Sort

- The previous algorithm, using divide-and-conquer approach, is called Merge Sort
- The key steps are summarized as follows:

Step 1. Divide list to two halves, A and B

Step 2. Sort A using Merge Sort

Step 3. Sort B using Merge Sort

Step 4. Merge sorted lists of A and B

Question: Why is this algorithm correct?

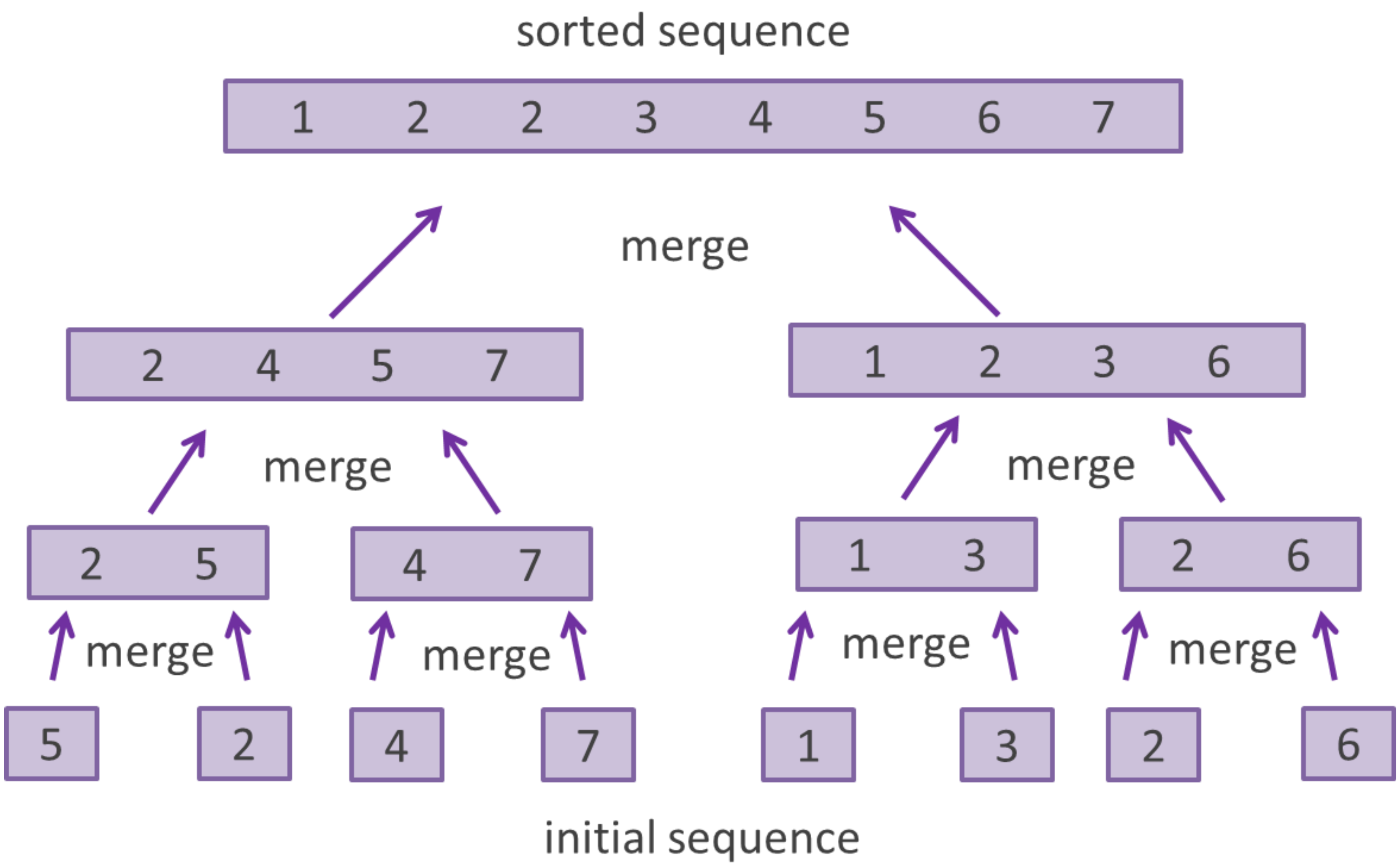


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Merge Sort (Running Time)

The following is a partial pseudo-code for Merge Sort.

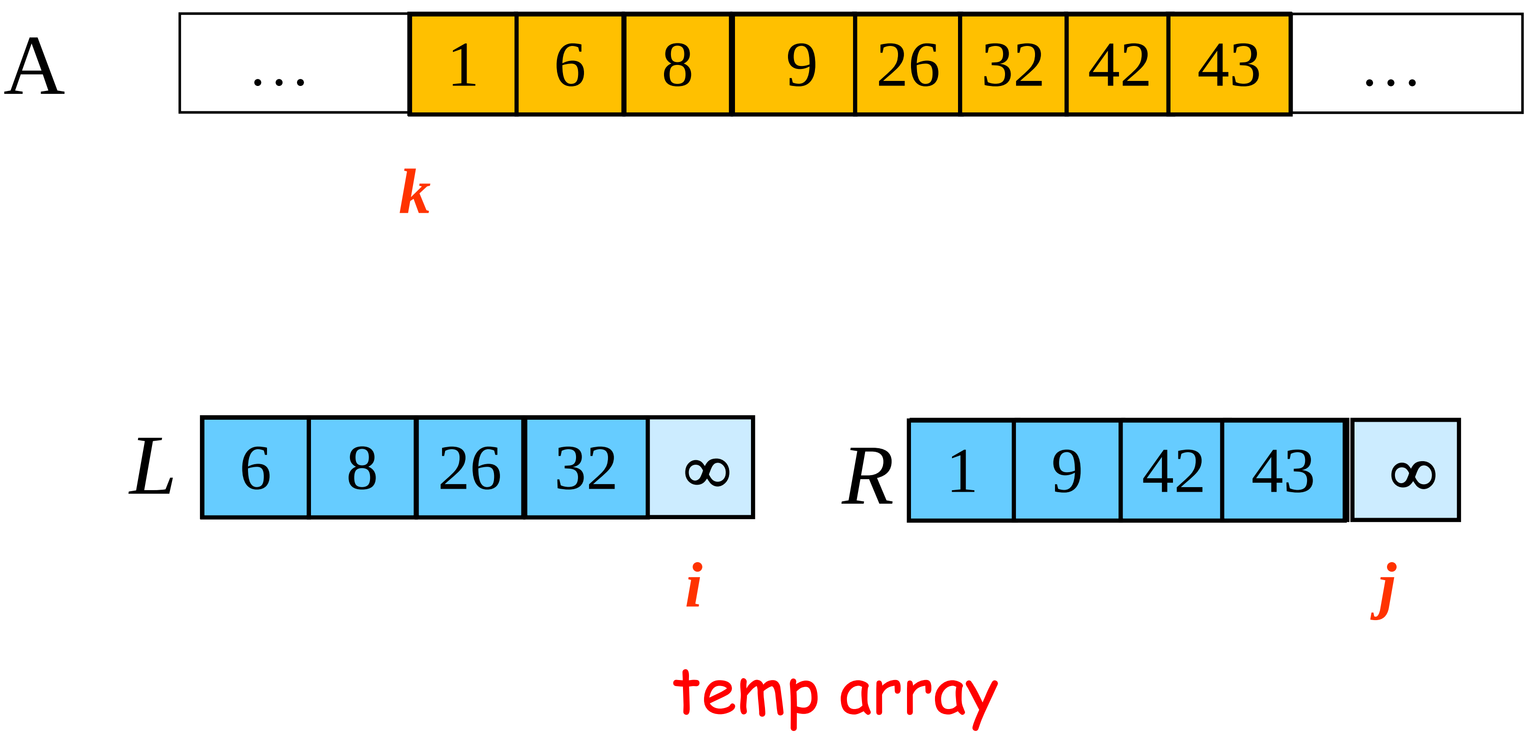
```
MERGE-SORT(A, p, r)
if p < r
    then q ← ⌊(p + r)/2⌋
         MERGE-SORT(A, p, q)
         MERGE-SORT(A, q + 1, r)
         MERGE(A, p, q, r)
```

The subroutine MERGE(A,p,q,r) is missing.

Can you complete it?

Hint: Create a temp array for merging

Merge – Example



Procedure Merge

```
Merge(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
    for i ← 1 to n1
        do L[i] ← A[p + i − 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1+1] ← ∞
    R[n2+1] ← ∞
    i ← 1
    j ← 1
    for k ←p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                    i ← i + 1
            else A[k] ← R[j]
                    j ← j + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Merge Sort (Running Time)

- Let $T(n)$ denote the running time of merge sort, on an input of size n
- Suppose we know that Merge( ) of two lists of total size n runs in $c_1 n$ time
- Then, we can write T(n) as:

  $T(n) = 2T(n/2) + c_1 n$ when $n > 1$

  $T(n) = c_2$ when $n = 1$
- Solving the recurrence, we have
- $T(n) = c_1 n \log n + c_2 n$

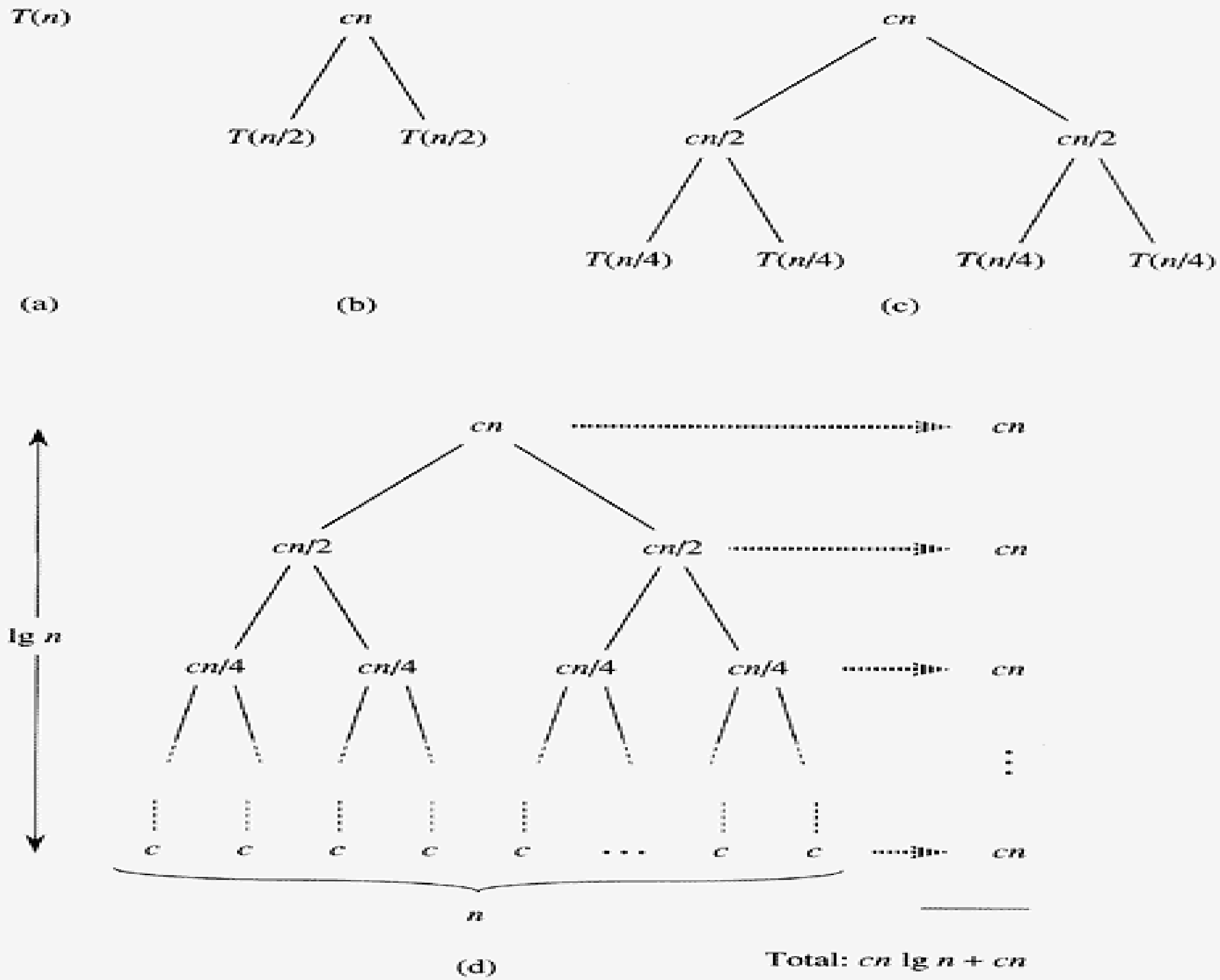


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Which Algorithm is Faster?

- Unfortunately, we still cannot tell
  - since constants in running times are unknown
- But we do know that if *n* is VERY large, worst-case time of Merge Sort must be smaller than that of Insertion Sort
- Merge Sort is **asymptotically** faster than Insertion Sort

天平與撞球

- 你有8顆撞球，其中一顆比較重，唯一的工具是一根天平，請問你最少要稱幾次，才能找出**較重**的那顆球?

天平與撞球

- 先比(1+2+3) 與 (4+5+6)球 的重量，如一樣重則有瑕疵的球為7, 8其中之一。

- 如不一樣重，則比較重的一邊任兩顆球，即可求得答案。

- 延伸題：(1)若有9顆撞球，請問你最少要稱幾次，才能找出**較重**的那顆球? (2)若有*N*顆撞球呢?

天平與撞球

- 你有8顆撞球，其中一顆重量跟其他7顆不一樣重，唯一的工具是一根天平，請問你最少要稱幾次，才能找出不一樣重的那顆球?

Practice at Home

- Problem: 2-1, 2-4
- Exercises: 2.1-4, 2.2-4, 2.3-4, 2.3-6, 2.3-7

Practice at Home

- Suppose we have *N* identical looking balls numbered 1 through *N* and only one of them is counterfeit ball whose weight is different with the others. Suppose further that you have one balance scale. Develop a method for finding the counterfeit coin with minimum number of weighing times in worst case.