



第2-3章 程序性能与渐进分析

——精打细算：于微小处论得失



计算机学院

回顾

- 拷贝构造函数
 - `<类名> (类名 &);`
- 递归函数
 - 寻找“递归规律”和“终止状态”
- 程序测试
 - 黑盒测试：从输入输出和程序功能出发
 - 等价类划分
 - 白盒测试：从代码出发
 - 语句覆盖、分支覆盖、从句覆盖、执行路径覆盖



程序测试示例

- 题目：一个程序读入3个整数作为三角形的3条边，请输出信息，说明这个三角形是等边三角形？还是一般三角形？
- 如果别人写了一段程序完成上述题目，要求你来检查是否正确 ... 此时，
 - 你不了解他的实现细节，只能**黑盒测试**
 - 显然可能的类别应有3个：等边三角形、一般三角形、不是三角形
 - 因此你至少要设计3个测试用例：
(2, 2, 2), (3, 4, 5), (1, 2, 9)



程序测试示例

- 如果你自己写程序解决了上述题目，想检查一下是否存在错误 ... 此时，
 - 你了解实现细节，可以**黑盒测试**或**白盒测试**
 - **最好先绘制流程图**

L1: int a, b, c; cin>>a>>b>>c;

L2: if (a<=0 || b<=0 || c<=0 || a+b<=c || a+c<=b || b+c<=a)

L3: cout<<"不是三角形 ";

L4: else if (a==b&&b==c)

L5: cout<<"等边三角形";

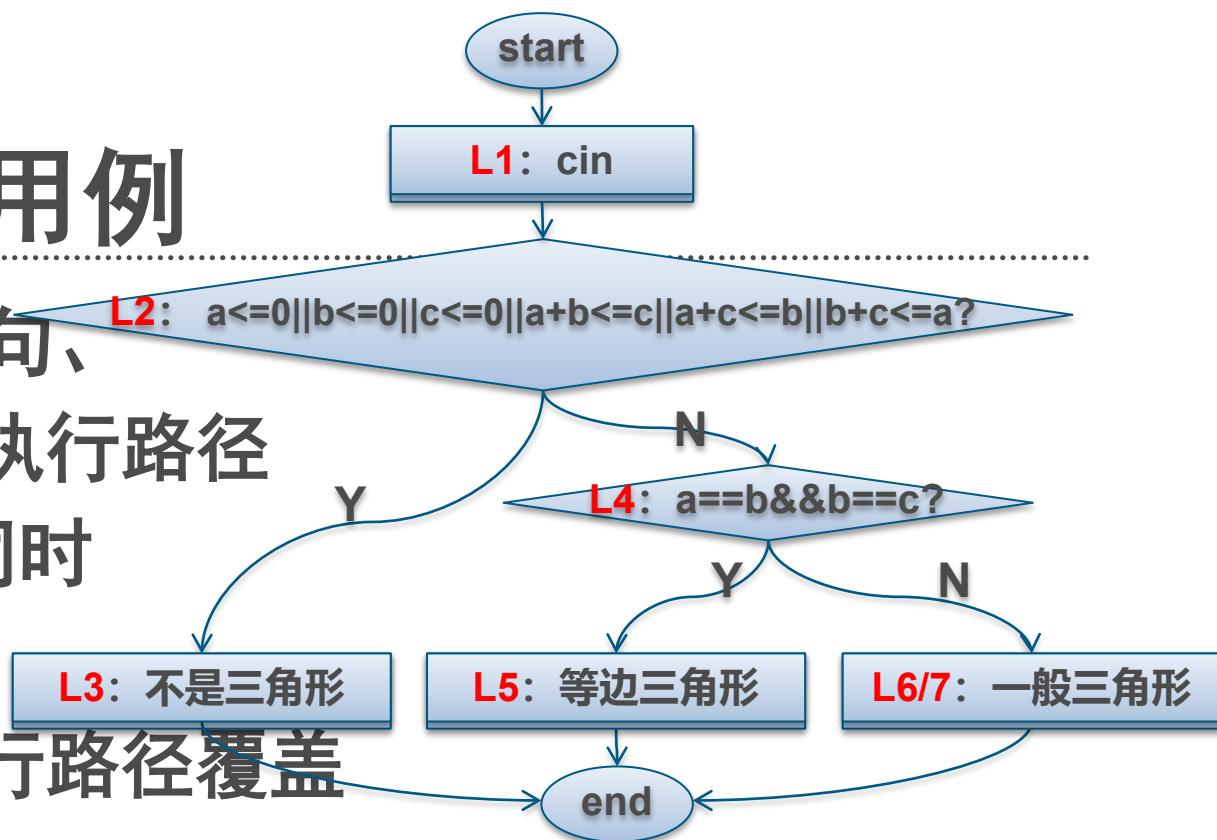
L6: else

L7: cout<<"一般三角形 ";



程序测试用例

- 本例共7条语句、2个分支、3条执行路径
- 以下用例可同时满足语句覆盖、分支覆盖和执行路径覆盖



用例	覆盖的语句	覆盖的分支	覆盖的路径	覆盖的从句 (C1~C8)
(1,2,9)	1/2/3	L2(True)	1,2,3	FFFTFFFF
(2,2,2)	1/2/4/5	L2(Flase),L4(True)	1,2,4,5	FFFFFFTT
(3,4,5)	1/2/4/6/7	L2(Flase),L4(Flase)	1,2,4,6,7	FFFFFFFF



测试过程

用例	预期结果	实际结果	是否一致	问题分析
(1,2,9)	不是三角形	【待填】	【待填】	【待填】
(2,2,2)	等边三角形	【待填】	【待填】	【待填】
(3,4,5)	一般三角形	【待填】	【待填】	【待填】

如果对应某条输入的预期结果与实际结果不一致，则说明程序有错，需要进一步分析。

所谓程序测试，就是设计上述表格→执行程序→填充结果→分析的过程



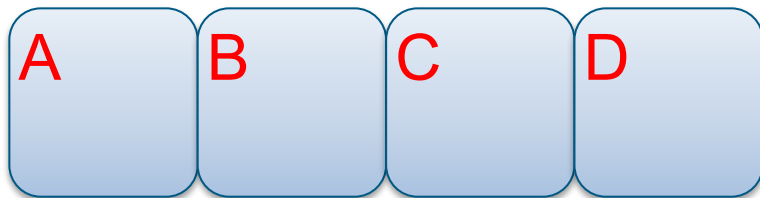
主要内容

- 空间复杂性
- 时间复杂性
 - 指标（计数对象）
 - 渐进符号（ O 、 Ω 、 Θ 、 o ）
- 性能测量

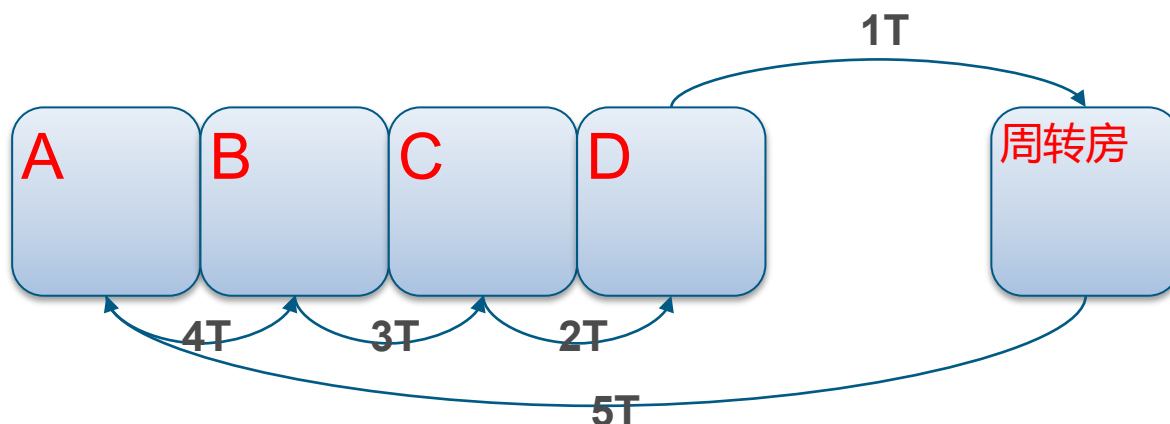


提出问题： 如何评价算法？

- 假设有4间寝室（A/B/C/D），如果要将A寝室的物品搬到B寝室，B寝室的物品搬到C寝室，C寝室的物品搬到D寝室，D寝室的物品搬到A寝室，且规定只有当某间寝室为空时才能往进搬东西，物品从一间寝室搬到另一间寝室的时间为T。请描述完成上述任务的算法，并评价其算法优劣。



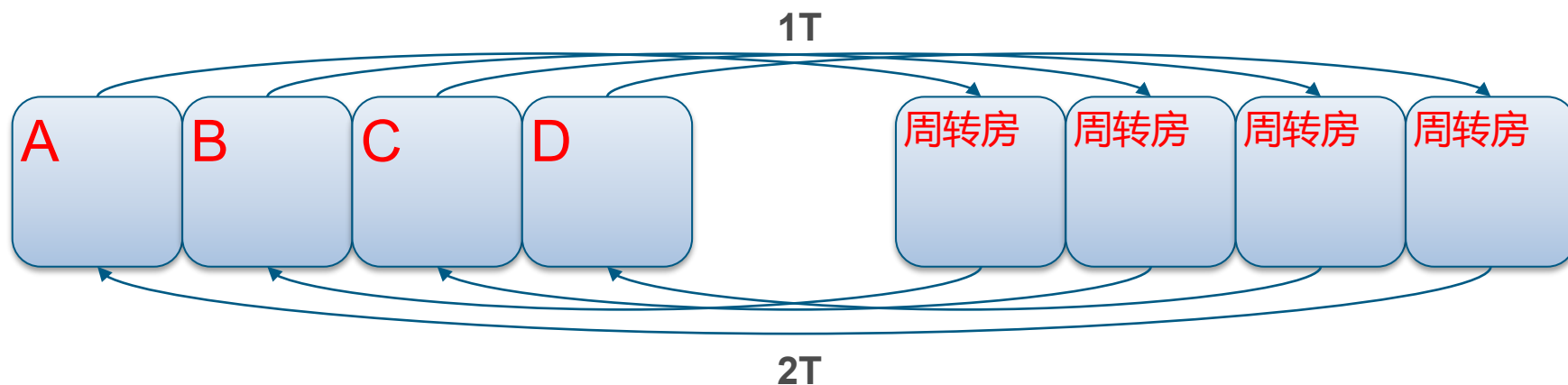
方案一：占用尽量少的额外空间



- 占用额外空间1
- 消耗时间5T



方案二：在最短时间内完成



- 占用额外空间4
- 消耗时间 $2T$



结论

- 评价程序性能
 - 分析或测量程序的时空开销
- 选择算法
 - 针对某一具体问题，在其不同解决方案之间进行空间和时间的权衡



影响程序性能评价的因素

- 计算机
 - 编译器及其选项
 - 问题规模（问题实例长度）
 - 具体输入
 - 代码本身（数据结构及算法）
- 暂不考虑
- 重点考虑



空间复杂性

- 程序所需空间

- 指令空间

- 存储编译后的指令所需的空间
 - 与目标机器、编译器及选项有关
 - 不在本课程研究范围之内

- 数据空间



- 存储常量、简单变量、复合变量及动态分配的空间

- 环境栈空间



- 发生函数调用时所需的空间

数据空间：VC++中变量所占空间

类型	占用字节数
char, unsigned char	1
short	2
int, unsigned int	4
long, unsigned long	4
float	4
double, long double	8
pointer	4

完成书中
练习时可
以参照图
2-2也可以
参照左表，
注明即可。

- `double a[100];` → 800字节
- `int maze[r][c];` → $4*r*c$ 字节



程序空间分析

- 程序空间分为固定部分和可变部分
- 其中可变部分是指随问题规模变化的空间
 - 复合变量所需的空间
 - 动态分配的空间
 - 递归栈所需的空间

$$S(P) = c + S_p(\text{实例特征})$$



空间复杂性分析例1：累加

```
template<class T>
```

```
T Sum(T a[], int n)
```

```
{
```

```
    T tsum=0;
```

```
    for (int i=0; i<n; i++)
```

```
        tsum+=a[i];
```

```
    return tsum;
```

```
}
```

消耗空间与n无关

$$S_{Sum}(n)=0$$

```
template<class T>
```

```
T Rsum(T a[], int n)
```

```
{
```

```
    if (n>0)
```

```
        return Rsum(a, n-1)+a[n-1];
```

```
    return 0;
```

```
}
```

消耗空间(递归栈空间)与n有关

$$S_{Rsum}(n)=(4+4+4) * (n+1)$$

$$S_{Sum}(n) < S_{Rsum}(n)$$



空间复杂性分析例2：阶乘

```
int Lfac(int n)
{
    int fac=1;
    for(int i=1;n>1&&i<=n;i++)
        fac*=i;
    return fac;
}
```

消耗空间与n无关

$$S_{Lfac}(n)=0$$

```
int Fac(int n)
{
    if(n<=1)
        return 1;
    else
        return n*Fac(n-1);
}
```

消耗空间(递归栈空间)与n有关

$$S_{Fac}(n)=(4+4) * \max\{n, 1\}$$

$$S_{Lfac}(n) < S_{Fac}(n)$$



小结

- 一个程序的空间开销包括
 - 生成代码指令后所占的空间
 - 简单变量、常量所占的空间
 - 复合变量所占的空间
 - 动态分配的空间
 - 递归栈空间

重点考查
可变部分



主要内容

- 空间复杂性
- 时间复杂性
 - 指标（计数对象）
 - 渐进符号（ O 、 Ω 、 Θ 、 o ）
- 性能测量



时空分析对比

- 空间复杂性

$$S(P) = c + s_p(\text{实例特征})$$

- 时间复杂性

$$T(P) = c + t_p(\text{实例特征})$$

不变部分 可变部分



应该摒弃的指标

- 代码的实际执行时间
- 运行过程中循环的次数
- 代码行数（LOC）

这些指标无法反映算法本质！



应该采用的指标

- 操作计数（基本操作数）
 - 程序运行中起主要作用且花费最多时间的操作
 - 排序问题中的比较操作、交换操作
 - 矩阵乘法中的数乘操作
- 执行步数
 - 程序运行中一个语法意义上的片段
 - 可能比操作计数更加准确



操作计数分析示例：多项式求值

$$P(x) = \sum_{i=0}^n c_i x^i = c_0 x^0 + c_1 x^1 + \dots + c_n x^n$$

鲁莽算法

```
template<class T>
T PolyEval(T coeff[], int n, const T& x)
{
    T y=1, value=coeff[0];
    for(int i=1; i<=n; i++)
    {
        y*=x;
        value+=y*coeff[i];
    }
    return value;
}
```

认定加法和乘法是关键操作，
该算法的关键操作有多少次呢？
对于问题规模n来说

累计将循环n次

} 每次循环做2个乘法和1个加法



多项式求值优化

$$P(x) = \sum_{i=0}^n c_i x^i = c_0 x^0 + c_1 x^1 + \dots + c_n x^n$$
$$= (\dots(((c_n * x + c_{n-1}) * x + c_{n-2}) * x + c_{n-3}) * x + \dots) * x + c_0$$

Horner算法

```
template<class T>
```

```
T Horner(T coeff[], int n, const T& x)
```

```
{
```

```
    T value=coeff[n];
```

```
    for(int i=1;i<=n;i++)
```

```
    {
```

```
        value = value * x + coeff[n-i];
```

```
    }
```

```
    return value;
```

```
}
```

对于问题规模n来说

累计将循环n次

每次循环做1个乘法和1个加法

结论: Horner算法更快!
因为其关键操作数更少!



H1. 四种排序算法及其操作分析

- 计数排序
- 选择排序
- 冒泡排序
- 插入排序
- **讲解**：思想→示例→代码→分析
- **共识**：所谓排序是指将一组无序元素按照从小到大的次序重新排列的过程，其关键操作是**元素的比较**



计数排序：思想

- 先求得元素在序列中的名次，具体过程是
 - 对于每一个元素，将它与其左边的所有元素比较
 - 谁大，谁的名次就加1
 - 直至结束
- 然后将元素调整到与其名次对应的位置上



计数排序：示例

a	26	33	35	29	19	12	22
---	----	----	----	----	----	----	----

r	3	5	6	4	1	0	2
---	---	---	---	---	---	---	---

u	12	19	22	26	29	33	35
---	----	----	----	----	----	----	----



计数排序：代码和分析

```
template<class T>
void Rank(T a[], int n,int r[]){
    for(int i=1; i<n; i++)
        r[i]=0;
    for (int i = 1; i < n; i++)
        for(int j=0; j<i; j++)
            if (a[j] <= a[i])
                r[i]++;
            else r[j]++;
}
```

循环 $n(n-1)/2$ 次

```
template<class T>
void Rearrange(T a[], int n, int r[]){
    T *u=new T[n+1];
    for(int i=0; i<n; i++)
        u[r[i]]=a[i];
    for(int i=0; i<n; i++)
        a[i]=u[i];
    delete []u;
}
```

结论：比较次数 $(n-1)n/2$
移动次数 $2n$ ，占用了额外空间！



计数排序改进思想

a

12	19	22	26	29	33	35
----	----	----	----	----	----	----

r

0	1	2	3	4	5	6
---	---	---	---	---	---	---

○ $r[i] == i$?

- Y: 很好, $a[i]$ 就应该是第 i 个, 它已经在正确位置上了, 什么也不用做, 继续考虑 $a[i+1]$
- N: $a[i]$ 的位置不对, 怎么办?
 - 谁应该在第 i 位我们不知道
 - 但, 我们知道 $a[i]$ 应该在哪—— $r[i]$, 将它与 $a[r[i]]$ 交换, $a[i]$ 即到达正确位置, 新交换来的应该在第 i 位吗? 继续



改进计数排序

```
template<class T>
```

```
void Rearrange(T a[], int n, int r[]){
```

```
    T *u=new T[n+1];
```

```
    for(int i=0; i<n; i++)
```

```
        u[r[i]]=a[i];
```

```
    for(int i=0; i<n; i++)
```

```
        a[i]=u[i];
```

```
    delete []u;
```

```
}
```

```
template<class T>
```

```
void Rearrange(T a[], int n, int r[]){
```

```
    for(int i=0; i<n; i++)
```

```
        while(r[i]!=i){
```

```
            int t=r[i];
```

```
            Swap(a[i],a[t]);
```

```
            Swap(r[i],r[t]);
```


```
        }
```

*结论：移动次数可能增加
占用空间减少！*



选择排序：思想

- 首先找出最小（大）元素，把它移动到 $a[0]$ ($a[n-1]$)；
- 然后在余下的 $n-1$ 个元素中寻找最小（大）的元素并把它移动到 $a[1]$ ($a[n-2]$)；
- 如此进行下去，直至全部排完。

$$A_0 \leq A_1 \leq \cdots A_{i-1} \mid A_i, \cdots A_{\min}, \cdots, A_{n-1}$$




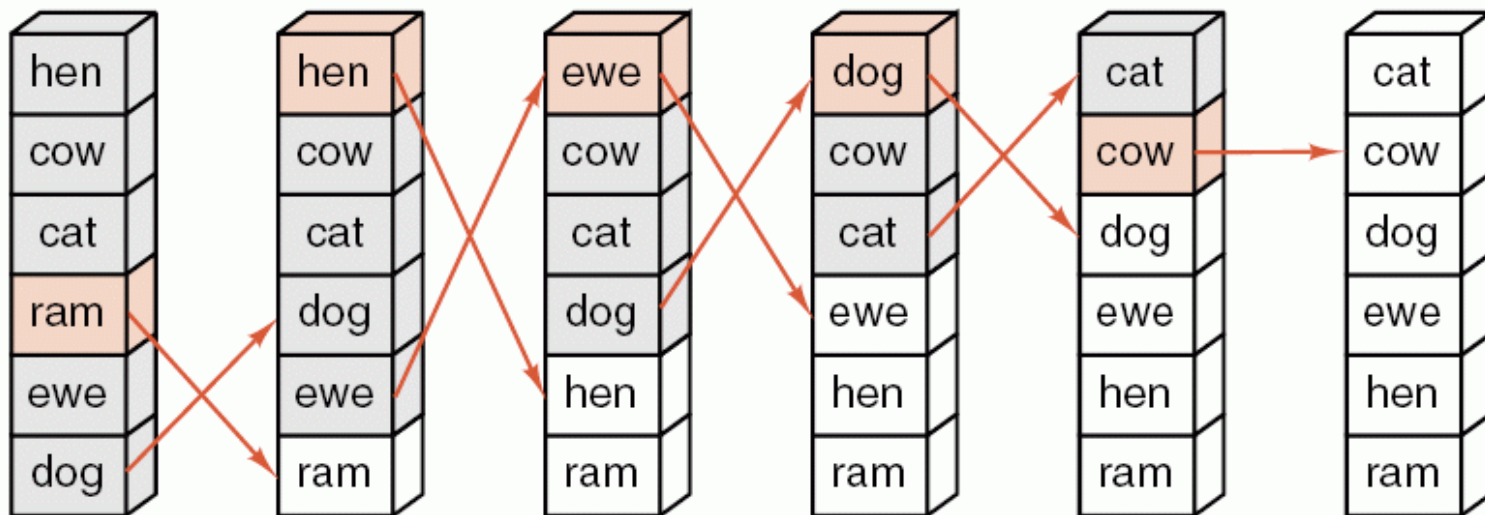
选择排序：示例

	89	45	68	90	29	34	17
17		45	68	90	29	34	89
17	29		68	90	45	34	89
17	29	34		90	45	68	89
17	29	34	45		90	68	89
17	29	34	45	68		90	89
17	29	34	45	68	89		90



选择排序： 示例2

Initial order



Sorted

Colored box denotes largest unsorted key.
Gray boxes denote other unsorted keys.

选择排序：代码和分析

```
template<class T>
```

```
int Max(T a[], int n)
```

```
{
```

```
    int pos = 0;
```

```
    for (int i = 1; i < n; i++)
```

```
        if (a[pos] < a[i])
```

```
            pos = i;
```

```
    return pos;
```

```
}
```

```
template<class T>
```

```
int SelectSort(T a[], int n)
```

```
{
```

```
    for (int size=n; size>1; size--)
```

```
    {
```

```
        int j=Max(a, size);
```

```
        Swap(a[j], a[size-1]);
```

```
    }
```

```
}
```

循环n-1次

结论：比较次数 $(n-1)n/2$
移动次数 $3(n-1)$



选择排序：改进思想

一般情况

| 89 45 68 90 29 34 17

次优情况

| 90 29 34 45 68 89 17

最优情况

| 17 29 34 45 68 89 90



改进选择排序

```
template<class T>
void SelectionSort(T a[], int n)
{
    bool sorted = false;
    for (int size = n; !sorted && (size > 1); size--) {
        int pos = 0;
        sorted = true;

        for (int i = 1; i < size; i++)
            if (a[pos] <= a[i]) pos = i;
        else sorted = false;
        Swap(a[pos], a[size - 1]);
    }
}
```

最好情况：只执行一次， $n-1$ 次比较

最坏情况与原来版本一样

数组有序，此语句不会被执行——
永远执行true分支——
 $a[0] \leq a[1] \leq \dots \leq a[n-1]$



冒泡排序：思想

- 首先**冒出**最小（大）元素，具体过程是：
 - 对相邻元素进行比较，如果左边的元素大于右边的元素，则交换
- 然后在余下的 $n-1$ 个元素中**冒出**最小（大）的元素；
- 如此进行下去，直至全部排完。

$$A_0 \leq A_1 \leq \cdots A_{i-1} \mid A_i, \cdots A_{j-1} \overset{?}{\longleftrightarrow} A_j \cdots, A_{n-1}$$



冒泡排序： 示例

89 45 68 90 29 34 17

89 45 68 90 29 17 34

89 45 68 90 17 29 34

89 45 68 17 90 29 34

89 45 17 68 90 29 34

89 17 45 68 90 29 34

17 | 89 45 68 90 29 34

17 | 89 45 68 90 29 34

17 | 89 45 68 29 90 34

17 | 89 45 29 68 90 34

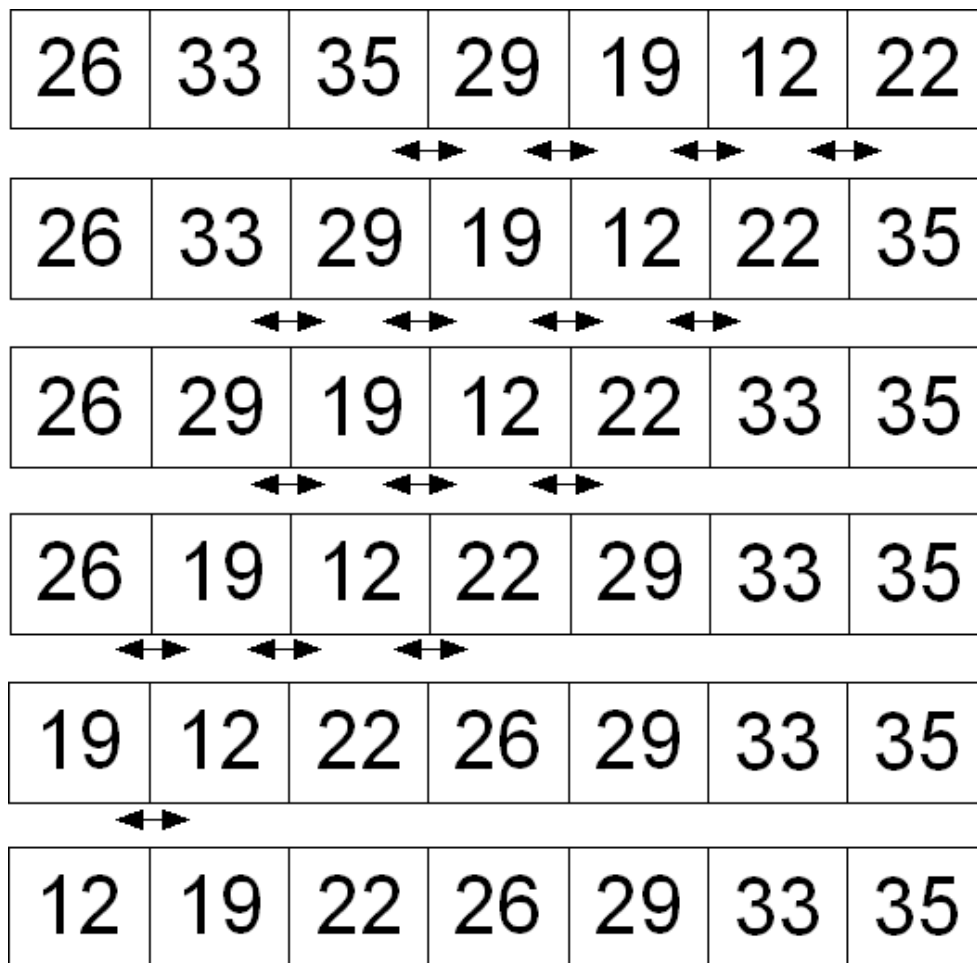
17 | 89 29 45 68 90 34

17 29 | 89 45 68 90 34

.....



冒泡排序：示例2



冒泡排序：代码和分析

```
template<class T>
```

```
void Bubble(T a[], int n)
```

```
{
```

```
    for (int i=0; i < n-1; i++)
```

循环n-1次 if (a[i+1] < a[i])

```
        Swap(a[i], a[i+1]);
```

```
}
```

```
template<class T>
```

```
void BubbleSort(T a[], int n)
```

```
{
```

```
    for (int size=n; size>1; size--)
```

```
    {
```

```
        Bubble(a, size);
```

```
    }
```

```
}
```

结论：比较次数 $(n-1)n/2$
移动次数不确定，但应该比选择排序多



改进冒泡排序

```
template<class T>
bool Bubble(T a[], int n)
{
    bool swapped = false;
    for (int i = 0; i < n - 1; i++)
        if (a[i] > a[i+1]) {
            Swap(a[i], a[i + 1]);
            swapped = true;
        }
    return swapped;
}
```

```
template<class T>
void BubbleSort(T a[], int n)
{
    for (int i = n; i > 1 &&
        Bubble(a, i); i--);
}
```


最好情况Bubble只执行一次， $n-1$ 次比较；最坏情况与原来版本一样

进行了相邻数据交换
如果这趟扫描没有交换呢？
所有 $a[i] \leq a[i+1]$ ——有序



插入排序：思想

- 从无序部分取一元素插入到有序部分的正确位置上，类似于玩家整理手中的扑克牌
- 把 $L[i+1]$ 插入到 $L[1 \cdots i]$ 中的正确位置， $i++$
- 直至全部插完停止


$$A_0 \leq \cdots A_j < A_{j+1} \leq \cdots A_{i-1} \mid A_i \cdots A_{n-1}$$



插入排序： 示例

89 | 45 68 90 29 34 17

45 89 | 68 90 29 34 17

45 68 89 | 90 29 34 17

45 68 89 90 | 29 34 17

45 68 89 ~~90~~ 90 34 17

45 68 ~~89~~ 89 90 34 17

45 ~~68~~ 68 89 90 34 17

~~45~~ 45 68 89 90 34 17

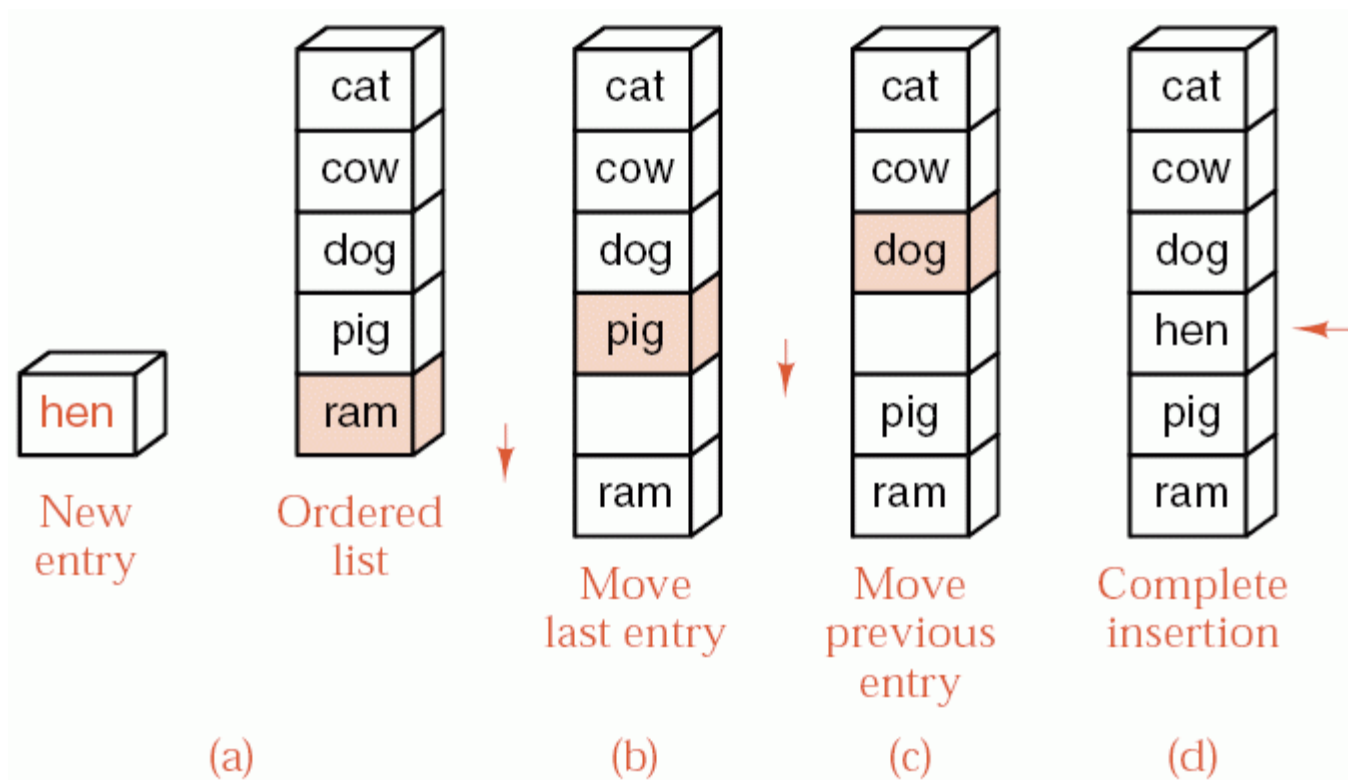
29 45 68 89 90 | 34 17

29 34 45 68 89 90 | 17

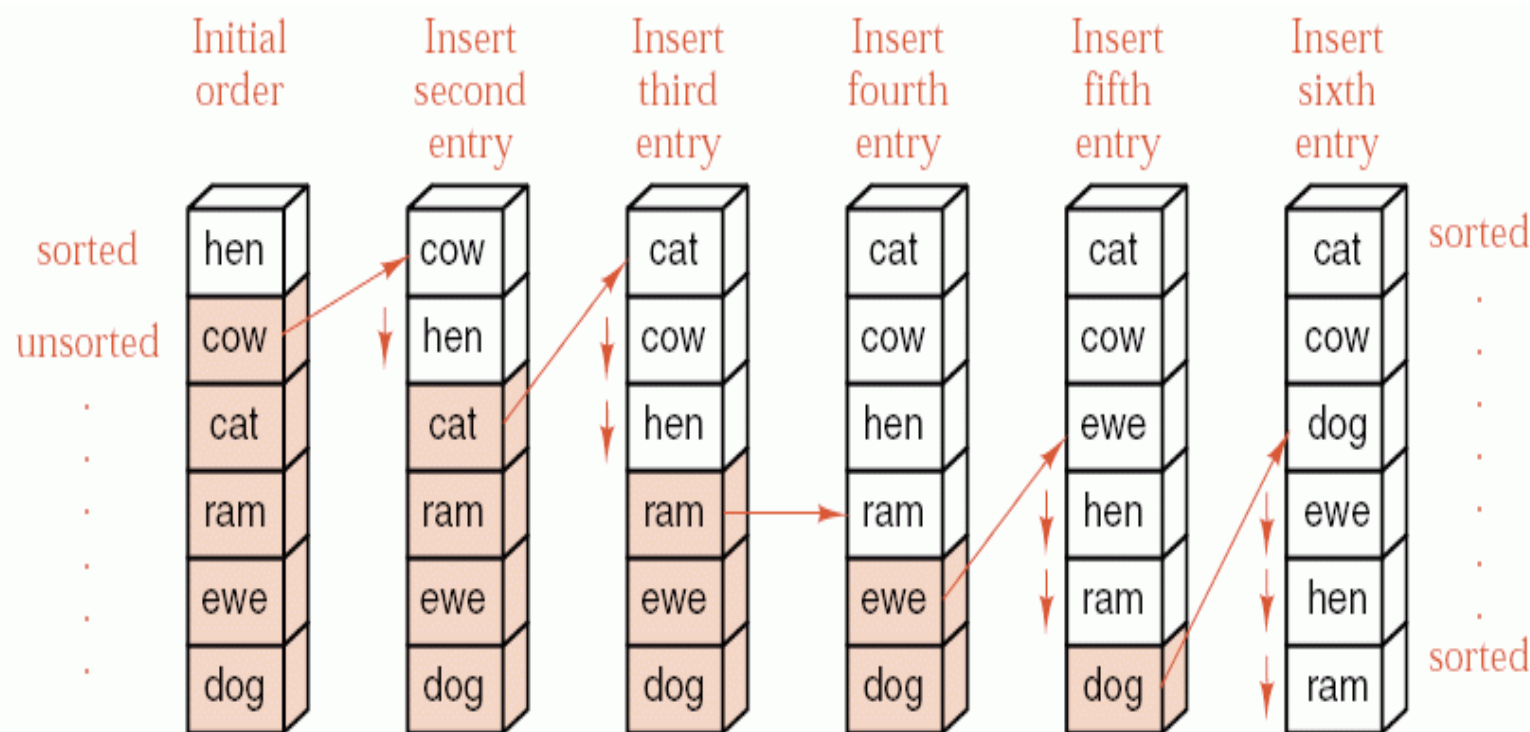
17 29 34 45 68 89 90



插入排序示例2



插入排序示例2



插入排序：代码和分析

```
template<class T>
void Insert(T a[], int n, const T& x)
{
    int i;
    for (i = n-1; i >= 0 && x < a[i]; i--)
        a[i+1] = a[i];
    a[i+1] = x;
}

template<class T>
void InsertionSort(T a[], int n)
{
    for (int i = 1; i < n; i++) {
        T t = a[i];
        Insert(a, i, t);
    }
}
```

插入数组末尾，1次比较，最好情况

插入第一个元素之前或之后，n次比较，最坏情况

平均比较次数

$(1+2+\dots+n+n)/(n+1) = n/2 + n/(n+1)$

1个元素：自然有序

Insert(a, 1, t): 2个有序

Insert(a, 2, t): 3个有序

...

Insert(a, n-1, t): 数组完全排序

结论：比较次数约为 $n^2/4$
优于选择排序



补充：稳定（stable）排序

- 序列 a_1, a_2, \dots, a_n 进行排序
- 对任意的 $i < j$, $a_i = a_j$,
若排序后 a_i, a_j 的位置 i', j' 一定满足
 $i' < j'$, 则称该排序算法为稳定的
- 否则, 为不稳定的

..... 26 26 26

..... 26 26 26

稳定的



..... 26 26 26

不稳定的



H1 小结

- 四种排序算法思想
 - 计数排序
 - 选择排序
 - 冒泡排序
 - 插入排序
- 以比较作为基本操作，如何评价算法优劣？
 - 与问题规模有关
 - 与具体输入有关（最优情况、最差情况）



H1小结（续）

```
for (int i=0; i<n; i++)
```

$$\sum_{i=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

```
for (int i=0; i<n; i++)
```

```
    for (int j=0; j<n; j++)
```

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = n^2$$

```
for (int i=0; i<n; i++)
```

```
    for (int j=0; j<i; j++)
```

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = n(n-1-1+1)/2 = n(n-1)/2$$



主要内容

- 空间复杂性
- 时间复杂性
 - 指标（计数对象）
 - 渐进符号（ O 、 Ω 、 Θ 、 o ）
- 性能测量



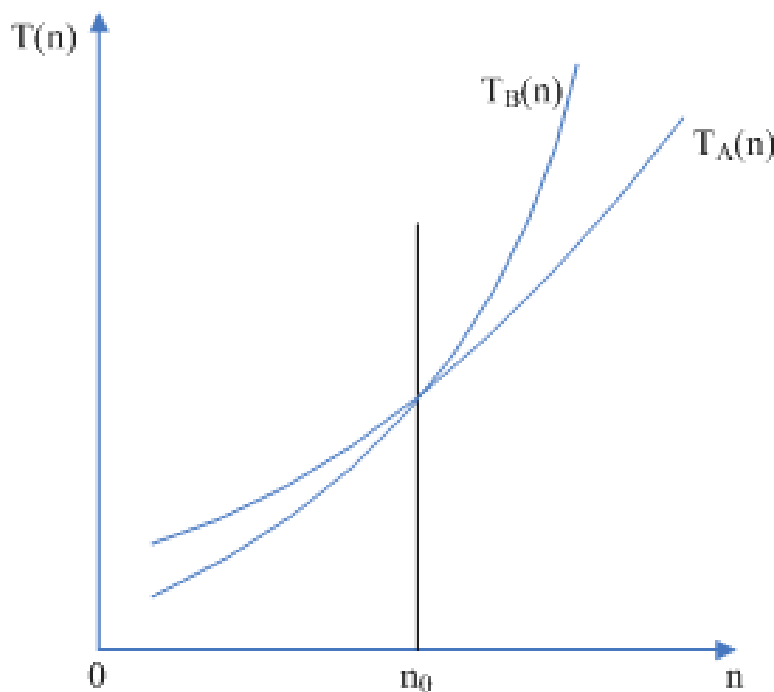
问题提出

- 操作计数和执行步数的作用？
 - 比较两个功能相同的程序的时间复杂性【横向】
 - 预测随实例规模的变化，程序运行时间的变化【纵向】
- 执行步试图比“关键操作”更精确
- “更精确”是没有必要的
 - 两个程序时间复杂性： $c_1n^2+c_2n$ 和 c_3n
 - 总存在一个点，当 n 超过此值，后者更快
 - “渐进”：大实例特征（趋向无穷）下，程序时间复杂性函数的“变化”情况



复杂度的渐进性质

- 如果解决问题P的程序A和程序B，其时间复杂度分别是 $T_A(n)$ 和 $T_B(n)$ ，则判断A、B性能优劣的标准是查看在 n 足够大时 $T_A(n)$ 和 $T_B(n)$ 的大小关系



H2. 大写 O 符号

- 函数上界
- 定义：
 $f(n) = O(g(n))$ ，当且仅当存在正常数 c 和 n_0 ，使得对所有 $n \geq n_0$ ，有 $f(n) \leq cg(n)$
- f 至多是 g 的 c 倍，对足够大的 n ， g 是 f 的上界（不考虑常数因子 c ）
- g 取简单函数——容易研究 f 的上界



常用做g的简单函数

函数	名称
1	常数
$\log n$	对数
n	线性
$n \log n$	n 个 $\log n$
n^2	平方
n^3	立方
2^n	指数
$n!$	阶乘

快、简单

慢、复杂

对数没有给出对数基，因为

$\log_a n = \log_b n / \log_b a$ ，仅常数不同，相差 $\log_b a$ 倍



大O符号例子

- 线性函数

$f(n) = O(g(n))$, 当且仅当
存在正常数 c 和 n_0 , 使得对
所有 $n \geq n_0$, 有 $f(n) \leq cg(n)$

- $f(n) = 3n + 2$

- $n \geq 2$ 时, $3n + 2 \leq 3n + n = 4n$, $f(n) = O(n)$

- $n > 0$ 时, $3n + 2 \leq 10n$; $n \geq 1$, $3n + 2 \leq 3n + 2n = 5n$ ——
结论不变, 可见 c 和 n_0 并不重要

- $f(n) = 3n + 3$: $n \geq 3$ 时, $3n + 3 \leq 3n + n = 4n$

- $f(n) = 100n + 6$: $n \geq 6$ 时,
 $100n + 6 \leq 100n + n = 101n$



大O符号例子（续）

$f(n) = O(g(n))$, 当且仅当
存在正常数 c 和 n_0 , 使得对
所有 $n \geq n_0$, 有 $f(n) \leq cg(n)$

- 平方函数

- $f(n) = 10n^2 + 4n + 2$

$n \geq 2$ 时, $f(n) \leq 10n^2 + 5n$

当 $n \geq 5$ 时, $5n \leq n^2$, 因此 $n \geq n_0 = 5$ 时,
 $f(n) \leq 10n^2 + n^2 = 11n^2$, 所以 $f(n) = O(n^2)$

- $f(n) = 1000n^2 + 100n - 6$

对于所有 n 有 $f(n) \leq 1000n^2 + 100n$,

对于 $n \geq 100$, 有 $100n \leq n^2$,

因此对于 $n \geq n_0 = 100$, 有 $f(n) < 1001n^2$,
所以 $f(n) = O(n^2)$



大O符号例子（续）

$f(n) = O(g(n))$, 当且仅当
存在正常数 c 和 n_0 , 使得对
所有 $n \geq n_0$, 有 $f(n) \leq cg(n)$

- 指数函数

- $f(n) = 6 \cdot 2^n + n^2$

- $n \geq 4$ 时, $n^2 \leq 2^n$,

- 所以对于 $n \geq 4$, $f(n) \leq 6 \cdot 2^n + 2^n = 7 \cdot 2^n$,

- 因此 $6 \cdot 2^n + n^2 = O(2^n)$

- 常数函数

- $f(n) = 9$ 或 $f(n) = 2033$, $f(n) = O(1)$, 证明很简单:

- $f(n) = 9 \leq 9 \cdot 1$, $c = 9$ 、 $n_0 = 0$

- $f(n) = 2033 \leq 2033 \cdot 1$, $c = 2033$ 、 $n_0 = 0$



大O符号例子（续）

- 松散界限

- 当 $n \geq 2$ 时, $3n+3 \leq 3n^2 \rightarrow 3n+3 = O(n^2)$, 不是最小上界
- 当 $n \geq 2$ 时, $10n^2+4n+2 \leq 10n^4 \rightarrow 10n^2+4n+2 = O(n^4)$
- $6n2^n+20 = O(n^22^n)$, 更小上界 $n2^n$

- 逐步用更低阶的函数替换高阶函数, 直到找到最小上界



错误界限

- $3n+2 \neq 0(1)$ ，反证法：
 - 假定存在 c 及 n_0 ，使得对所有的 $n \geq n_0$ ，有 $3n+2 \leq c \rightarrow n \leq (c-2)/3$ ，因此取 $n > \max\{n_0, (c-2)/3\}$ ，矛盾！
- $10n^2+4n+2 \neq 0(n)$
 - 假定存在 c 和 n_0 ，使得对于所有的 $n \geq n_0$ ，有 $10n^2+4n+2 \leq cn \rightarrow 10n+4+2/n \leq c$ 取 $n > \max\{n_0, (c-4)/10\}$ ，矛盾



错误界限（续）

- $f(n) = 3n^2 2^n + 4n 2^n + 8n^2 \neq O(2^n)$
 - 假定存在 $c > 0$ 和 n_0 ，使得对于所有的 $n \geq n_0$ ，有 $f(n) \leq c \cdot 2^n \rightarrow 3n^2 + 4n + 8n^2 / 2^n \leq c$ ，
左边随着 n 的增长而增大，右边是常数
取一个“足够大”的 n ，不等式不成立



多项式的阶

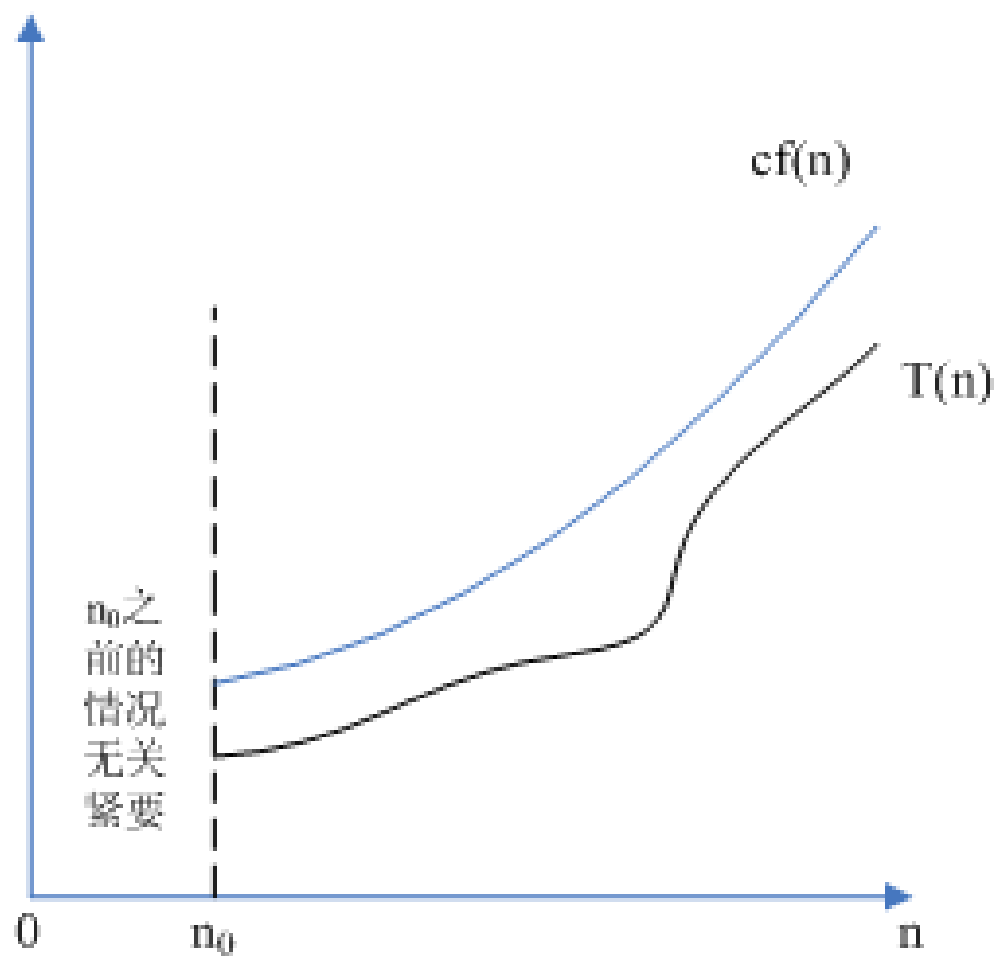
- 定理2-1：如果 $f(n) = a_m n^m + \dots + a_1 n + a_0$ 且 $a_m > 0$ ，则 $f(n) = O(n^m)$

证明： 对所有 $n \geq 1$

$$\begin{aligned} f(n) &\leq \sum_{i=0}^m |a_i| n^i \\ &\leq n^m \sum_{i=0}^m |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^m |a_i| \end{aligned}$$



大O原理图示



H2小结

- 关于大O符号有如下认识
 - 时间复杂度的“**级别**”比“具体量”更重要！
 - 确定时间消耗是什么级别，而非具体多少
 - 是问题规模的函数
 - 根据渐进性质，考虑问题足够大的情况
 - 本质上是最差情况，这一点符合工业界需求



Ω 符号

- 函数下界

- 定义：

$f(n) = \Omega(g(n))$ ，当且仅当存在正常数 c 和 n_0 ，使得对所有 $n \geq n_0$ ，有 $f(n) \geq c g(n)$

- f 至少是 g 的 c 倍，对足够大的 n ， g 是 f 的一个下界



Ω 符号例子

- 线性函数

- 对于所有的 n , 有 $f(n)=3n+2>3n \Rightarrow f(n)=\Omega(n)$
- $f(n)=3n+3>3n \Rightarrow f(n)=\Omega(n)$
- $f(n)=100n+6>100n$, 所以 $100n+6=\Omega(n)$

- 平方函数

- 对所有 $n \geq 0$, $f(n)=10n^2+4n+2>10n^2 \Rightarrow f(n)=\Omega(n^2)$
- $1000n^2+100n-6=\Omega(n^2)$

- 指数函数

- $6*2^n+n^2>6*2^n \Rightarrow 6*2^n+n^2=\Omega(2^n)$



Ω 符号例子（续）

- 非最大下界

- $3n+3 = \Omega(1)$, $10n^2+4n+2 = \Omega(n)$,
 $10n^2+4n+2 = \Omega(1)$, $6 \cdot 2^n + n^2 = \Omega(n^{100})$

- $6 \cdot 2^n + n^2 = \Omega(n^{50.2})$, $6 \cdot 2^n + n^2 = \Omega(n^2)$,
 $6 \cdot 2^n + n^2 = \Omega(n)$ 和 $6 \cdot 2^n + n^2 = \Omega(1)$

- $3n+2 \neq \Omega(n^2)$

- 假定 $3n+2 = \Omega(n^2) \rightarrow$ 存在正数 c 和 n_0 , 使得对所有 $n \geq n_0$, 有 $3n+2 \geq cn^2 \rightarrow cn^2 / (3n+2) \leq 1$, 左边随 n 的增大而变得无限大, 右边常数, 不等式变为不成立

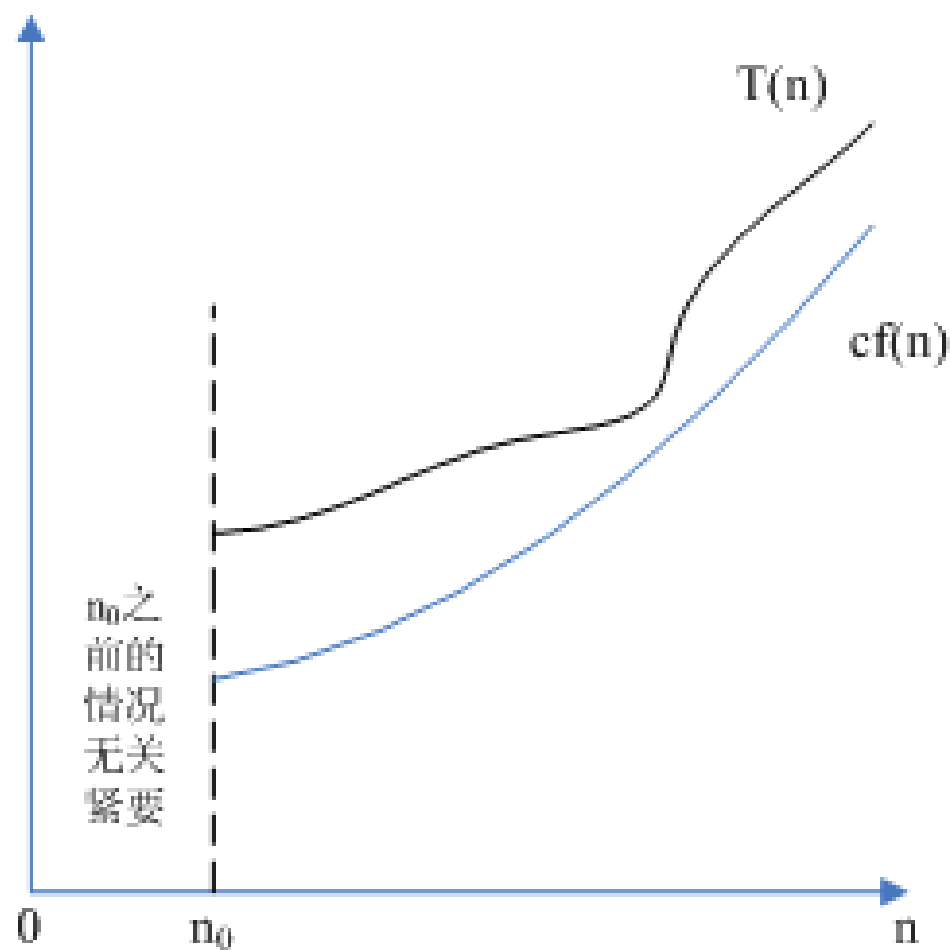


关于 Ω 的多项式定理

- 定理2-3: 如果 $f(n) = a_m n^m + \dots + a_1 n + a_0$ 且 $a_m > 0$, 则 $f(n) = \Omega(n^m)$
- 例: $3n+2 = \Omega(n)$, $10n^2+4n+2 = \Omega(n^2)$, $100n^4+3500n^2+82n+8 = \Omega(n^4)$



Ω 原理图示



⊕ 符号

- 同一个 g 既作为 f 的上界，又作为下界
- 定义：
 $f(n) = \Theta(g(n))$ ，当且仅当存在正常数 c_1 、 c_2 和 n_0 ，使得对所有 $n \geq n_0$ ，有
$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$
- f 介于 g 的 c_1 倍和 c_2 倍之间，对足够大的 n ， g 既是 f 的上界也是下界



⊕ 符号例子

- 从前例可知: $3n+2 = \Theta(n)$, $3n+3 = \Theta(n)$,
 $100n+6 = \Theta(n)$, $10n^2+4n+2 = \Theta(n^2)$,
 $1000n^2+100n-6 = \Theta(n^2)$, $6*2^n+n^2 = \Theta(2^n)$
- 对 $n \geq 16$, $\log_2 n < 10 * \log_2 n + 4 \leq 11 * \log_2 n$
 $\rightarrow 10 * \log_2 n + 4 = \Theta(\log n)$
- $3n+2 \neq \Theta(1)$, $3n+3 \neq \Theta(1)$, $100n+6 \neq \Theta(1)$
- $3n+3 \neq \Theta(n^2)$
- $10n^2+4n+2 \neq \Theta(n)$, $10n^2+4n+2 \neq \Theta(1)$
- $6*2^n+n^2 \neq \Theta(n^2)$, $6*2^n+n^2 \neq \Theta(n^{100})$,
 $6*2^n+n^2 \neq \Theta(1)$



关于 Θ 的多项式定理

- **定理2-5:** 如果 $f(n)=a_m n^m+\dots+a_1 n+a_0$ 且 $a_m>0$, 则 $f(n)=\Theta(n^m)$
- 例: $3n+2=\Theta(n)$, $10n^2+4n+2=\Theta(n^2)$,
 $100n^4+3500n^2+82n+8=\Theta(n^4)$



小写o符号

- 定义:

$f(n) = o(g(n))$, 当且仅当 $f(n) = O(g(n))$, 且 $f(n) \neq \Omega(g(n))$

- $3n+2 = O(n^2)$ 且 $3n+2 \neq \Omega(n^2) \Rightarrow 3n+2 = o(n^2)$ 但 $3n+2 \neq o(n)$

- $10n^2+4n+2 = o(n^3)$, 但 $10n^2+4n+2 \neq o(n^2)$



主要内容

- 空间复杂性
- 时间复杂性
 - 指标（计数对象）
 - 渐进符号（ O 、 Ω 、 Θ 、 o ）
- 性能测量



实际复杂性

- 利用渐进复杂性
 - 比较：两个解决相同问题的程序，其时间随问题规模变化而变化情况
- $P: \Theta(n)$, $Q: \Theta(n^2) \rightarrow n$ 足够大, P 快
- “足够大”——应考虑问题实际规模,
 $10^6 n$ 实际中几乎总是比 n^2 慢

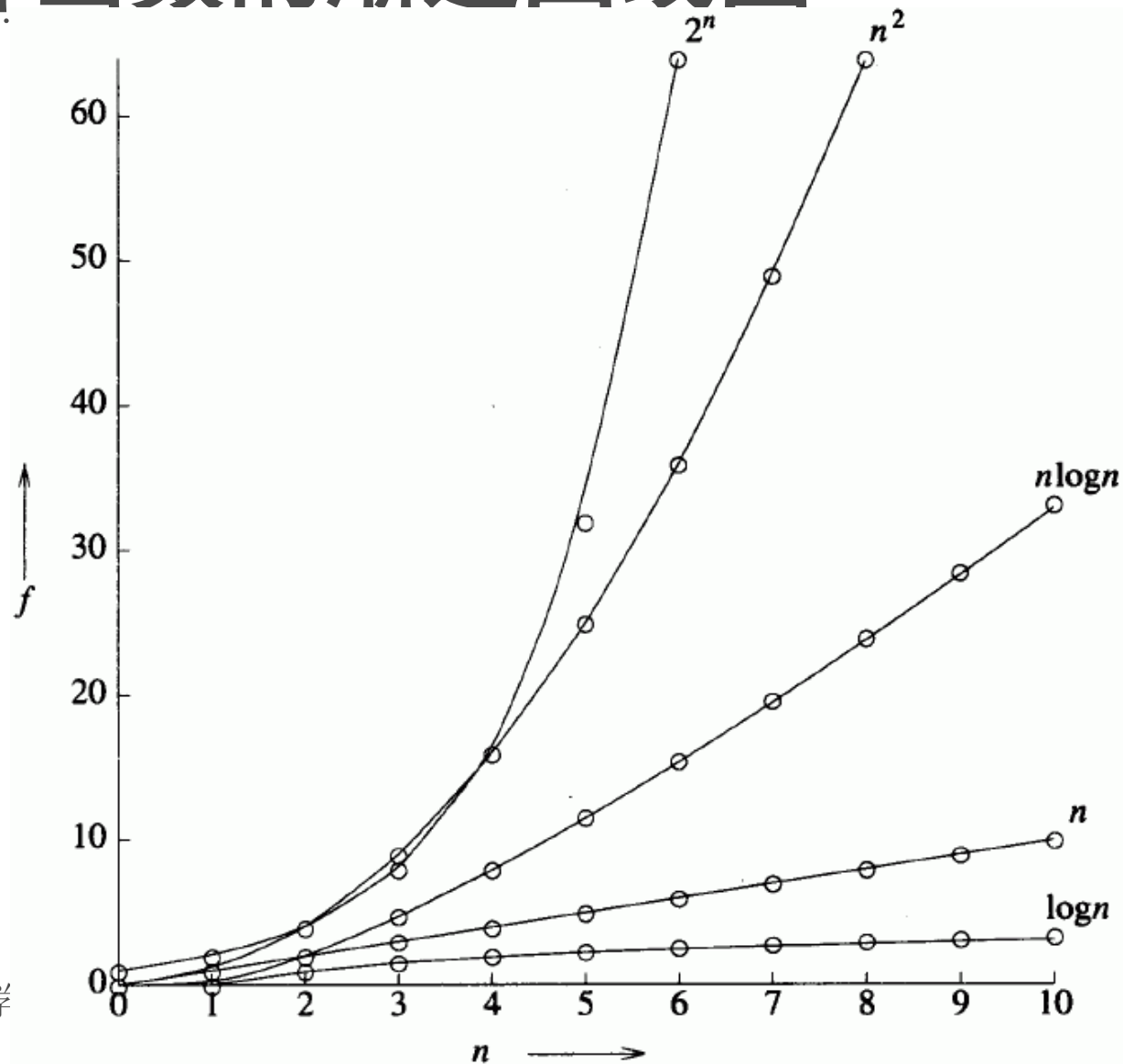


各种函数的渐进变化表

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4 096	65 536
5	32	160	1 024	32 768	4 294 967 296



各种函数的渐进曲线图



实际性能测量：Insert VS Bubble

- 机器环境
 - HP P6-1095
 - CPU: 3G * 2
 - RAM: 4G
- 实验1：逆序数列，1组
- 实验2：逆序数列，多组



实验1：一组逆序数列

```
void main(void)
{
    int a[10000], step = 500;
    clock_t start, finish;
    for (int n = 0; n <= 10000; n += step) {
        // get time for size n
        for (int i = 0; i < n; i++)
            a[i] = n - i; // initialize
        start = clock( );
        InsertionSort(a, n);
        finish = clock( );
        cout << n << ' ' << (finish - start) /
float(CLOCKS_PER_SEC) << endl;
        if (n == 1000) step = 1000;
    }
}
```



实验1：一组逆序数列

规模	Insert	Bubble
100	0	0
500	0	0
1000	0	0.016
2000	0.016	0.062
3000	0	0.110
4000	0.031	0.202
5000	0.031	0.312
6000	0.047	0.468
7000	0.062	0.640
8000	0.078	0.826
9000	0.094	1.046
10000	0.125	1.279



实验2：多组逆序数据

```
void main(void)
{
    int a[10000], n, i, step = 500;
    long counter;
    float seconds;
    clock_t start, finish;
    for (n = 0; n <= 10000; n += step) {
        // get time for size n
        start = clock( ); counter = 0;
        while (clock( ) - start < 100) {
            counter++;
            for (i = 0; i < n; i++)
                a[i] = n - i; // initialize
            InsertionSort(a, n);
        }
        finish = clock( );
        seconds = (finish - start) / float(CLOCKS_PER_SEC);
        cout << n << ' ' << counter << ' ' << seconds
            << ' ' << seconds / counter << endl;
        if (n == 1000) step = 1000;}
}
```



实验2：多组逆序数列

规模	Insert	Bubble
100	0.0000000654	0.0000000950
500	0.000326347	0.00320588
1000	0.00128235	0.013625
2000	0.00495455	0.052
3000	0.011	0.109
4000	0.0206667	0.219
5000	0.03125	0.327
6000	0.047	0.453
7000	0.062	0.624
8000	0.078	0.812
9000	0.1015	1.03
10000	0.125	1.295



本节课我们学习了：

- 评判程序性能的两个标准
 - 空间复杂性
 - 时间复杂性
 - 分析法【接下来会频繁用到】
 - 实验法
- 掌握了四种比较简单的排序算法



思考

- 能否通过实验，全面对比四种排序算法的优劣？实验结果与分析结果一致吗？
- 如果你在写完某个程序后发现运行特别慢，必须加以改进，该如何做？



本章结束

