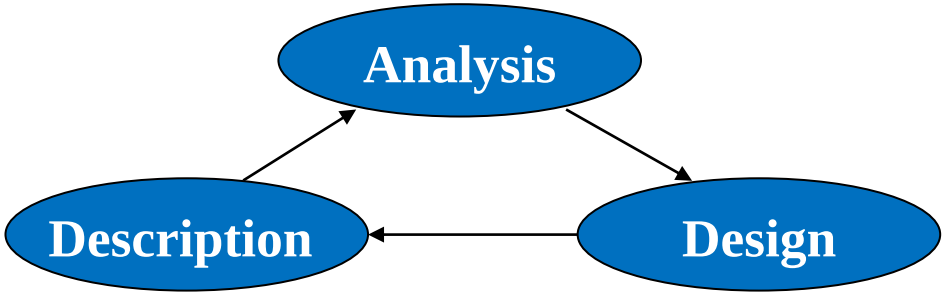
# Chapter 2

# Framework for analyzing algorithms

# 2 Framework for analyzing algorithms




- **Description**：See how to describe algorithms in pseudocode, C/C++ and argue the correctness of that.（如何描述算法）
- **Analysis**：Begin using asymptotic notation to express running-time analysis.（使用一致性符号来分析算法的运算时间）
- **Design**：for example, the technique of "divide and conquer" in the context of merge sort.（通过分而治之方法进行merge sort）
- Start using framework for describing and analyzing algorithms（描述和分析算法的框架）
- Examine two algorithms for sorting: insertion sort and merge sort

# 2 Framework for analyzing algorithms

**Design**

- **Description**
- **Correctness**

**Analysis**

- **Effeciency (running time)**

- **How**
- **Why**
- **What**

- **Who are you**
- **Where are you from**
- **Where will you go**

# 2 Framework for analyzing algorithms

**2.1 Insertion sort: Framework for describing algorithms**

**2.2 Insertion sort: Analyzing algorithms**

**2.3 Merge sort: Designing algorithms**

# 2.1 Insertion sort

Problem: to sort a sequence of numbers into nondecreasing order

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 $a_1' \leq a_2' \leq \ldots \leq a_n'$

# 2.1 Insertion sort --Description

- **Pseudocode**

  **That is similar in many respects to C, Pascal, or Java, py, …**

- **Differences between psuedocode and real code**
  - **Psuedocode is clear and concise to specify a given algorithm**（清晰、准确）
  - **Pseudocode is not typically concerned with issues of software engineering (data abstraction, modularity(模块化), error handling)**（不用考虑太多技术细节）

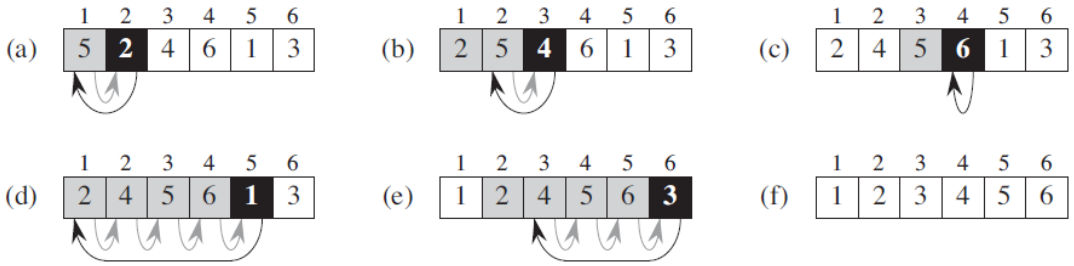
用伪代码可以体现算法的本质

永远不会过时（因为算法永远不会过时）

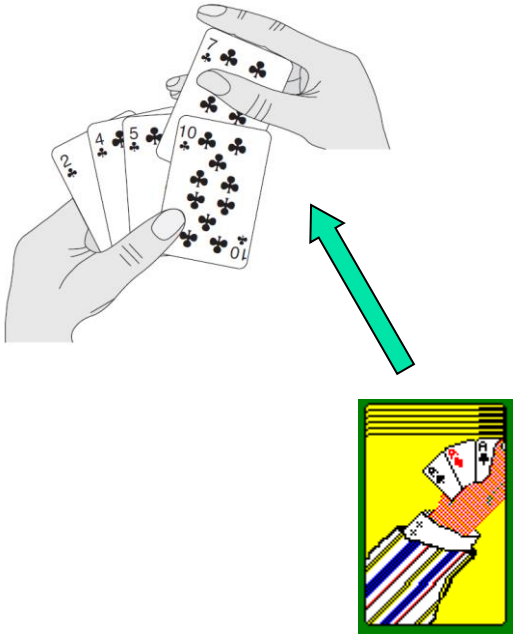
## 2.1.1 Pseudocode conventions

- Indentation indicates block structure. （缩排）
- The looping constructs (**while**, **for**, **repeat**) and the conditional constructs (**if**, **then**, **else**) have interpretations similar to those in Pascal, C.
- "// " or " ▷"indicates a comment.
- A multiple assignment $i \leftarrow j \leftarrow e$ is equivalent to $j \leftarrow e$ *then* $i \leftarrow j$.
- Variables are local to the given procedure.
- Array elements access: $A[i]$ ; $A[1 .. j] = < A[1], A[2], \ldots, A[j] >$
- Compound data are typically organized into objects.
- Parameters are passed to a procedure by value.
- The boolean operators "and" and "or" are short circuiting: for example " $x$ and (or) $y$ ": whether or not evaluate $y$ rely on the evaluating of $x$.

# 2.1 Insertion sort -- Description




```
INSERTION-SORT(A)
1     for( j = 2; j <=length[A]; j++) // loop header
2     {       key = A[j]
3             // Insert A[j] into the sorted sequence A[1 .. j-1]
4             i←j-1
5             while( i > 0 && A[i] > key)
6             {       A[i+1] = A[i]
7                     i = i-1
8             }
9             A[i+1] = key
10    }
```



**Is the algorithm correct ?**

## 2.1.2 Loop invariants and the **correctness** of insertion sort

- **Loop invariants（循环不变量）**

  **Loop invariant:** At the start of each iteration of **for** loop, the subarray $A[1 .. j-1]$ consists of the elements originally in $A[1 .. j-1]$ but in sorted order.

  It can help us to understand why an algorithm is correct.

- **Three properties about a loop invariant**

  （初值、维持、终值）（初始化、保存、终止）

  - **Initialization**: it is true prior to the first iteration of the loop.(base case)
  - **Maintenance**: if it is true before an iteration of the loop, it remains true before the next iteration. (inductive step) ( $[i-1]$ true => $[i]$ true )
  - **Termination**: When the loop terminates, the invariant show that the algorithms is correct. (stopping the "induction" when the loop terminates; the inductive step of mathematical induction is used infinitely. 类似于数学归纳法，但数学归纳法能用于无限情况)

  When the first two properties hold, the loop invariant is true prior to every iteration of the loop. It is similar to mathematical induction.

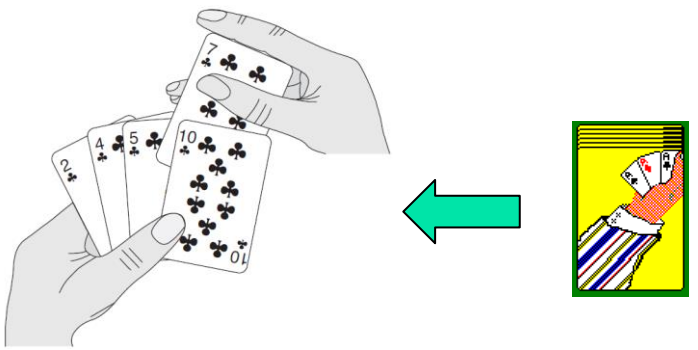
## 2.1.2 Loop invariants and the correctness of insertion sort

**Three properties hold for insertion sort**

Loop invariant: original $A[1..j\text{-}1]$ is permuted to $A'[1..j\text{-}1]$ but in sorted order

- **Initialization**: $j=2$, $A[1 .. j\text{-}1]$ consists of just the single $A[1]$ , which is the original element in $A[1]$, and is sorted. Obviously, the loop invariant holds prior to the first iteration of the loop.
- **Maintenance**: $A'[1 .. j\text{-}1]$ is in sorted order, sorting $A[j]$ into $A'[1 .. j\text{-}1]$, s.t. $<a'_1, a'_2, \ldots, a'_k, a_j, a'_{k+1} \ldots, a'_{j-1}>$ , obviously, it is sorted.The second property holds for the outer loop.
- **Termination**: when $j = n+1$, $A'[1 .. j\text{-}1] = A'[1 .. n]$ consists of the elements originally in $A[1 .. n]$, but in sorted sorder! Hence, the algorithm is correct.

```
INSERTION-SORT(A)
        for( j = 2; j <=length[A]; j++)
        {       key = A[j]
                i←j-1
                while( i > 0 && A[i] > key)
                {       A[i+1] = A[i]
                        i = i-1
                }
                A[i+1] = key
        }
```



# Analyzing an algorithm means predicting the resources

- Resources:
  - computational time (in CPU, I/O, …)
  - space (memory)
  - communication bandwidth
  - primary concern computer hardware
- By analyzing several candidate algorithms for a problem, a most efficient one can be identified, but several inferior are discarded（最优方案选取）

# 2.2 Analyzing algorithms -- Model

几种等价的计算模型：原始递归函数，λ 演算，图灵机，RAM

Computing model (计算模型)

- RAM, Random-access machine 随机存取器（冯.诺依曼结构）：带着可随机访问的内存，主要是能用电子元器件做出来。

- 原始递归函数（哥德尔，数学意义上）：一种能停机的递归函数，可以用图灵机计算。
- λ演算（丘奇，数理逻辑意义上）：可以被称为最小的通用程序设计语言。它包括一条变换规则 (变量替换) 和一条函数定义方式，λ演算之通用在于，任何一个可计算函数都能用这种形式来表达和求值。
- 图灵机（七元组的符合体系，抽象的计算机器）：状态自动转移，就是机器指令的例行程序。但指令不能存储，不能形成程序结构。

# 2.2 Analyzing algorithms -- Model

**RAM model: Algorithms will be implemented as computer programs.**

- **RAM, a generic one-processor, instructions are executed one after another, with no concurrent operations.**
- **RAM contains instructions commonly (basic instructions)**
  - **Arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling)**
  - **Data movement (load, store, copy)**
  - **Control (conditional and unconditional branch, subroutine call and return)**
- **Each such instruction takes a constant amount of time.**

# 2.2 Analyzing algorithms -- Challenge

**Analyzing a simple algorithm can be a challenge**

- **Mathematics required: probability theory, combinatorics, algebraic dexterity**
  （数学功底好）
- **An ability to identify the most significant terms in a formula**
  （洞察能力强，能抓住最重要的部分）
- **Summarizing an algorithm in simple formulas**
  （抽象思维强，能将算法的本质特征归纳成简单形式）

## 2.2.1 Analysis of insertion sort -- Running time

**Running time of a program is described as a function of the size of its input.** （通常，程序的运算时间是输入规模的函数）

- **input size *n***: *number of items in the input*, depends on the problem

  sorting array A: $n$ = length of A

  graph: $n$ can be the numbers of verices $V$ and edges $E$

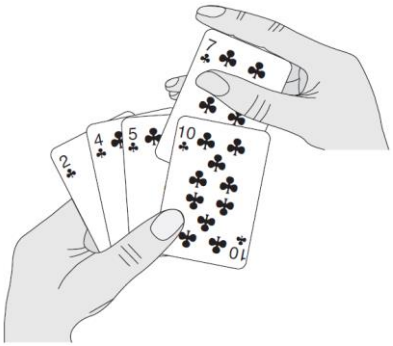
- **Running time *T*(*n*)**: on a particular input, *the number of primitive operations or "steps"* executed （基本操作数）

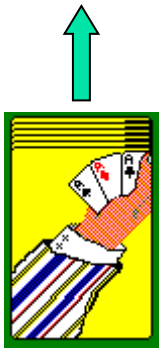
  运行时间满足如下一致性原则

  - The steps are machine-independent（基本操作独立于机器）
  - A constant amount of time $c_i$ is required to execute the $i$th line pseudocode（每行指令的执行时间为常数）

# 2.2.1 Analysis of insertion sort

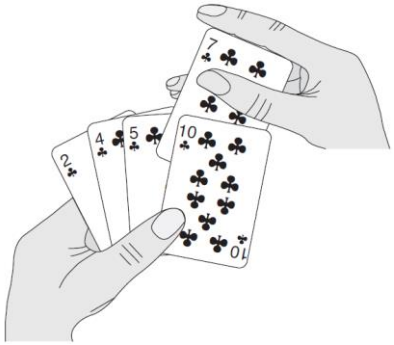
| INSERTION-SORT(*A*) | | cost | times |
|---|---|---|---|
| 1 | for( $j = 2; j <= length[A]; j++$) | $c_1$ | $n$ |
| 2 | { $key = A[j]$ | $c_2$ | $n-1$ |
| 3 | // Insert $A[j]$ into the sorted sequence $A[1 .. j-1]$ | 0 | $n-1$ |
| 4 | $i = j-1$ | $c_4$ | $n-1$ |
| 5 | while( $i > 0$ && $A[i] > key$) | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 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 | } | | |
| 9 | $A[i+1] = key$ | $c_8$ | $n-1$ |
| 10 | } | | |





$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

# Analysis of insertion sort



```
INSERTION-SORT(A)
1  for( j = 2; j <=length[A]; j++) // loop header
2  {       key = A[j]
3          // Insert A[j] into the sorted sequence
4          i←j-1
5          while( i > 0 && A[i] > key)
6          {     A[i+1] = A[i]
7                i = i-1
8          }
9          A[i+1] = key
10 }       // loop body below
```

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

**Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given.**

（相同输入规模的情况下，输入不同，运算时间也可能不同）

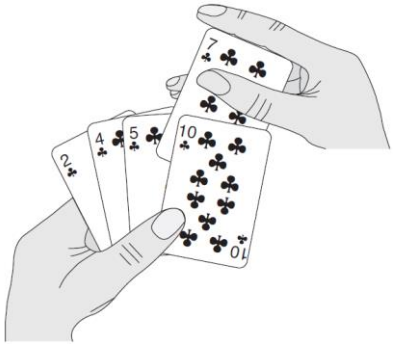
The best case occurs if the input is already sorted. For each $j$ = 2, 3, …, $n$, $A[i] \le key$ in line 5 when $i$ has its initial value of $j$-1. Thus $t_j$=1, and the best-case running time is

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

$$=(c_1+c_2+c_4+c_5+c_8)n-(c_2+c_4+c_5+c_8)=an+b$$

It is thus a **linear function** of $n$.

# Analysis of insertion sort



```
INSERTION-SORT(A)
1  for( j = 2; j <=length[A]; j++) // loop header
2  {       key = A[j]
3          // Insert A[j] into the sorted sequence
4          i←j-1
5          while( i > 0 && A[i] > key)
6          {     A[i+1] = A[i]
7                i = i-1
8          }
9          A[i+1] = key
10 }      // loop body below
```

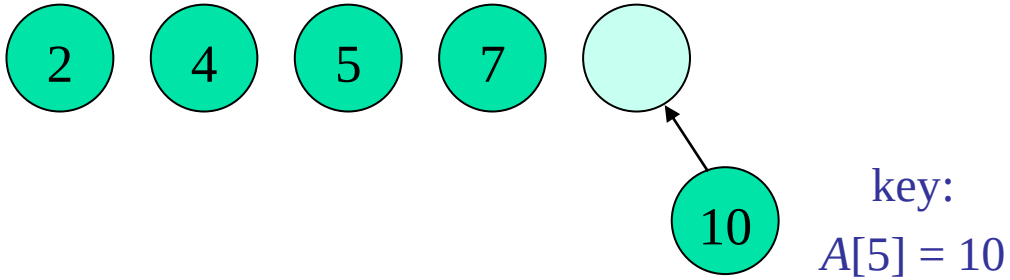
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

**Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given.**

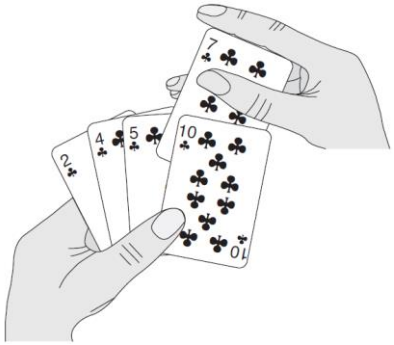
（相同输入规模的情况下，输入不同，运算时间也可能不同）

best case: $T(n) = an + b$

仅考虑第5行的 $A[i] > key$ 作为基本操作，每次for循环，$A[i] > key$ 仅执行一次（因为条件不成立，while循环体不执行），算法共执行 $n$-1 次比较。



# Analysis of insertion sort



```
INSERTION-SORT(A)
1  for( j = 2; j <=length[A]; j++) // loop header
2  {       key = A[j]
3          // Insert A[j] into the sorted sequence
4          i←j-1
5          while( i > 0 && A[i] > key)
6          {     A[i+1] = A[i]
7                i = i-1
8          }
9          A[i+1] = key
10 }      // loop body below
```

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

**An algorithm's running time may depend on factual input.**

The worst case results if the input is in reverse sorted order. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 .. j\text{-}1]$, and so $t_j = j$. Noting that

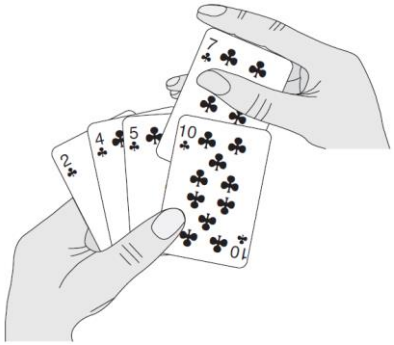
$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad , \qquad \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

Then

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

It is thus a **quadratic function ($n^2$)** of $n$ .

# Analysis of insertion sort



```
INSERTION-SORT(A)
1  for( j = 2; j <=length[A]; j++) // loop header
2  {     key = A[j]
3        // Insert A[j] into the sorted sequence
4        i←j-1
5        while( i > 0 && A[i] > key)
6        {    A[i+1] = A[i]
7             i = i-1
8        }
9        A[i+1] = key
10 }    // loop body below
```

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

**An algorithm's running time may depend on factual input.**

Exercise:

worst case: $T(n) = an^2 + bn + c$

仅考虑第5行的条件作为基本操作，算法共执行？次基本操作。

## 2.2.1 Analysis of insertion sort

best case

$$T(n) = an + b$$

worst case

$$T(n) = an^2 + bn + c$$

```
INSERTION-SORT(A)
1  for( j = 2; j <=length[A]; j++) // loop header
2  {       key = A[j]
3          // Insert A[j] into the sorted sequence
4          i←j-1
5          while( i > 0 && A[i] > key)
6          {     A[i+1] = A[i]
7                i = i-1
8          }
9          A[i+1] = key
10 }     // loop body below
```

插入排序的效率实际非常高！

对于好的输入，比快排还快很多！

## 2.2.2 Worse-case and average-case analysis

- **Worst-case: the longest running time for any input of size *n*. We usually concentrate on finding the worst-case.**
  - The worst-case running time of an algorithm is an upper bound on the running time for any input.
  - The worst case occurs fairly often. (in searching a databse)
    （最坏情况经常发生）
  - The "average case" is often roughly as bad as the worst case. (suppose that we randomly choose *n* numbers and apply insertion sort.)
  - 因此, OJ 上测试时，样例通过不算，记得测试各种边界条件
    （如：样例只给了5个数，但 $n \le 10^6$，程序需要考虑数据为$10^6$的情形）！
- **Average-case: or expected running time of an algorithm**
  - Probabilistic analysis (Knuth)
  - Randomized algorithm

## 2.2.3 Order of growth

$$T(n)=c_1n+c_2(n-1)+c_4(n-1)+c_5(\frac{n(n+1)}{2}-1)+c_6(\frac{n(n-1)}{2})+c_7(\frac{n(n-1)}{2})+c_8(n-1)=an^2+bn+c=\Theta(n^2)\approx n^2$$

**Rate of growth (order of growth，函数增长率)**

- the more simplifying abstraction.
- considering only the leading term of a formula (e.g., $\boldsymbol{an^2}$ of $\boldsymbol{an^2+bn+c}$), since the lower-order terms are relatively insignificant for large $n$.
- ignoring the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.
- Thus, the insertion sort has a worst-case running time of $\Theta(n^2)$.
- Usually, one algorithm is more efficient than another if its worst-case running time has a lower order of growth (small inputs exception, e.g., $100n\lg n$ vs $2n^2$ ).

**Rate of growth (order of growth，函数增长率)**

$T_1(n) = 100n\lg n < T_2(n) = 2n^2$ （ $n$ 很大时的含义 ）

数学与计算机算法的区别

- 数学看上去很美！计算机算法不但美，还很真实！
- 数学更多是研究无穷的情况，算法研究有限但很大的情况；
- 数学的无穷远处能想像，很美，但看不见够不着，算法很大的地方能想像，也很美，看得见但原始人力也够不着，计算机可以够得着。
- 数学帮助算法想像计算世界的美，算法帮助数学实现和走进计算世界的美。

# Exercise

编程实现该插入排序算法。产生规模为$10^5$的整数作为输入数据，用你的程序对数据进行排序。数据至少需要包括如下3组：

1. 正序数据；
2. 随机序数据；
3. 逆序数据。

对于每组数据，统计基本操作（第5行）执行的次数。结果说明了什么？

```
INSERTION-SORT(A)
1   for( j = 2; j <=length[A]; j++)
2   {   key = A[j]
3       // Insert A[j] into thesequence
4       i←j-1
5       while( i > 0 && A[i] > key)
6       {   A[i+1] = A[i]
7           i = i-1
8       }
9       A[i+1] = key
10  }
```

# 2.3 Designing algorithms

**many ways to design algorithms（模拟、分治、剪枝、回溯、贪心、...）**

**Sorting problem**

- Bubble sort: bubbling
- Insertion sort: incremental approach （增量靠近）
- Merge sort: divide-and conquer （特定位置，分而治之）
- Quick sort: location（精准定位，分而治之）

……

- The worst-case running time of divide-and-conquer algorithms is much less than that of insertion sort.

**Divide-and-conquer approach**（分而治之，各个击破）

**Recursive** : call themselves recursively one or more times to deal with closely related subproblems.

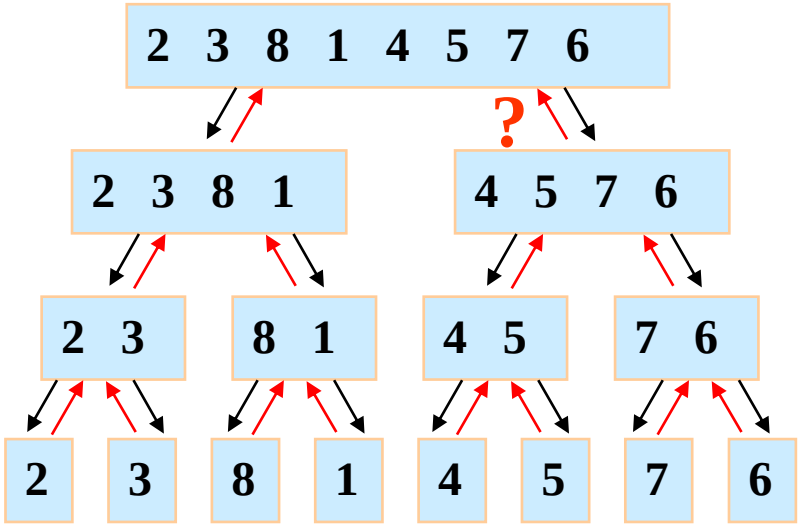
- **Divide** the problem into a number of subproblems.
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine** the sloutions to the subproblems into the solution for the original problem.

## Merge sort algorithm

- **Divide:** divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each.

  (Exercise: how about four subsequences, $n/4$ elements each?)
- **Conquer:** merge sort the two subsequences recursively.
- **Combine:** merge the two sorted subsequences to produce the sorted answer.

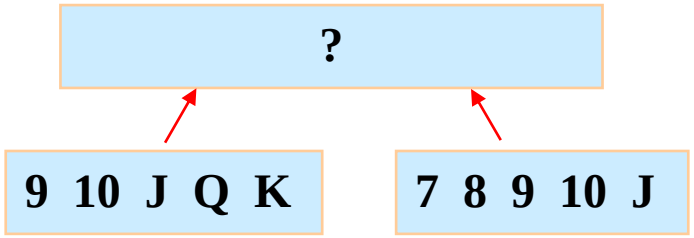
The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

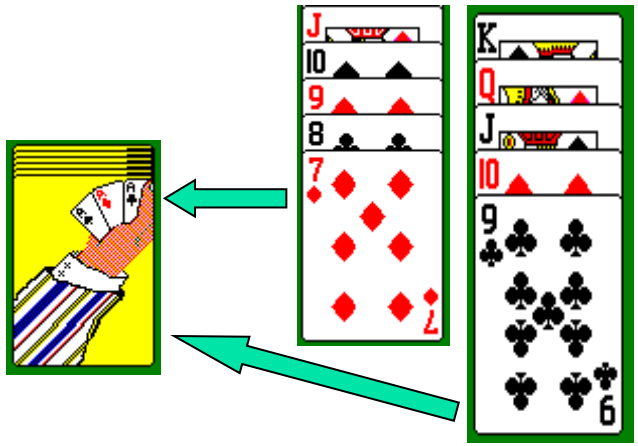


## 2.3.1 The divide-and-conquer approach

**MERGE(*A*, *p*, *q*, *r*):** the key operation of the MERGE sort algorithm, the merging of two sorted sequences in the "combine" step.

- ***A*** is an array and $p$, $q$, and $r$ are indices numbering elements of the array such that $p \le q < r$ .
- The procedure assumes the subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ are in sorted order. The procedure merges them to form a single sorted subarray that replaces the current subarray $A[p \,..\, r]$.
- Working step:

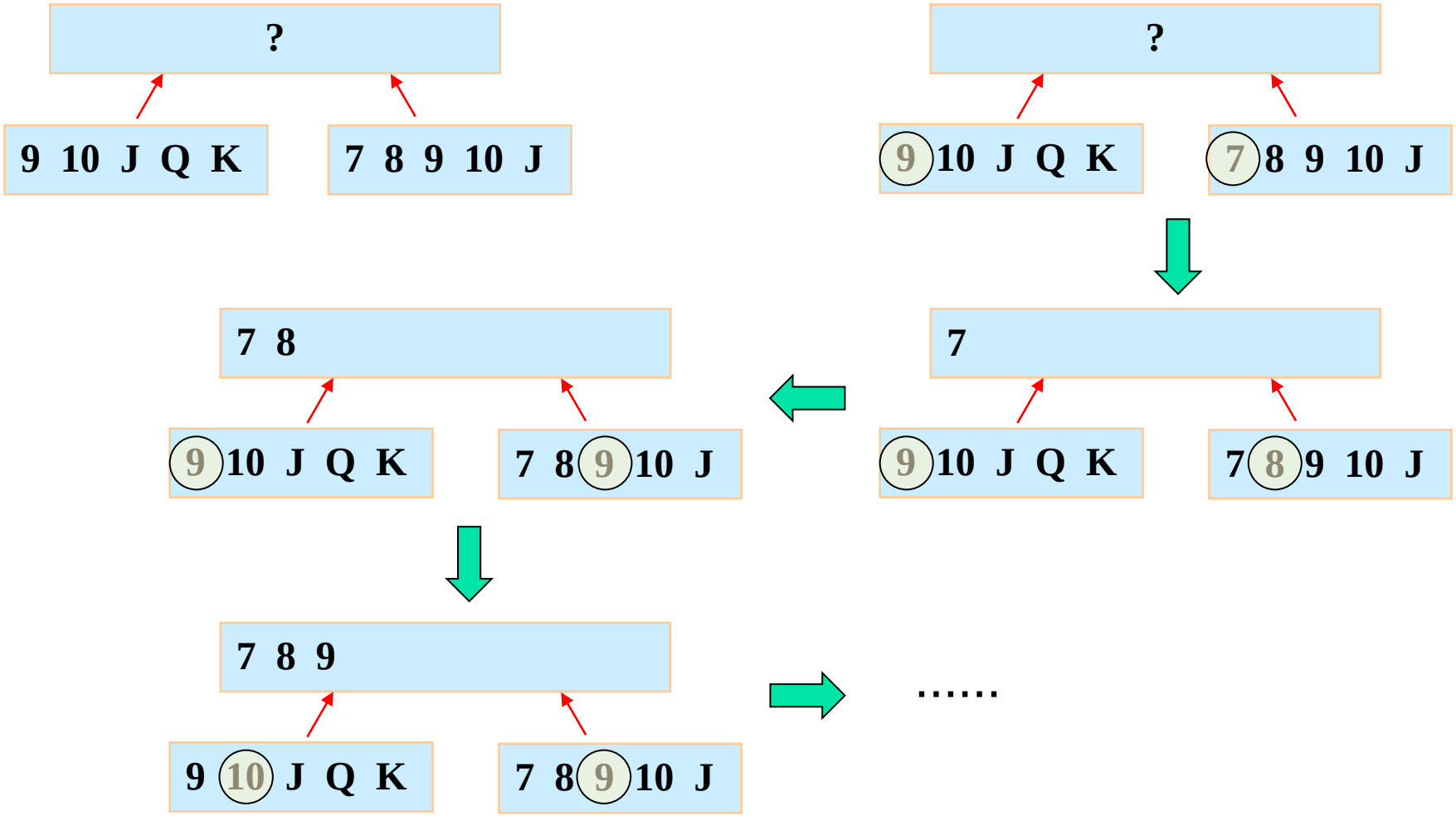


The procedure takes time $\Theta(n)$, $n = r-p+1$

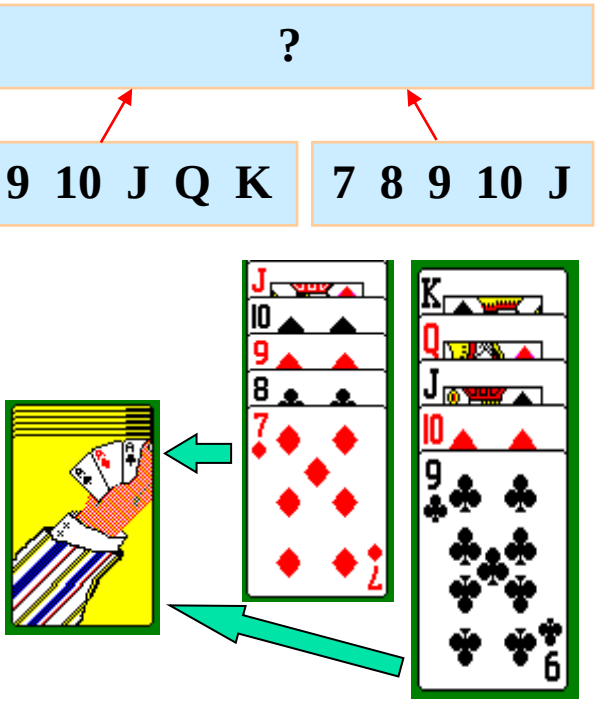
## 2.3.1 The divide-and-conquer approach

**MERGE(*A*, *p*, *q*, *r*):** $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ are in sorted order ⇒ $A[p \,..\, r]$



# 2.3.1 The divide-and-conquer approach

```
MERGE(A, p, q, r)
1    n1 ← q-p+1
2    n2 ← r-q
3    create arrays L[1 .. n1+1] and R[1 .. n2+1]
4    for i←1 to n1
5        L[i]←A[p+i-1]
6    for j←1 to n2
7        R[j]←A[q+j]
8    L[n1+1]←∞
9    R[n2+1]←∞
10   i←1
11   j←1
12   for k←p to r
13       if L[i]≤R[j]
14           A[k]←L[i]
15           i←i+1
16       else A[k]←R[j]
17           j←j+1
// ∞ : To avoid having to check whether either pile is empty in each
// basic step, a sentinel card is put on the bottom of each pile.
```

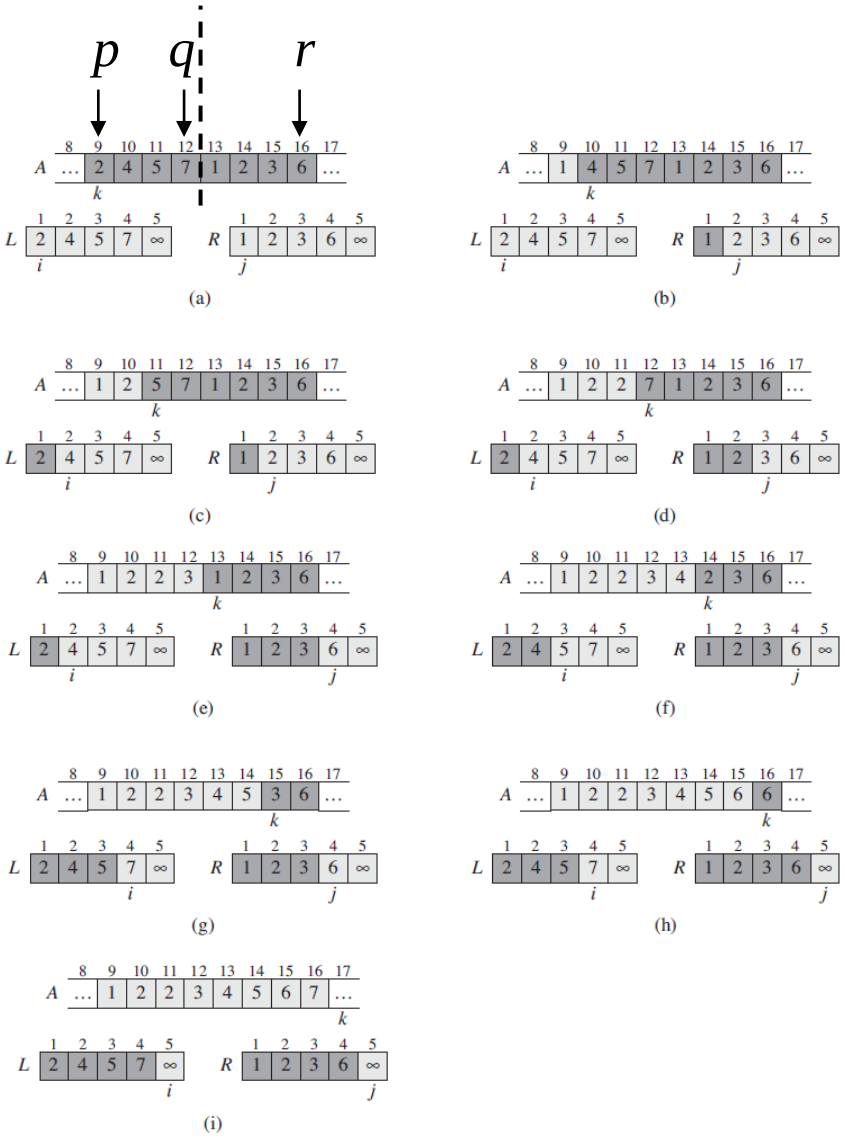



How does the subroutine work?

# 2.3.1 The divide-and-conquer approach

```
MERGE(A, p, q, r)
1    n1 ← q-p+1
2    n2 ← r-q
3    create arrays L[1 .. n1+1] and R[1 .. n2+1]
4    for i←1 to n1
5        L[i]←A[p+i-1]
6    for j←1 to n2
7        R[j]←A[q+j]
8    L[n1+1]←∞
9    R[n2+1]←∞
10   i←1
11   j←1
12   for k←p to r
13       if L[i]≤R[j]
14           A[k]←L[i]
15           i←i+1
16       else A[k]←R[j]
17           j←j+1
```

**Is the algorithm correct?**



# 2.3.1 The divide-and-conquer approach

```
MERGE(A, p, q, r)
1     n1 ← q-p+1
2     n2 ← r-q
3     create L[1 .. n1+1], R[1 .. n2+1]
4     for i←1 to n1
5         L[i]←A[p+i-1]
6     for j←1 to n2
7         R[j]←A[q+j]
8     L[n1+1]←∞
9     R[n2+1]←∞
10    i←1
11    j←1
12    for k←p to r
13        if L[i]≤R[j]
14            A[k]←L[i]
15            i←i+1
16        else  A[k]←R[j]
17            j←j+1
```

## properties of loop invariant for MERGE

- **Initialization**: prior to the first iteration of the loop, $k = p$, so that $A[p \,..\, k\text{-}1]$ is empty. $A[p \,..\, k\text{-}1]$ contains 0 samllest elements of $L$ and $R$, and since $i=j=1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.
- **Maintenance**: firstly suppose that $L[i] \le R[j]$. Because $A[p \,..\, k\text{-}1]$ contains the $k\text{-}p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \,..\, k]$ will contain the $k\text{-}p\text{+}1$ smallest elements. If $L[i] > R[j]$, so do that.
- **Termination**: $k=r\text{+}1$, subarray $A[p \,..\, k\text{-}1]$ is $A[p \,..\, r]$, contains $r\text{-}p\text{+}1$ smallest elements of $L[1 \,..\, n_1\text{+}1\,]$ and $R[1 \,..\, n_2\text{+}1]$, in sorted order. All but the two largest (sentinels) have been copied back into $A$.

# 2.3.1 The divide-and-conquer approach -- Running time

| | MERGE($A$, $p$, $q$, $r$) | cost | times |
|---|---|---|---|
| 1 | $n_1 \leftarrow q-p+1$ | $c$ | 1 |
| 2 | $n_2 \leftarrow r-q$ | $c$ | 1 |
| 3 | create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$ | $c$ | 1 |
| 4 | **for** $i \leftarrow 1$ **to** $n_1$ | $c$ | $n_1+1$ |
| 5 | $L[i] \leftarrow A[p+i-1]$ | $c$ | $n_1$ |
| 6 | **for** $j \leftarrow 1$ **to** $n_2$ | $c$ | $n_2+1$ |
| 7 | $R[j] \leftarrow A[q+j]$ | $c$ | $n_2$ |
| 8 | $L[n_1+1] \leftarrow \infty$ | $c$ | 1 |
| 9 | $R[n_2+1] \leftarrow \infty$ | $c$ | 1 |
| 10 | $i \leftarrow 1$ | $c$ | 1 |
| 11 | $j \leftarrow 1$ | $c$ | 1 |
| 12 | **for** $k \leftarrow p$ **to** $r$ | $c$ | $r-p+2$ |
| 13 | **if** $L[i] \le R[j]$ | $c$ | $r-p+1$ |
| 14 | $A[k] \leftarrow L[i]$ | $c$ | $x$ |
| 15 | $i \leftarrow i+1$ | $c$ | $x$ |
| 16 | **else** $A[k] \leftarrow R[j]$ | $c$ | $r-p+1-x$ |
| 17 | $j \leftarrow j+1$ | $c$ | $r-p+1-x$ |

$r - p + 1$

$= n_1 + n_2 = n$

$1 \le x \le n_1$

$\Theta(n_1+n_2)=\Theta(n)$

# 2.3.1 The divide-and-conquer approach -- Running time

```
MERGE(A, p, q, r)
1    n1 ← q-p+1
2    n2 ← r-q
3    create arrays L[1 .. n1+1] and R[1 .. n2+1]
4    for i←1 to n1
5        L[i]←A[p+i-1]
6    for j←1 to n2
7        R[j]←A[q+j]
8    L[n1+1]←∞
9    R[n2+1]←∞
10   i←1
11   j←1
12   for k←p to r      // 设 n = r-p，算法执行 n 次（每次循环，循环体语句执行常数次）
13       if L[i]≤R[j]  // 这是基本操作，比较 n 次
14           A[k]←L[i]
15           i←i+1
16       else A[k]←R[j]
17           j←j+1
```

**以后分析复杂度时，只关心基本操作即可。**

**哪个（些）是基本操作?**

$\Theta(n)$

## 2.3.1 The divide-and-conquer approach

- MERGE procedure as a subroutine.
- MERGE-SORT($A$, $p$, $r$) sorts the elements in the subarray $A[p \,..\, r]$.
- If $p \geq r$, the subarray has at most one element, already sorted, base case.
- Otherwise, the divide step computes an index $q$ that partitions $A[p \,..\, r]$ into two subarrays: $A[p \,..\, q]$, containing $\lfloor n/2 \rfloor$ elements, and $A[q+1 \,..\, r]$, containing $\lceil n/2 \rceil$ elements.

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

```
MERGE(A, p, q, r)
1   n1 ← q-p+1
2   n2 ← r-q
3   create arrays L[1 .. n1+1] and R[1 .. n2+1]
4   for i←1 to n1
5       L[i]←A[p+i-1]
6   for j←1 to n2
7       R[j]←A[q+j]
8   L[n1+1]←∞
9   R[n2+1]←∞
10  i←1
11  j←1
12  for k←p to r
13      if L[i]≤R[j]
14          A[k]←L[i]
15          i←i+1
16      else A[k]←R[j]
17          j←j+1
```

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

```
MERGE(A, p, q, r)
1   n1 ← q-p+1
2   n2 ← r-q
3   create arrays L[1 .. n1+1] and R[1 .. n2+1]
4   for i←1 to n1
5       L[i]←A[p+i-1]
6   for j←1 to n2
7       R[j]←A[q+j]
8   L[n1+1]←∞
9   R[n2+1]←∞
10  i←1
11  j←1
12  for k←p to r
13      if L[i]≤R[j]
14          A[k]←L[i]
15          i←i+1
16      else A[k]←R[j]
17          j←j+1
```

忽略递归过程！

函数重在接口！

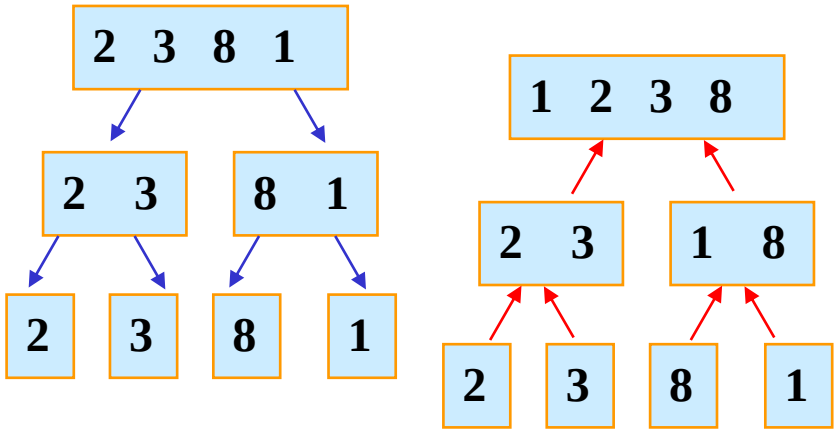
递归重在调用！

```
MERGE-SORT(A, 1, 4)
1   if 1 < 4
2       q ← ⌊(1+4)/2⌋ = 2
3       MERGE-SORT(A, 1, 2)
            1   if 1 < 2
            2       q ← ⌊(1+2)/2⌋ = 1
            3       MERGE-SORT(A, 1, 1)
                        1   if 1 < 1
            4       MERGE-SORT(A, 2, 2)
                        1   if 2 < 2
            5       MERGE(A, 1, 1, 2)
4       MERGE-SORT(A, 3, 4)
            1   if 3 < 4
            2       q ← ⌊(3+4)/2⌋ = 3
            3       MERGE-SORT(A, 3, 3)
                        1   if 3 < 3
            4       MERGE-SORT(A, 4, 4)
                        1   if 4 < 4
            5       MERGE(A, 3, 3, 4)
5       MERGE(A, 1, 2, 4)
```

思考题：自己画一下该递归的函数调用栈

## 2.3.1 The divide-and-conquer approach

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




```
MERGE-SORT(A, 1, 4)
1   if 1 < 4
2       q ← ⌊(1+4)/2⌋ = 2
3       MERGE-SORT(A, 1, 2)
            1   if 1 < 2
            2       q ← ⌊(1+2)/2⌋ = 1
            3       MERGE-SORT(A, 1, 1)
                        1   if 1 < 1
            4       MERGE-SORT(A, 2, 2)
                        1   if 2 < 2
            5       MERGE(A, 1, 1, 2)
4       MERGE-SORT(A, 3, 4)
            1   if 3 < 4
            2       q ← ⌊(3+4)/2⌋ = 3
            3       MERGE-SORT(A, 3, 3)
                        1   if 3 < 3
            4       MERGE-SORT(A, 4, 4)
                        1   if 4 < 4
            5       MERGE(A, 3, 3, 4)
5       MERGE(A, 1, 2, 4)
```

## 2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, its running time can often be described by a **recurrence equation** or **recurrence**, which described the overall running time on a problem of size $n$ in terms of the running time on smaller inputs.

We get the recurrence

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

- $D(n)$ is the time of dividing the problem into subproblems.
- $C(n)$ is the time of combining the solutions to the subproblems into the solution to the original problem.

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

## 2.3.2 Analyzing divide-and-conquer algorithms

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

Assume that the original problem size is a power of 2, the recurrence analysis is simplified. Each divide step then yields two subsequences of size exactly $n/2$ (the assumption does not affect the order of growth of the solution to the recurrence).

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

## 2.3.2 Analyzing divide-and-conquer algorithms

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

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

How to set up the recurrence for $T(n)$, the worst-case running time of merge sort on $n$ numbers?

- If one element $n$=1, takes constant time.
- When $n$>1,
  - **Divide**: One step, just computes the mid of the subarray, Thus, $D(n)=\Theta(1)$.
  - **Conquer**: Recursively solve two subprolems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
  - **Combine**: MERGE procedure shows $C(n)=\Theta(n)$ .

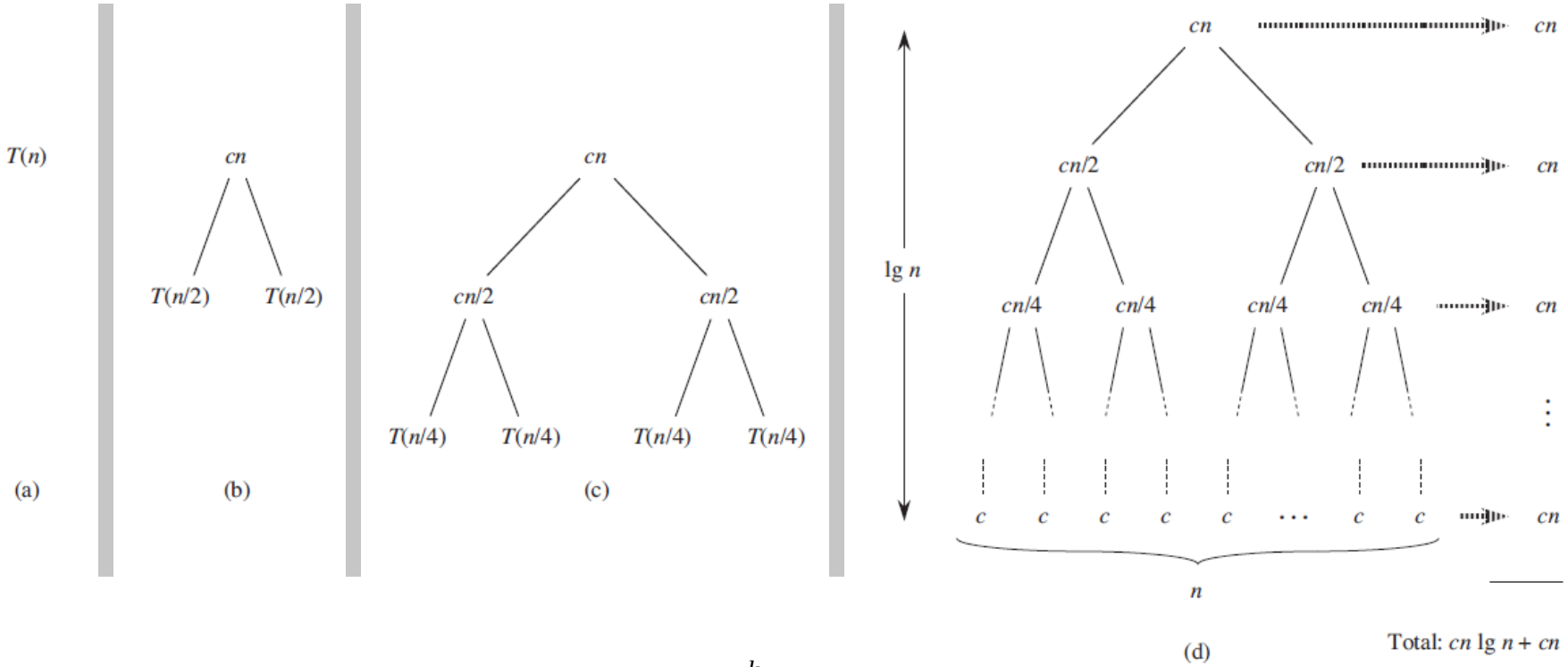
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

## 2.3.2 Analyzing divide-and-conquer algorithms

Rewrite the recurrence equation

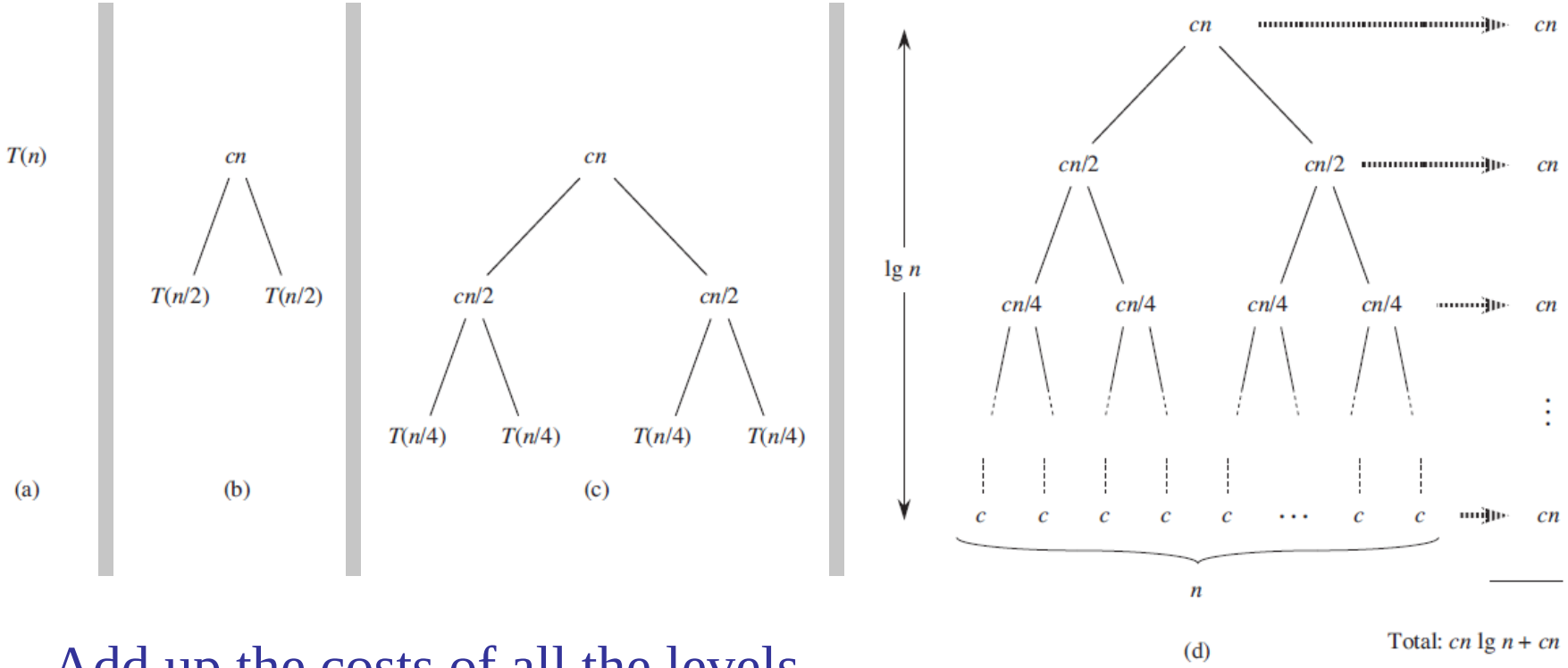
$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ 2T(n/2)+\Theta(n) & \text{if } n>1\end{cases} \Rightarrow T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ 2T(n/2)+cn & \text{if } n>1\end{cases}$$

solve the recurrence equation.




$$n/2^k = 1 \Rightarrow k = \lg n$$

## 2.3.2 Analyzing divide-and-conquer algorithms




Add up the costs of all the levels.

There are lg$n$+1 levels, each costing $cn$, for a total cost

$$cn(\lg n + 1) = cn\lg n + cn = \Theta(n\lg n)$$

obviously, it outperforms insertion sort, whose running time is $\Theta(n^2)$ , in the worst case, for large enough inputs.

# Exercises and problems

## Exercises

分析Fibonacci序列 $F(n) = F(n\text{-}1) + F(n\text{-}2)$ 如下算法的时间复杂度。

```
F(n)
    if(n<=2)
        return 1;
    else
        return F(n-1)+F(n-2);
```

[F(n), F(n-1)]

$$= [F(n\text{-}1), F(n\text{-}2)] * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2}$$

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

```
F(n)
    f1 = 1;
    f2 = 1;
    for(i=3; i<=n; i++)
    {
        f = f1 + f2;
        f1 = f2;
        f2 = f;
    }
```

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \frac{\varphi^n}{\sqrt{5}} - \frac{(1-\varphi)^n}{\sqrt{5}}$$

# Exercises and problems

## Exercises

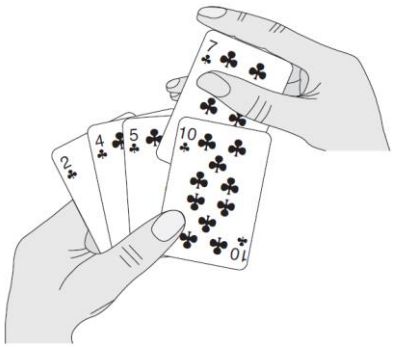
- 所有课后习题
- Horner's rule for evaluating a polynomial.
- 补充：自己编程实现几种常用的排序算法（插入、冒泡、选择、归并、快排、等等），并比较分析其运算时间。输入分别为正序、随机序、逆序的n个数，n的规模达到$10^5$。
- 写一篇关于排序的实现、对比、分析、应用相关的小论文
- 做一个快排动画的小软件

## Problems

- 所有课后Problems

# Exercises and problems

## Analysis of insertion sort



```
INSERTION-SORT(A)
1  for( j = 2; j <=length[A]; j++) // loop header
2  {      key = A[j]
3         // Insert A[j] into the sorted sequence
4         i←j-1
5         while( i > 0 && A[i] > key)
6         {    A[i+1] = A[i]
7              i = i-1
8         }
9         A[i+1] = key
10 }    // loop body below
```

**An algorithm's running time may depend on factual input.**

Exercise:

worst case: $T(n) = an^2 + bn + c$

仅考虑第5行的条件作为基本操作，算法共执行？次基本操作。