算法设计与分析
Algorithms Design & Analysis

第一讲：概述

参考教材

- Thomas H. Cormen，《算法导论，第二版》，高等教育出版社（影印，英文）。
- 吴伟昶，《算法设计技巧与分析》，电子工业出版社，2005。

提醒：
不看教材很难会有好成绩
光看教材不保证有好成绩

提醒：
不看教材很难有好基础
光看教材很难有好发展

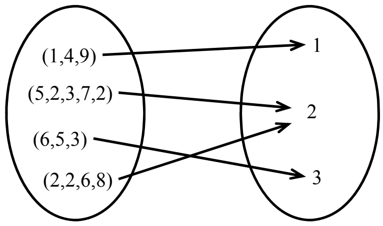
课程训练内容

- 相关基本概念
- 算法设计方法
- 算法分析技术
- 解决实际问题的能力

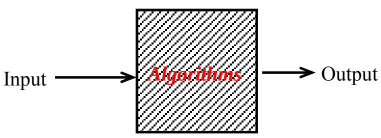
注意：
不包含算法在某种具体语言下的实现
不包含算法调试
是一门理论基础课程
编程实现是你自己的事情
（Program implementations are on your own）

从简单的问题开始

- 求集合中的极小值



解决问题



三要素：输入、输出、算法

算法在解决问题层面上具有“黑盒”特征

数据结构（Data Structure）

- 输入、输出和数据结构直接相关
- 数据之间的逻辑关系、数据在计算机上的存储方式、数据的操作。
- 数据结构与算法设计是密切相关的

算法（Algorithm）

- Informally, 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 ***algorithm*** is a step-by-step description of a procedure which, if followed closely, produces a **well-defined** result.
- 算法是为了求解问题而给出的指令序列。

程序只是算法的一种实现

7

算法表述

- 自然语言（ENGLISH）
- 算法描述语言（Pseudo-code）
- 计算机程序语言(C++, Java)
- 硬件设计（DSP）

8

算法一般特性

- 正确性：对于符合输入类型的任意输入数据，都产生正确的输出。
- Example: Given an election of 100,000,000 votes, determine the winner.（选举统计）

Approach 1: Count all 100,000,000 votes!

Approach 2: Select a sample of 1000,000 votes and declare the winner of the sample as the winner of the election.

==> with high probability, approach 2 will determine the winner correctly! But not always.(但是可以获得很高的正确性概率)

9

算法一般特性

- 有效性：每一步指令能够被有效的执行，并且规定了指令的执行效果，结果应该具有的数据类型，而且是可以预期的。
- 确定性：每一步之后都要有确定的下一步指令。
- 有穷性：有限步内结束

10

算法效率（ Efficiency）

- 有限资源

  Computational resources ：
  - CPU time (running time)计算时间
  - memory usage (space)存储空间
  - messages sent along the network网络带宽
- These resources should be used wisely，and algorithms that are efficient in terms of time or space will help you do so。(效率分析是为了有效的利用资源)

11

资源开销与输入

- **Resource consumption** differs depending on the size of the input(**资源开销与输入大小相关**)

  length of text(文本长度)

  number of records to be sorted(排序记录条目数量)
- **Resource consumption** may even differ greatly for inputs of the same size, depending on their structure(**资源开销与输入的组织结构有关**)

  highly unsorted input(无序)

  almost sorted input (有序)
- Specify ***resource consumption as a function of the inputs size***.随输入规模变化的资源开销函数

12

时空资源折中原理

- 对于同一个求解问题，一般会存在多种算法，这些算法在时空开销上的优劣往往表现出“时空折中”的性质，即为了改善一个算法的时间开销，往往可以通过增大空间开销为代价，设计出一个新算法来。

AB=Constant

算法设计与分析
Algorithms Design & Analysis

第二讲：算法渐进分析

华中科技大学软件学院
邱德红　主讲

渐进分析（Asymptotic Analysis）

- 回答→ **How does algorithm behave as the problem size gets very large?**
  - Running time(运行时间)
  - Memory/storage requirements(存储需求)
- **Remember that we use the RAM model(RAM模式):**
  - All memory equally expensive to access（存储空间访问开销均等）
  - No concurrent operations(顺序无并发)
  - All reasonable instructions take unit time
    - Except, of course, function calls(指令执行周期相同,除了程序调用之外)
  - Constant word size(字节长度一致)
    - Unless we are explicitly manipulating bits

渐进分析（续1）

- 渐进分析的目的是得到一个开销函数的渐进表达式，比如大O渐进表达式

$$\underset{n\to\infty}{rateT(n)} = O(f(n))$$

n：问题规模
T（*n*）：资源开销函数
f（*n*）：问题规模的整函数表达式

确定情况：矩阵求和（例）

```
void matrix_addition(double *M1, double *M2, int k)
{                                           cost    times
   for(int i=0; i<k; i++)                   c1      k+1
      for(int j=0; j<k; j++)                c2      k（K+1）
         M1[i][j]=M1[i][j]+M2[i][j];        c3      k·k
}
```

问题规模：$n=k^2$

$T(n)=c1(k+1)+c2k(k+1)+c3kk$
$=ak^2+bk+c=an+bn^{1/2}+c$

$$\underset{n\to\infty}{rateT(n)} = O(f(n))$$

$f(n)=n$

条件情况: 插入排序（Insertion Sort）

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = ∅   j = ∅   key = ∅
A[j] = ∅   A[j+1] = ∅

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = 2   j = 1   key = 10
A[j] = 30   A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 1   key = 10
A[j] = 30   A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 1   key = 10
A[j] = 30   A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 0   key = 10
A[j] = ∅   A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 0   key = 10
A[j] = ∅   A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

i = 2   j = 0   key = 10
A[j] = ∅   A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

i = 3   j = 0   key = 10
A[j] = ∅   A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

i = 3   j = 0   key = 40
A[j] = ∅   A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

i = 3   j = 0   key = 40
A[j] = ∅   A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

i = 3   j = 2   key = 40
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

i = 3   j = 2   key = 40
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
      }
      A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
     }
     A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```


An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

Done!

Insertion Sort

InsertionSort(*A*, *n*)	Cost	Repetitions
1 **for** $i = 2..n$	C_1	n
2 **do** $key \leftarrow A[i]$	C_2	$n-1$
3 $j \leftarrow i-1$	C_4	$n-1$
4 **while** $j > 0$ and $A[j] > key$	C_5	$\sum_{i=2}^{n} t_i$
5 **do** $A[j+1] \leftarrow A[j]$	C_6	$\sum_{i=2}^{n} (t_i - 1)$
6 $j \leftarrow j-1$	C_7	$\sum_{i=2}^{n} (t_i - 1)$
7 $A[j+1] \leftarrow key$	C_8	$n-1$

不同情况：

- 最好情况：输入为10、20、30、40

$$t_i = 1$$

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

不同情况：

- 最坏情况：输入为40、30、20、10

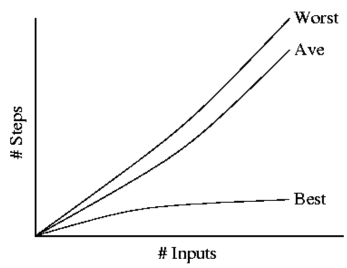
$$t_i = i$$

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

最坏、最好和平均情况

- The *worst case running time* of an algorithm is the function defined by the maximum number of steps taken on any instance of size.(输入量为n时的最大运行步骤数目)
- The *best case running time* of an algorithm is the function defined by the minimum number of steps taken on any instance of size n.(输入量为n时的最小运行步骤数目)
- The *average-case running time* of an algorithm is the function defined by an average number of steps taken on any instance of size n.(输入量为n时的平均运行步骤数目)

最坏、最好和平均情况(续)



Average case analysis

- Drawbacks(平均分析缺点)
  - Based on a probability distribution of input instances(与输入的分布有关)
  - How do we know if distribution is correct or not?（问题：如何确定输入的分布？）
- Usually more complicated to compute than worst case running time(通常比最坏情况计算困难)
  - Often worst case running time is comparable to average case running time(最坏情况通常比较接近于平均情况)

Worst case analysis

- Typically much simpler to compute as we do not need to “average” performance on many inputs(简单)
  - Instead, we need to find and understand an input that causes worst case performance(需要确定导致最坏情况下的输入)
- Provides guarantee that is independent of any assumptions about the input(独立于有关输入的假设)
- Often reasonably close to average case running time(通常接近平均情况)

渐进分析与阶的增长

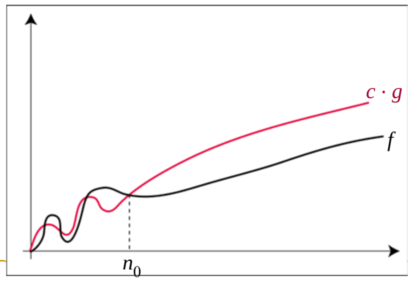
- 开销函数的估计是相对的而不是绝对的
- 独立于机器的算法开销估计
- 独立于实现技术的算法自身测度表示
- 关心的是大规模输入的情况

忽略了开销函数的低阶项和常数项，度量的是算法渐进的开销，此时，开销函数的阶的增长决定了开销的大小。

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

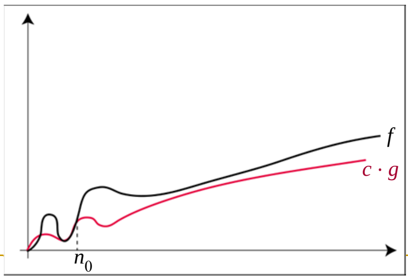
符号表示: O-Notation

$\boldsymbol{f(n) = O(g(n))}$ if $\exists c > 0, n_0 > 0$ s.t. $\forall n \geq n_0$: $f(n) \leq c \cdot g(n)$



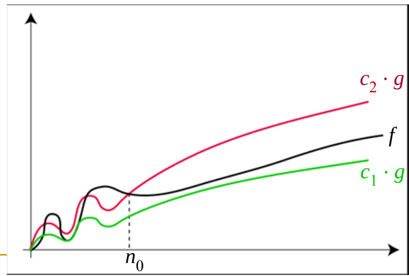
符号表示: Ω-Notation

$\boldsymbol{f(n) = \Omega(g(n))}$ if $\exists c > 0, n_0 > 0$ s.t. $\forall n \geq n_0$: $f(n) \geq c \cdot g(n)$



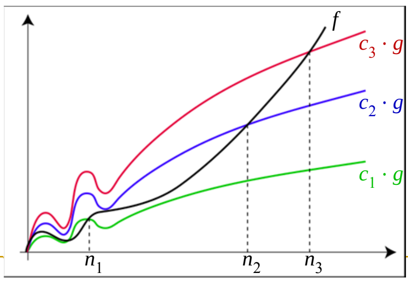
符号表示: Θ-Notation

$\boldsymbol{f(n) = \Theta(g(n))}$ if $\exists c_1, c_2 > 0, n_0 > 0$ s.t.
$\forall n \geq n_0$: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$



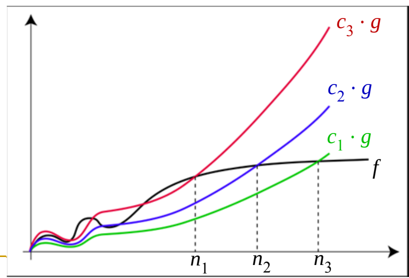
符号表示: ω-Notation

$\boldsymbol{f(n) = \omega(g(n))}$ if $\forall c > 0\ \exists n_0 > 0$ s.t. $\forall n \geq n_0$: $f(n) \geq c \cdot g(n)$



符号表示: o-Notation

$\boldsymbol{f(n) = o(g(n))}$ if $\forall c > 0\ \exists n_0 > 0$ s.t. $\forall n \geq n_0$: $f(n) \leq c \cdot g(n)$



A Few Simple Facts

- $f(n) = O(f(n)) \qquad f(n) = \Omega(f(n)) \qquad f(n) = \Theta(f(n))$
- $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
- $f(n) = O(g(n)) \Longleftrightarrow g(n) = \Omega(f(n))$
- $f(n) = o(g(n)) \Longleftrightarrow g(n) = \omega(f(n))$
- $f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \Rightarrow f(n) = \Theta(g(n))$
- $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n)) \Rightarrow f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$
- $f(n) = O(g(n)) \Rightarrow f(n) + g(n) = O(g(n))$

渐进分析和极限（Asymptotic Analysis and Limits）

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = o(g(n))$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, for some constant $c > 0$, then $f(n) = \Theta(g(n))$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $a^{f(n)} = o(a^{g(n)})$, for any $a > 1$.

$f(n) = o(g(n)) \Longrightarrow a^{f(n)} = o(a^{g(n)})$, for any $a > 1$.

$\cancel{f(n) = \Theta(g(n)) \not\Rightarrow a^{f(n)} = \Theta(a^{g(n)})}$

常见的规模的整函数表达式的*f(n)*

- *f(n)=1*, 常数函数，不依赖于*n*
- *f(n)=log n*, 对数函数，比线性函数增长慢
- *f(n)=n*
- *f(n)=n*^{2}
- *f(n)=nlog(n)*
- *f(n)=a*^{n}, 指数增长

算法效率比较

- **We consider algorithm *A* better than algorithm *B* if**
  （算法**A**好于算法**B**，如果满足以下公式）
  $T_A(n) = o(T_B(n))$

合并排序（Merge Sort）

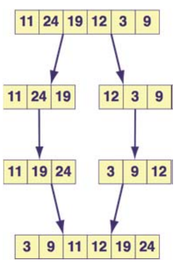
If $|A| = 1$, the data is already sorted; done.（一个元素的情况）

Otherwise:

Split the input into two pieces of equal size.（二等分）

Recursively sort these two pieces.（二部分递归排序）

Construct a sorted sequence from the two sorted subsequences.（整合排序好的两部分）



合并：Merge(*A, p, q, r*)

- Copy $A[p..q]$ and $A[q+1..r]$ to temporary arrays L and R（左右临时数组）

```
n1 ← q − p + r
n2 ← r − q
for i = 1..n1
do L[i] ← A[p + i − 1]
for j = 1..n2
do R[j] ← A[q + j]
```

- Create two artificial end markers that are never copied to *A*:（结束标志）

```
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
```

- Repeatedly copy the smallest element from *L* and *R* to *A*:（重复的将左右零时数组中最小的元素存储到A中）

```
i ← 1
j ← 1
for k = p..r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
             i ← i + 1
        else A[k] ← R[j]
             j ← j + 1
```

Example:

L: 3 5 15 28 30

R: 6 10 14 22 43 50

LSize: 5

RSize: 6

A:											

Example:

L: 3 5 15 28 30

i=0

R: 6 10 14 22 43 50

j=0

A:											

k=0

Example:

L: 3 5 15 28 30

i=0

R: 6 10 14 22 43 50

j=0

A:	3										

k=0

Example:

L: 3 5 15 28 30

i=1

R: 6 10 14 22 43 50

j=0

A:	3	5									

k=1

Example:

L: 3 5 15 28 30

i=2

R: 6 10 14 22 43 50

j=0

A:	3	5	6								

k=2

Example:

L: 3 5 15 28 30

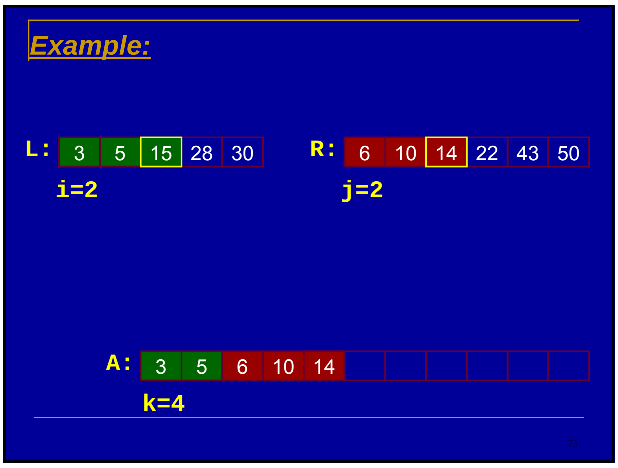
i=2

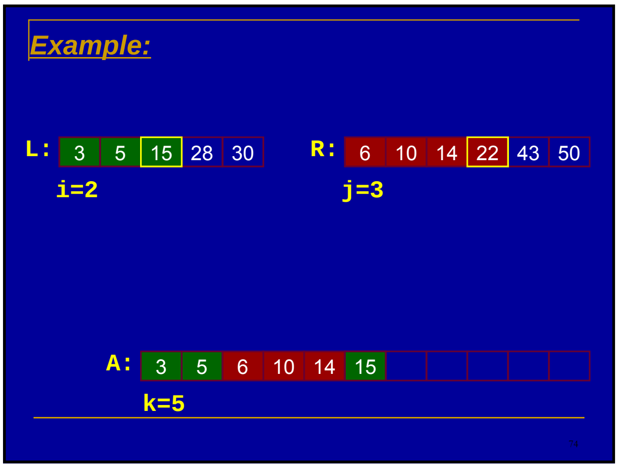
R: 6 10 14 22 43 50

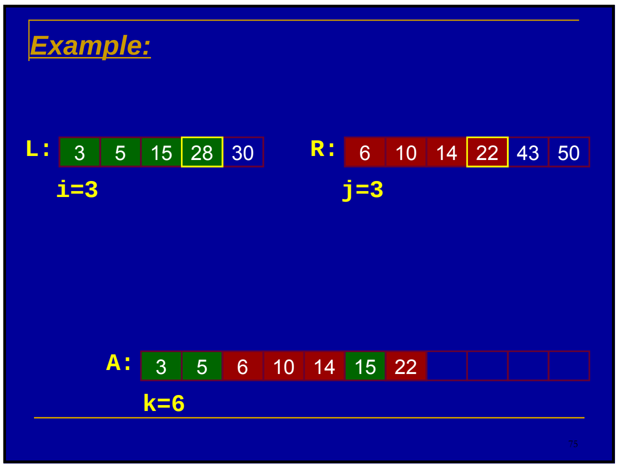
j=1

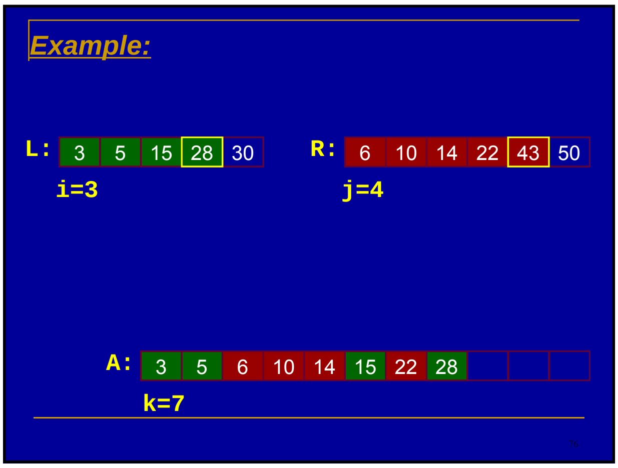
A:	3	5	6	10							

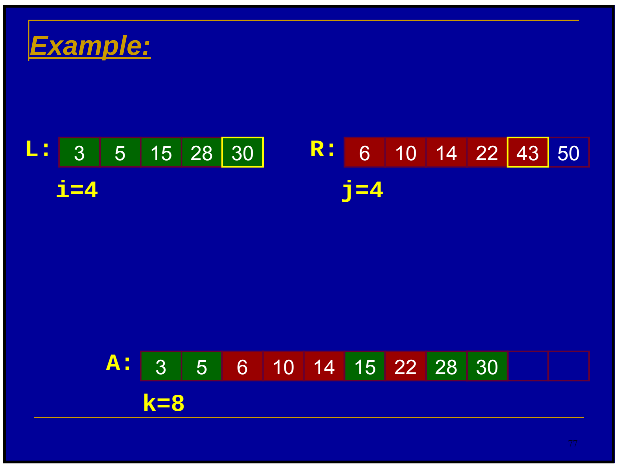
k=3

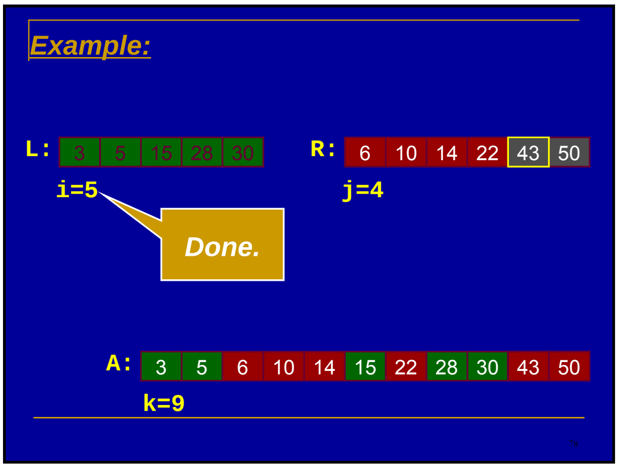

Example:
L: 3 5 15 28 30
i=2
R: 6 10 14 22 43 50
j=2
A: 3 5 6 10 14
k=4


Example:
L: 3 5 15 28 30
i=2
R: 6 10 14 22 43 50
j=3
A: 3 5 6 10 14 15
k=5


Example:
L: 3 5 15 28 30
i=3
R: 6 10 14 22 43 50
j=3
A: 3 5 6 10 14 15 22
k=6


Example:
L: 3 5 15 28 30
i=3
R: 6 10 14 22 43 50
j=4
A: 3 5 6 10 14 15 22 28
k=7


Example:
L: 3 5 15 28 30
i=4
R: 6 10 14 22 43 50
j=4
A: 3 5 6 10 14 15 22 28 30
k=8


Example:
L: 3 5 15 28 30
i=5
Done.
R: 6 10 14 22 43 50
j=4
A: 3 5 6 10 14 15 22 28 30 43 50
k=9

合并排序（Merge Sort）

MergeSort(*A*, *p*, *r*)

if $p < r$
then

Split the input into 2 pieces of approximately equal size

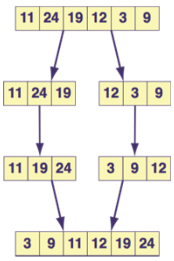
$q \leftarrow \lfloor (p + r) / 2 \rfloor$

Recursively sort the two halves

MergeSort(*A*, *p*, *q*)
MergeSort(*A*, *q* + 1, *r*)

Merge the two sorted subsequences

Merge(*A*, *p*, *q*, *r*)

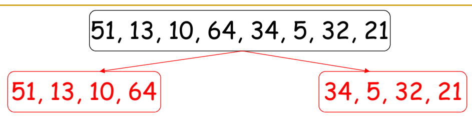


79

51, 13, 10, 64, 34, 5, 32, 21

we want to sort these 8 numbers,
divide them into two halves

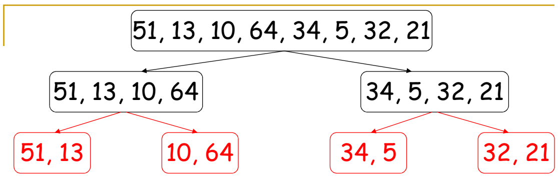
80



divide these
4 numbers
into halves

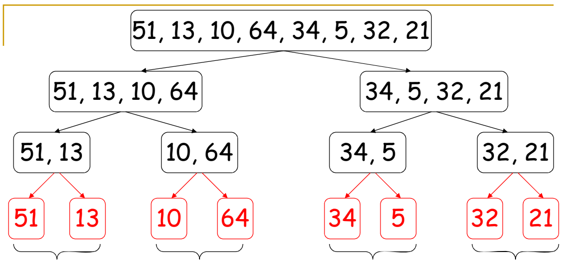
similarly for
these 4

81



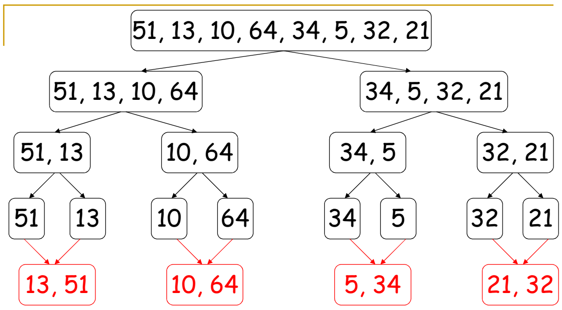
further divide each shorter sequence …
until we get sequence with only **1** number

82



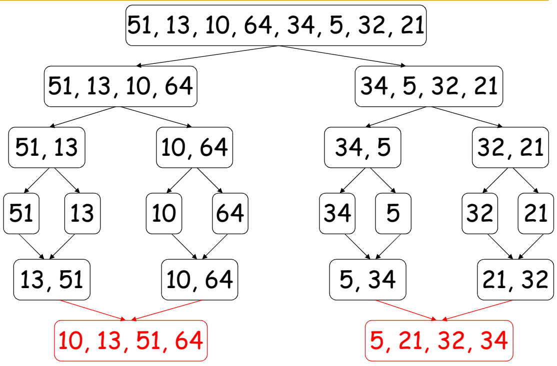
merge pairs of
single number
into a sequence
of 2 sorted
numbers

83

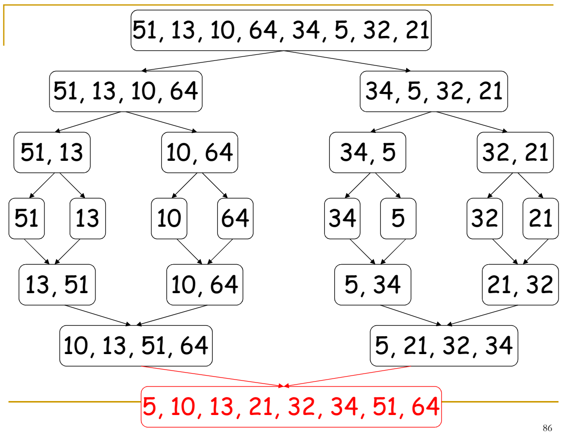


then **merge** again into
sequences of 4 sorted numbers

84



one more merge give the **final** sorted sequence



合并排序分析（Running Time of Merge Sort）

	$T(n) =$
MergeSort(*A*, *p*, *r*)	
if $p < r$	
then	
Split the input into 2 pieces of approximately equal size	
$q \leftarrow \lfloor (p + r) / 2 \rfloor$	$c_1 +$
Recursively sort the two halves	
MergeSort(*A*, *p*, *q*)	$T(n / 2) +$
MergeSort(*A*, *q* + 1, *r*)	$T(n / 2) +$
Merge the two sorted subsequences	
Merge(*A*, *p*, *q*, *r*)	$c_2 \cdot n$

小结（Summation）

- We have studied two sorting algorithms（两种排序算法）：
  - Insertion Sort takes $c_1 \cdot n^2$ time.（插入排序）
  - Merge Sort takes $c_2 \cdot n \lg n$ time.（合并排序）
- Even if $c_2 \gg c_1$, Merge Sort will ultimately outperform Insertion Sort（合并排序优于插入排序）