中国科学技术大学

算法基础

第一讲：算法概念

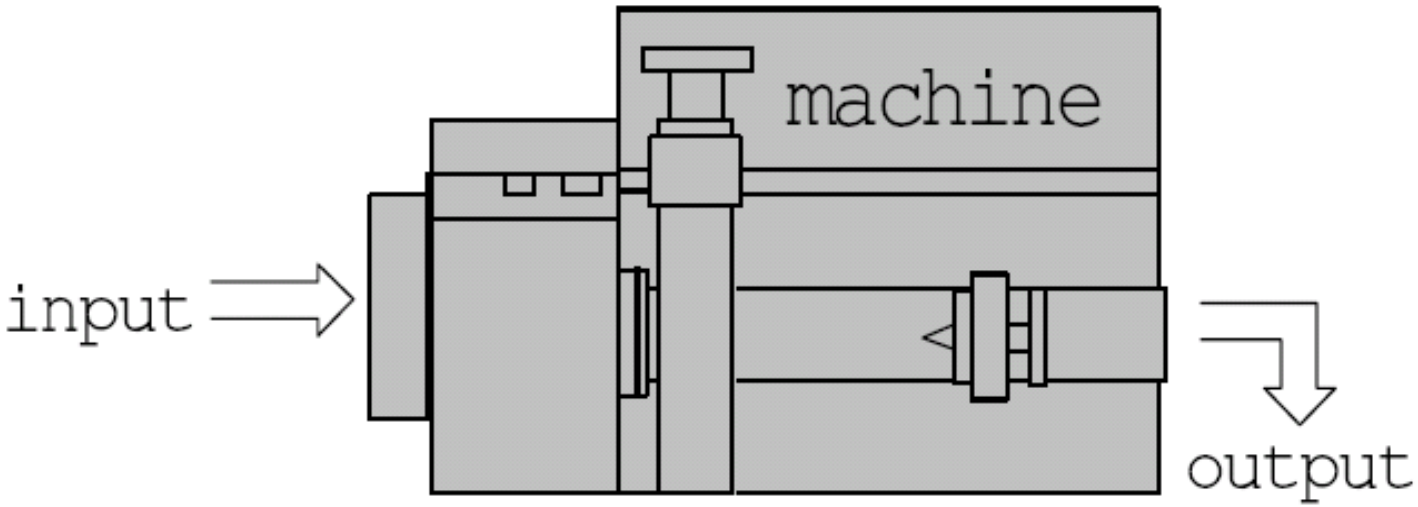
主　讲：顾乃杰　教授
单　位：计算机科学技术学院
学　期：2015—2016 (秋)

1. The Role of Algorithms

- What are Algorithms?
- Why is the Study of Algorithms Worthwhile?
- What is the Role of Algorithms?

1.1 Algorithms

- **Algorithm—算法**



$$x \xrightarrow{\text{input}} \sum \xrightarrow{\text{output}} y$$

概念：Problem & Instance

- **Problem—问题：**

  The general terms the desired input/output relationship. A computational Problem is a specification of the desired input-output relationship.

- **Instance of a problem--问题实例：**

  An Instance of a problem is all the inputs needed to compute a solution to the problem.

举例：Problem & Instance

- **Sorting Problem:**

  Input — A sequence of n numbers $(a_1, a_2, a_3, \ldots, a_n)$;

  Output—A permutation $(a_1', a_2', a_3', \ldots, a_n')$ of the input sequence such that $a_1' \le a_2' \le a_3' \le \ldots \le a_n'$

- **Instance of Sorting Problem :**

  任给一个输入序列，如： 23, 52, 39, 16, 41, 58,

  排序算法将其排序并输出结果：16, 23, 39, 41, 52, 58.

  Such an input sequence is called an instance of the sorting problem.

概念：算法的正确性

- Correctness of Algorithm--正确性：

  An algorithm is said to be correct if, for every input instance, it halts with the correct output. A correct algorithm solves the given computational problem

- Note

  Random algorithms and probabilistic algorithms are also useful, though sometimes they might not halt, or it might halt with an answer other than the desired one.

1.2 Algorithms as a technology

Rich Man

Smart Programmer

Heroes of Algorithm

Euclid

al-Khwarizmi

D. Hilbert

A. M. Turing

The word algorithm is derived from the name of Abu Ja'far Muhammad ibn Musa al-Khwarizmi, 9th-century Persian mathematician.

研究与学习算法的意义

- Need to demonstrate that our solution method terminates with the correct answer;
- Need to chose algorithm that easy to implement;
- Need to consider algorithms that are space and time efficient.
- Algorithms efficiency is more important than computer speed.
- ……

举例：效率比速度更重要

- Computer A: 10^9 insstructions per second, running insertion sort. craftiest programmer codes insertion sort, requires $2n^2$ instructions.
- Computer B: 10^7 insstructions per second, running mergesort, an average programmer codes, requires $50n \log n$ instructions.
- Computer A is 100 times faster than computer B.

举例：效率比速度更重要

- To sort one million numbers, computer A takes:

$$\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}} = 2000 \text{ seconds}$$

- Computer B takes:

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 100 \text{ seconds}$$

- Computer B runs 20 times faster than computer A!

2. Getting Started

- Pseudocode: 伪代码

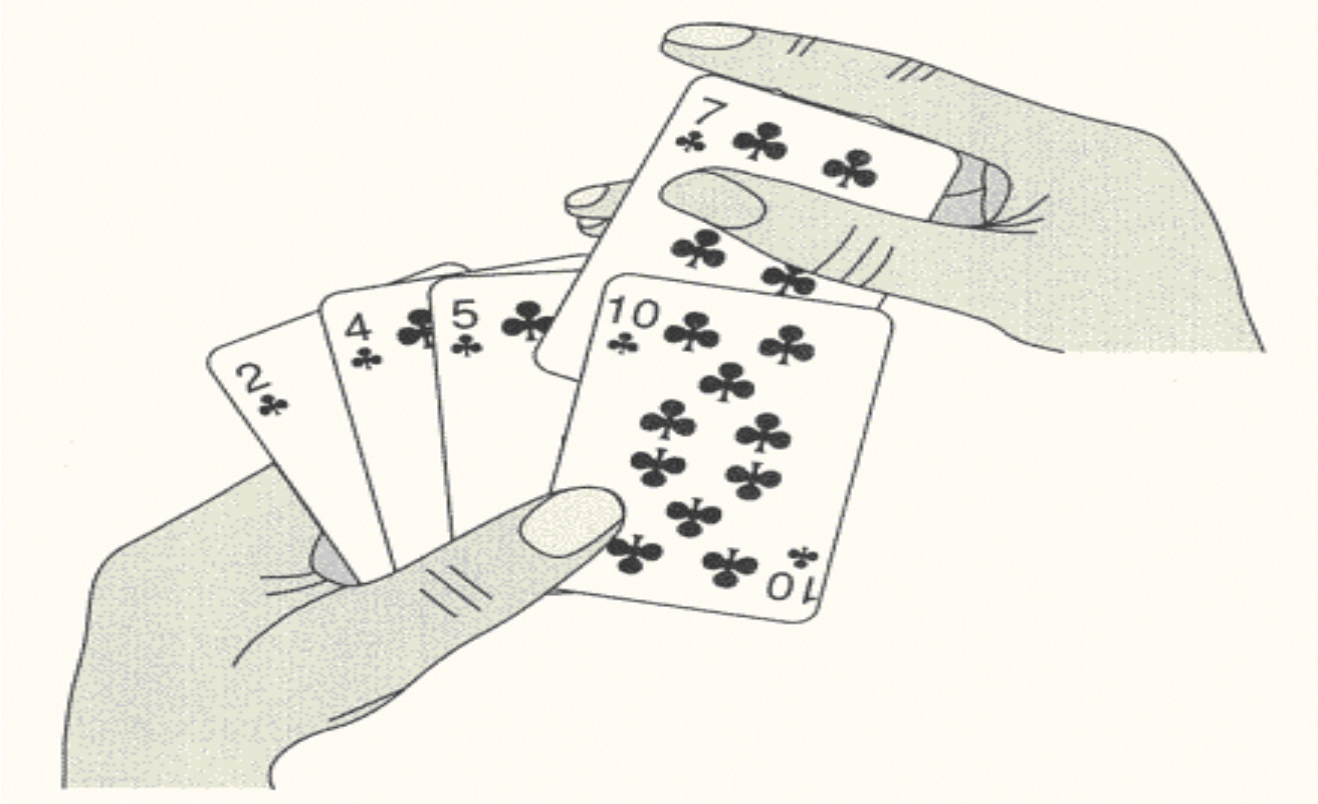
  we shall typically describe algorithms as programs written in a ***pseudocode*** that is similar in many respects to C, Pascal, or Java.

- The Difference between pseudocode and "real" code

  In pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm;

  Pseudocode is not typically concerned with issues of software engineering.

2.1 算法举例：Insertion Sort



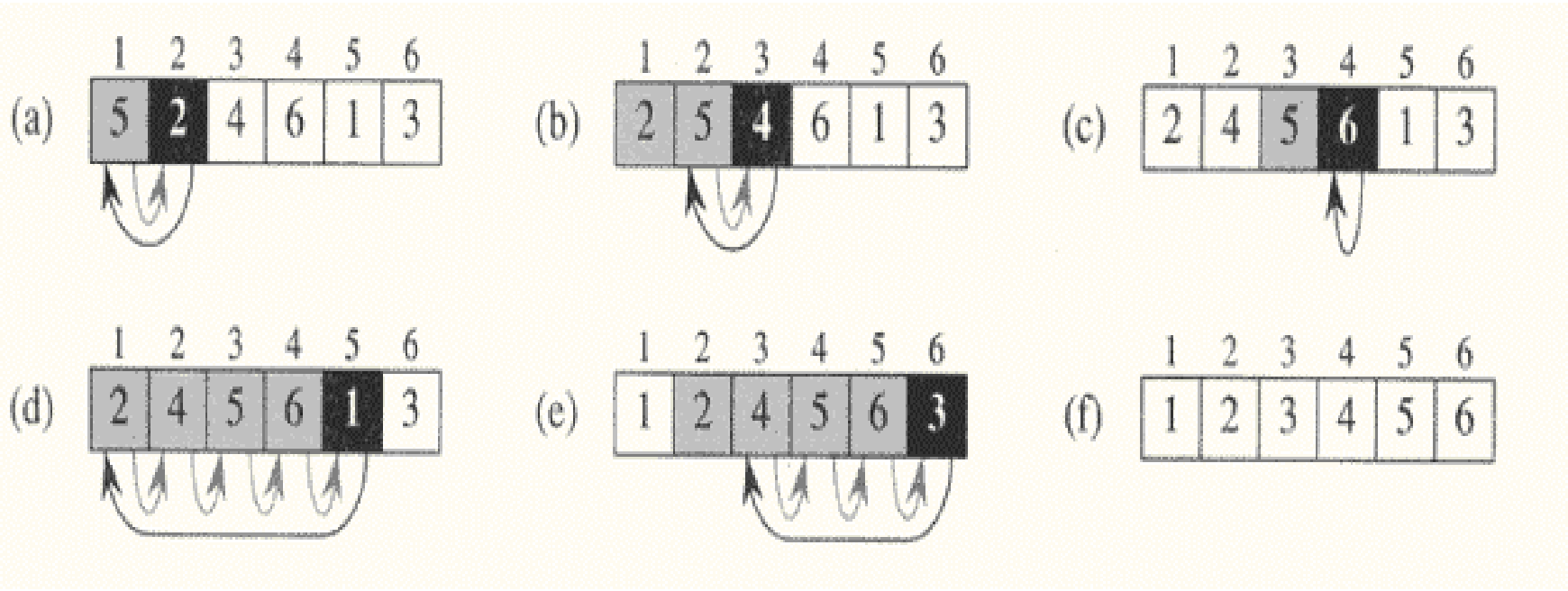
Insertion-Sort in Pseudocode

In-Place Sort:

Use only a fixed amount of storage beyond that needed for the data.

```
INSERTION-SORT(A)
1 for j ← 2 to length[A]
2     do   key ← A[j]
3    ★ Insert A[j] into the sorted sequence A[1… j - 1].
4          i ← j - 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                  i ← i - 1
8          A[i + 1] ← key
```

Example for Insertion-Sort



Insertion-Sort on the array $A = 5, 2, 4, 6, 1, 3$

Correctness and **loop invariant**

- We often use a ***loop invariant*** to help us understand why an algorithm gives the correct answer.
- Here.s the loop invariant for INSERTION-SORT:

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

Correctness and **loop invariant**

- To use a loop invariant to prove correctness, we must show three things about it:
- **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, the invariant--usually along with the reason that the loop terminated--gives us a useful property that helps show that the algorithm is correct.

Correctness and **loop invariant**

- Using loop invariants is like mathematical induction:
- To prove that a property holds, you prove a base case and an inductive step.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the "induction" when the loop terminates.

Correctness of **insertion sort:**

- **Initialization:** Just before the first iteration, $j = 2$. The subarray $A[1 \ldots j-1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.
- **Maintenance:** To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.
- **Termination:** The outer **for** loop ends when $j > n$; this occurs when $j = n + 1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the subarray $A[1 \ldots n]$ consists of the elements originally in $A[1 \ldots n]$ but in sorted order. In other words, the entire array is sorted!

Homework 2.1

Page 12, 2.1-1, 2.1-3

2.2 Analyzing algorithms

- Usually we only want to measure computational time（运行时间） and/or Memory（内存空间） used;
- Occasionally, resources such as communication bandwidth, or computer hardware are concerned;
- Generally, by analyzing several candidate algorithms for a problem, find out a most suitable and most efficient one.
- Dependent on architecture, Model of computation, Sequential (RAM 模型), Parallel (PRAM 模型).

Computational Model

- Before we can analyze an algorithm, we need a model for the resources and the implementation costs of algorithms.
- ***Random-access machine*** (***RAM***) is a widely used one-processor model of computation.
- In the RAM model, instructions are executed one after another, with no concurrent operations.

RAM Model

Instructions -- Commonly found in real computers;

- Arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling);
- Data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return);
- Each such instruction takes a constant amount of time.

RAM Model

In RAM model, compute 2^k takes one constant-time, (by shifting to the left), when k is no more than the number of bits in a computer word.

RAM Model

Each memory access takes exactly one time step, and we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk, which simplifies the analysis.

RAM Model

Data types--integer and floating point;

Do not concern with precision;

Assume a limit size of each word of data. For inputs size n, integers are represented by $c \log n$ bits for some constant $c \geq 1$.

RAM Model

To analyzing algorithms in RAM model, we need some tools:

Combinatorics,

Probability theory,

Algebraic exterity（灵巧、技巧）,

The ability to identify the most significant terms in a formula.

影响运行时间的主要因素

- Numbers of inputs;
- The distribution of input data; Some (not all) algorithms can take different amounts of time to sort two different input sequences of the same size .
- Usually describe the running time of a program as a function of the input size. Since (in general) the time taken by an algorithm grows with the size of input.
- The running time of an algorithm may depending on how the algorithm is implemented as well as what kind of data structure is used.
- We need to define the terms "running time" and "size of input".

概念：输入规模- - Input size

- ***Input size*** depends on the problem being studied.

  对于许多计算问题，其输入规模即为输入项的个数，例如：排序和计算离散付立叶变换 (DFT), 输入数组的元素个数 *n* 即为 ***input size***.

  For many other problems, ***input size*** is the *total number of bits* needed to represent the input in ordinary binary notation--such as multiplying two integers.

  Sometimes, it is better to describe the size of the input with two numbers rather than one -- For instance, if the input is a graph.

Need to indicate which input size is being used for each problem.

概念：运行时间--Running time

- The ***running time*** is the number of primitive operations or "steps" executed. The notion of step is machine independent.
- A constant amount of time is required to execute each line of pseudocode;
- One line may take a different amount of time than another line, but we shall assume that each execution of the *i*th line takes time C_i , where C_i is a constant.

概念：最好、最坏、平均运行时间

- Best Case Running Time:

  同样的输入规模，不同的数据分布情况下，最快情况的运行时间。

- Worst Case Running Time:

  同样的输入规模，不同的数据分布情况下，最慢或运行步数最多时的运行时间。

- Average Case Running Time:

  同样的输入规模，不同的数据分布情况下，平均所需的运行时间，通常指概率平均或期望值 。

算法分析举例：Insertion-Sort

INSERTION-SORT(A)	cost	times
1 **for** $j \leftarrow 2$ **to** $length[A]$	C_1	n
2 **do** $key \leftarrow A[j]$	C_2	$n-1$
3 /* Insert $A[j]$ into the sorted sequence $A[1 \cdots j-1]$. */	0	$n-1$
4 $i \leftarrow j-1$	C_4	$n-1$
5 **while** $i > 0$ and $A[i] >$ key	C_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	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	$n-1$

算法分析举例：Insertion-Sort

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5\sum_{j=2}^{n} t_j + C_6\sum_{j-2}^{n}(t_j - 1) + C_7\sum_{j-2}^{n}(t_j - 1) + C_8(n-1)$$

The best case : 如果输入序列已经有序, 则 $t_j = 1$

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1)$$

$$= (C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8)$$

$$= an + b \quad (\text{Linear function of } n)$$

算法分析举例：Insertion-Sort

The worst case：如果序列为降序，则：$t_j = j$

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(\frac{n(n+1)}{2} - 1) + C_6(\frac{n(n-1)}{2}) + C_7(\frac{n(n-1)}{2}) + C_8(n-1)$$

$$= an^2 + bn + c \quad \text{(a quadratic function of } n\text{)}$$

The average case：与数据的概率分布有关，假设对所有 $j = 2, 3, \ldots, n$ 有：$t_j = j/2$

$$T(n) = an^2 + bn + c \quad \text{(a quadratic function of } n\text{)}$$

Worst-case vs. Average-case

- Because it is so easy to cheat with the best case running time, we usually don't rely too much about it.
- The average running time is usually very hard to compute, we usually strive to analyze the worst case running time. The "average case" is often roughly as bad as the worst case.
- The worst-case running time is an upper bound on the running time for any input, it is usually fairly easy to analyze and often close to the average or real running time.

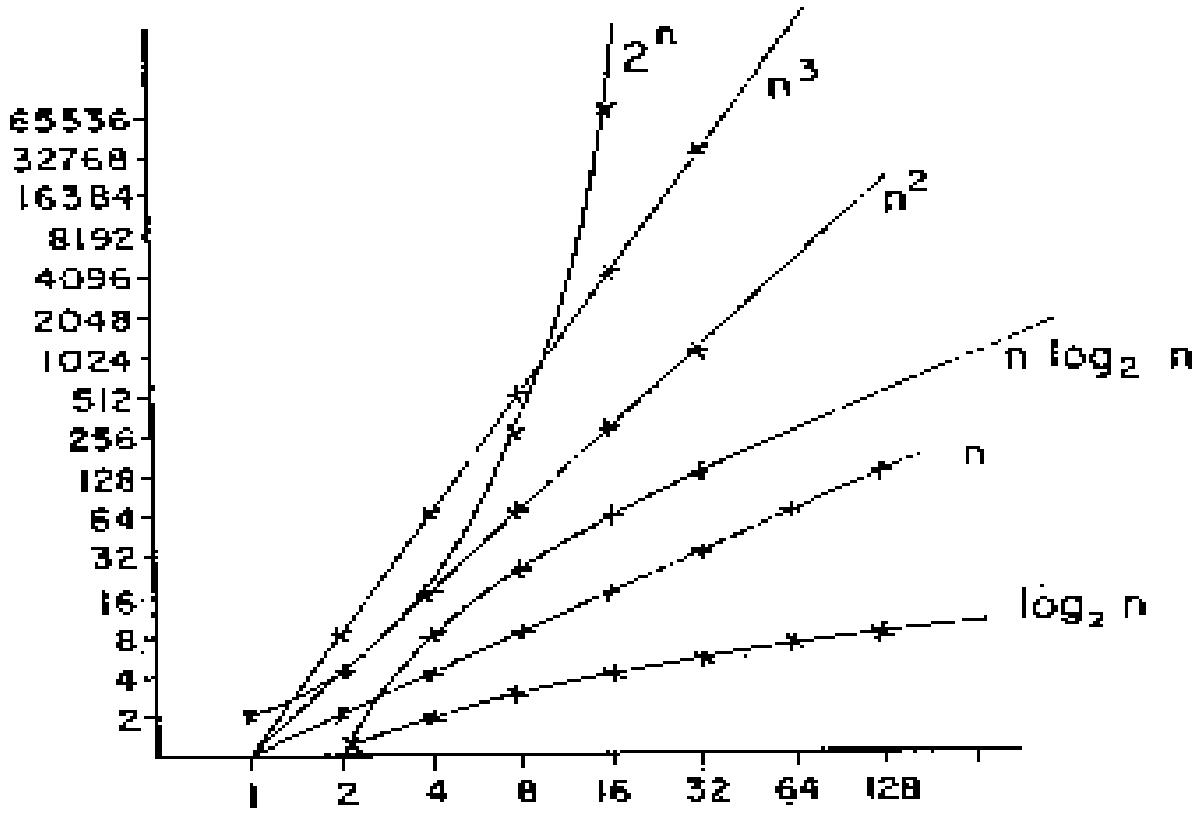
Order of growth

- 在算法分析过程中，通过抽象来简化分析过程，忽略每个语句的实际开销，代之以抽象的常数 C_i；
- Ignore not only the actual statement costs, but also the abstract costs C_i (using some constants a, b, and c).
- 对运行时间的增长量级(速度)感兴趣，只考虑运行时间表示式中的首项 (e.g. an^2).
- 忽略首项系数，如：Insertion sort 的最坏情况运行时间(又称最坏情况时间复杂度)为 $\Theta(n^2)$。

Order of growth

	Size n (Speed: 1M/s)					
Complexity	10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^3	.001 s	.008 s	.027 s	.064 s	.125 s	.216 s
n^5	.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	.001 s	1.0 s	17.9 min	12.7 days	35.7 years	366 centuries
3^n	.059 s	58 min	6.5 years	3855 centuries	$2x10^8$ centuries	1.3×10^{13} centuries

Rate of growth



Rate of growth of common computing time functions

Homework 2.2

- Page 16 , 2.2-2, 2.2-3

2.3 Designing algorithms

- Divide-and-conquer; (Chapter 2, 7, 9, 12, 28, 30)
- Greedy-Strategy; (Chapter 16, 23)
- Dynamic Programming; (Chapter 15, 25)
- Linear Programming; (Chapter 29)
- Backtracking; (Additional)
- Branch and Bound; (Additional)
- Other Methods. (Chapter 28, 31, 32)

The divide-and-conquer approach

- **Divide** problem into smaller subproblems;
- **Conque** subproblems by solving them recurisively;
- **Combine** solutions of subproblems into the solution for original problem;
- Problems can solved using Divide-and-Conque are ***recursive*** in structure
- Have three steps at each level of the recursion:

  Divide, Conque, Combine.

Merge Sort

- **Divide** the n-element sequence into two subsequences of $n/2$ elements each;
- **Conquer** (Sort) the two subsequences recursively using merge sort;
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

MERGE-SORT(A, p, r)

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

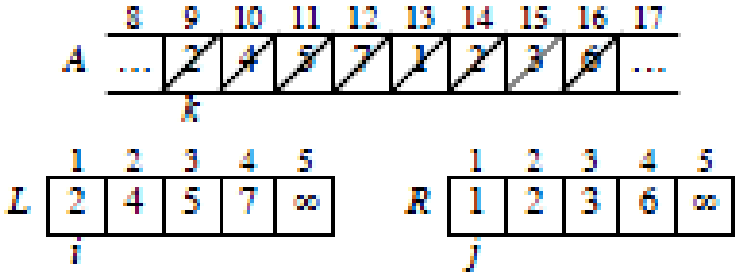
```
MERGE(A, p, q, r)
 1   n1 ← q - p + 1
 2   n2 ← r - q
 3   for i ←1 to  n1 do
 4        B[i] ← A[ p+i-1]
 5   for j ←1  to  n2 do
 6        C[j] ← A[q+j]
 7   B[n1 + 1] ← ∞
 8   C[n2 + 1] ← ∞
 9   i ← 1
10   j ← 1
11   for k ← p to r do
12        if B[i] ≤ C[j]
13             then A[k] ← B[i]
14                  i ← i + 1
15             else A[k] ← C[j]
16                  j ← j + 1
The End of Merge.
```

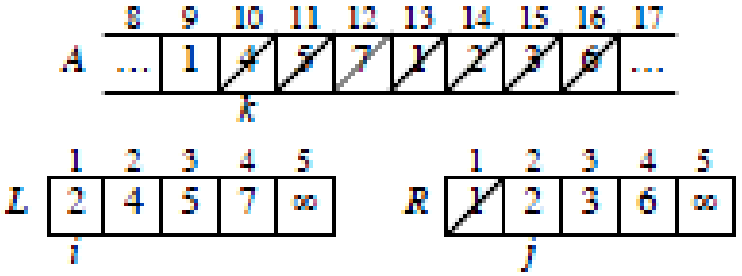
The merge procedure assumes that the subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ are in sorted order. The procedure ***merges*** them to a sorted subarray $A[p \ldots r]$.

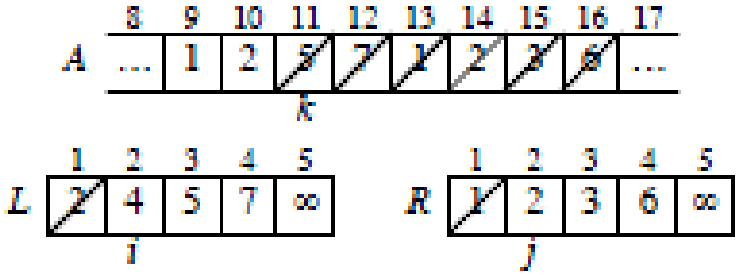
//增加监视哨,减少比较次数 //
//增加监视哨,减少比较次数 //

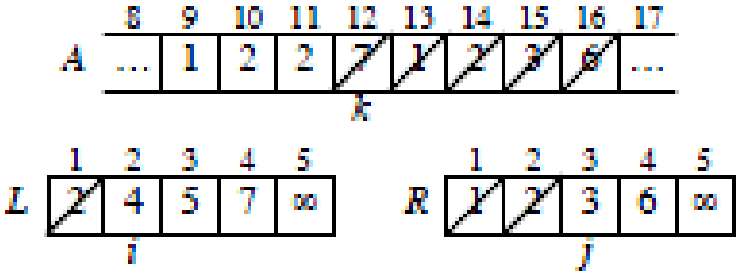
- **Merge 算法的运行时间：**
- The first two **for** loops take $\Theta(n1 + n2) = \Theta(n)$ time. The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
- Total time: $\Theta(n)$.

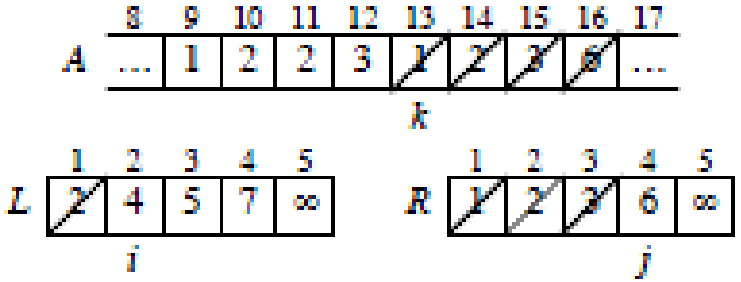
Example: A call of MERGE(9, 12, 16)

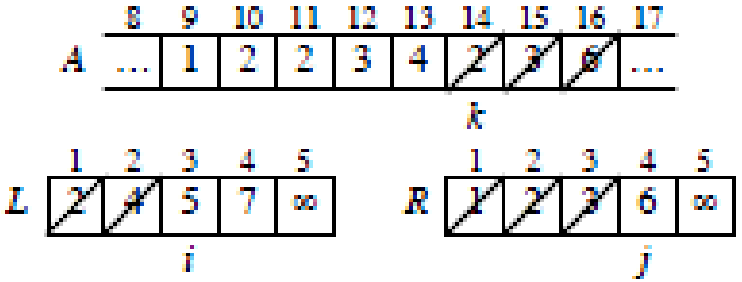


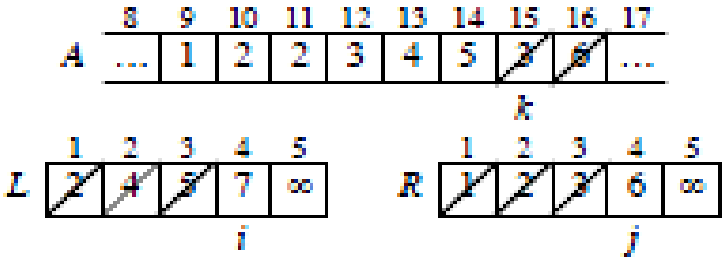


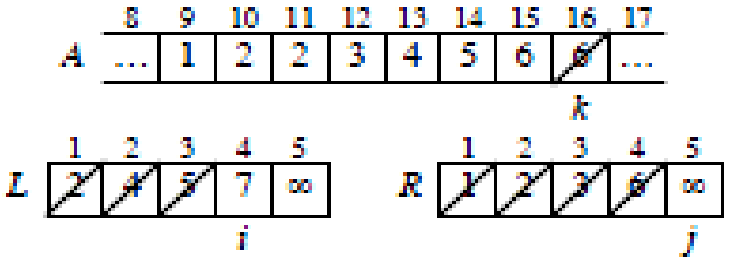


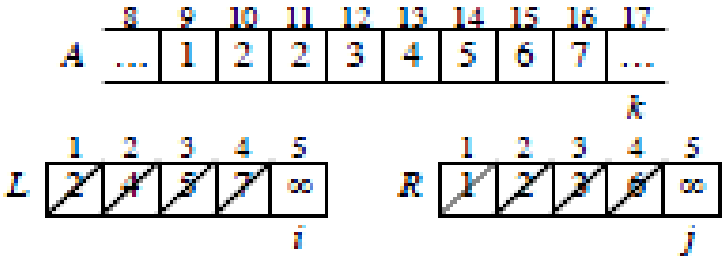







8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 ∞ ∞ …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 ∞ …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 ∞ ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 ∞ ∞
j

Sentinel--监视哨

- 在数组、线性表、序列的一端插入1个在数组（或序列）中不出现的特殊数字、字符， 从而减少控制变量的比较次数，达到提高效率的目的。这个特殊数字（字符）称为监视哨。

MERGE(A, p, q, r)-- Another Implementation

```
 1   for j = p  to  r  do
 2         B[j] = A[j]
 3    s = p
 4    t= q + 1
 5   i=p
 6   while (s ≤ q) and (t ≤ r)  do   //增加了比较次数//
 7         if B[s]≤ B[t]
 8               then A[i] = B[s]
 9                     s = s + 1
10               else  A[i] = B[t]
11                     t = t + 1
12          i = i + 1
13   if  s > q
14         then s = r+1
15               q = r
16   for j = s  to  q  do
17         A[i] = B[j]
18         i = i + 1
```

another implimentation of merge which doesn't use ***sentinel***.

分析基于分治法的算法

- Using ***recurrence equation*** to describe the running time of recusive algorithms;
- Using the running time on smaller inputs to describe the overall running time of size *n*.
- Using mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

提取递归方程

- Let $T(n)$ be the running time on a problem of size n;
- If the problem size is small enough, say $n \leq c$ for some constant c, then we can the solve it using constant time, write it as $\Theta(1)$;
- Suppose we need $D(n)$ time units to divide the problem of size n into a subproblems of size n/b each;
- Recursively solve subproblems, it takes $aT(n/b)$ running time;

提取递归方程

- Suppose we need $C(n)$ time units to combine the solutions of subproblems into the solution of original problem;
- We get the recurrence as follows:

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

Analysis of merge sort

To simplify the analysis, assume the problem size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$. This assumption does not affect the order of growth of the solution to the recurrence.

Analysis of merge sort

1) $T(n)$ is the worst-case running time of merge sort on n numbers.
2) Merge sort on just one element takes constant time.
3) When the number of elements $n > 1$, divide the sequence into two equal sized subsequences. We only need to compute the middle of the sequence, which takes constant time. Thus, $D(n) = \Theta(1)$.
4) Recursively solve two subproblems, each of size $n/2$, it takes $2T(n/2)$ running time.
5) The MERGE procedure on an n-element sequence takes $\Theta(n)$ running time, so $C(n) = \Theta(n)$.

Analysis of merge sort

- The worst-case running time $T(n)$ of merge sort is:

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

The solution of $T(n)$ is $\Theta(n \log n)$.

Homework 2.3

- Page 22, 2.3-2, 2.3-4,

思考：两个小问题

(1) 用ButtomUp的设计方式，通过两两归并实现归并排序，给出算法。

(2) 数组 $A[0..n-1]$ 循环左移 k 位；

循环左移问题

(1) 数组 $A[0..n-1]$ 循环左移 k 位；

```
Move—1 (A, k)
1.  reverse(0, k-1);
2.  reverse(k, n-1);
3.  reverse(0, n-1);
```

```
Reverse (A, p, r)
1.  mid ← [(r-p+1)/2];
2.  For i ← 0 to mid-1 do
3.        swap(A[p+i], A[r-i]);
```

内容读写： 约 3n 次

```
Move—2 (A, k)
1.  d ← gcd(n, k);
2.  For i ← 0 to d-1 do
3.        x ← A[i];
4         t ← i ;
5.        For j ← 1 to n/d-1 do
6.              A[t] ← A[(t+k) mod n ]
7.              t ← (t+k) mod n
8.        A[t] ← x;
```

内容读写： 约 n 次