算法设计与分析

第1章 算法概述（2）

学习要点

- 算法在计算机科学中的地位
- 算法的概念
- 算法分析
- 算法的计算复杂性概念
- 算法渐近复杂性的数学表述

算法分析

- 不是所有能计算的都有价值,不是所有有价值的都能被计算.

  阿尔伯特.爱因斯坦

- 当你对所讲的内容能够进行度量并能够用数字来表达的时候,证明你对这些内容是有所了解的；如果你不能用数字来表达,你的认识是不完整的,也是无法令人满意的.无论它是什么内容,他也许正处于知识的初级阶段，但在你的思想中,几乎从没把它上升到一个科学的高度.

  英国物理学家、发明家 Lord Kelvin

算法分析

- 算法”好”与”不好”？
  - 正确性分析
  - 时间效率分析
  - 空间效率分析
- 是否存在更好的算法？
  - 问题有没有下界
  - 算法是不是最优
- 途径
  - 理论/数学上的分析
  - 经验/计算机上的执行情况

算法属性的分析：正确性和效率

问题的复杂性：求解该问题的所有算法的复杂性的最小者

对弈软件：采用指数函数转换成四个近似的表的算法

示例 **JCS-聚类算法** JCS-团队生成 KBS-影响力

算法正确性分析

- 定义：一个算法是正确的，如果它对于每一个输入都最终停止,而且产生正确的输出
  - 不正确算法
    - ①不停止(在某个输入上)
    - ②对所有输入都停止，但对某个输入产生不正确的结果
  - 近似算法
    - ①对所有输入都停止
    - ②产生近似正确的解或产生不多的不正确解
  - 正确性证明
    - ①证明算法对所有输入都停止
    - ②证明对每个输入都产生正确结果

    数学归纳法
  - 调试程序=程序正确性证明?
    - “程序调试只能证明程序有错误，不能证明程序无错误”
  - 接受在一定概率下获得正确解

算法正确性分析（续）

- 数学归纳法进行证明的步骤：
  - 步骤1 （归纳奠基）证明当n取第一个值n0时命题成立；证明了第一步，就获得了递推的基础，但仅靠这一步还不能说明结论的普遍性.
  - 步骤2 （归纳递推）**假设n=k时命题成立**，证明当n=k+1时命题也成立；证明了第二步，就获得了递推的依据，但没有第一步就失去了递推的基础.只有把第一步和第二步结合在一起，才能获得普遍性的结论；
  - 步骤3 （下结论）命题对从n0开始的所有正整数都成立。

问题求解的启示：假设问题的一部分已经解决了，如何解决剩下的部分

算法复杂性分析

- 目的
  预测算法对不同输入所需**资源**量，为求解一个问题选择最佳算法、最佳设备
- 复杂性测度
  时间，空间， I/O 等, 是输入大小的函数
- 需要的数学基础
  离散数学，组合数学，概率论，代数等
- 需要的数学能力
  建立算法复杂性的数学模型
  数学模型化简

模型大小
参数个数
能量

大数据计算问题：
通信复杂性：分布式计算节点间的通信带宽
I/O复杂性：求解输入大小为n的问题的任一算法需要的I/O数据量
I/O数据量：内存与外存直接传输的数据量

算法复杂性分析

- 大数据研究的挑战
  - 数据规模导致难以应对的存储和计算量
  - 数据规模导致传统算法失效
    - 传统的多项式时间算法不适于求解大数据计算问题
  - 大数据复杂的数据关联性导致高复杂度的计算
- 大数据研究的三个基本途径
  - 继续寻找新算法降低计算复杂度
  - 降低大数据尺度，寻找数据尺度无关算法
  - 大数据并行化处理

学习要点

- 算法在计算机科学中的地位
- 算法的概念
- 算法分析
- 算法的计算复杂性概念
- 算法渐近复杂性的数学表述

算法的复杂性

- 算法的复杂性是**算法效率的度量**,是评价算法效率的重要依据
- 一个算法复杂性的高低体现在运行该算法所需的**计算机资源**的多少上
  - 计算机的资源,主要体现在**时间**和**空间**(存储器)资源上
  - 算法的复杂性分为时间复杂性和空间复杂性
    - 本课程主要对算法的时间复杂性进行分析
- 关于算法的复杂性,有两个问题需要搞清楚
  - ①用怎样的一个量来**表达**算法的复杂性
  - ②对一个具体的算法,怎样**计算**它的复杂性

算法的复杂性

- 算法复杂性 C= 算法所需要的计算机资源的量
  - 需要时间资源的量: 算法的时间复杂性T；
  - 需要空间资源的量: 算法的空间复杂性S。
- C=f(N, I, A)
  - N 要解问题的规模
  - I 算法的输入
  - A 算法本身

算法的复杂性

- 案例分析

```
def  compare_num(i,j):
    k=5
    if i>j:
        return i
    else:
        return k
```

常数时间赋值 c1

c2

c3

c4

总计：c1+c2+c3 or c4=常数时间

算法的复杂性

- 复杂性函数C=f(N,I,A)的简化
  - A隐含在复杂性函数名中　T(N, I)
  - T(N,I)是基本运算的使用次数e的加权和

T(N, I)
S(N, I)

$$\mathrm{T}(N,I)=\sum_{i=1}^{k} t_i e_i(N,I)$$

抽象计算机	k种元运算	O_1	O_2	…	O_k
	执行1次所需时间	t_1	t_2	…	t_k
算法A	用到元运算次数	e_1	e_2	…	e_k

算法复杂性分析的核心：计数

算法的复杂性

- 以一种**机器（或语言）无关**的方式进行分析
- 计算的RAM模型
  - 每一个简单操作(+,-,=,if,call)是一个基本步骤(step)
  - Loop和subroutine calls不是简单操作，它们依赖于数据的大小和子程序的内容
  - 每一次内存访问是一个基本步骤(step)
- 算法的运行时间就是这些所有步骤的**总数**

大局观念：分析独立于具体实现的属性

- 复杂度分析时除了求和，还会有什么问题？
  - 和输入I有关，和规模N有关

算法的复杂性-分析的类型

$$T(N,I)=\sum_{i=1}^{k} t_i e_i(N,I)$$

- 但不可能对规模为N的每一种合法输入都去统计e_i ，故只在**规模为N**的某些或某类有代表性的合法输入中统计相应的e_i.
  - 最坏情况$T_{max}(N)$
  - 最好情况$T_{min}(N)$
  - 平均情况$T_{avg}(N)$

算法的时间复杂性

有代表性的输入

- 最坏情况下的时间复杂性
  - $T_{max}(N) = \max\{ T(I) \mid size(I)=n \} = T(N,\mathbf{I^*})$
    - $= \max_{I \in D_N} \sum_{i=1}^{k} t_i e_i(N,I) = \sum_{i=1}^{k} t_i e_i(N, \boxed{\boldsymbol{I}^*})$

可操作性最好
最有实际价值

D_N问题规模为N的输入集合

- 最好情况下的时间复杂性
  - $T_{min}(N) = \min\{ T(I) \mid size(I)=n \} = T(N, \mathbf{I^{\sim}})$
    - $= \min_{I \in D_N} \sum_{i=1}^{k} t_i e_i(N,I) = \sum_{i=1}^{k} t_i e_i(N, \boxed{\boldsymbol{I}^{\sim}})$
- 平均情况下的时间复杂性
  - $T_{avg}(N) = \sum_{I \in D_N} P(I) T(N,I)$
    - $= \sum_{I \in D_N} \boldsymbol{P(I)} \sum_{i=1}^{k} t_i e_i(N, \boxed{I})$
- 其中I是问题的**规模为N**的实例，p(I)是实例I出现的概率。

问题的一个实例用输入可以确定

排序问题的最好情况是什么？
1个数的排序是不是最好情况？

算法分析

- 必须考虑的三个问题
  - 它是正确的吗？
  - 它复杂吗？
    - 将耗费多少时间
    - 其时间耗费关于n是一个什么样的函数？
  - 我们能改进它吗？

问题驱动的思维过程

算法分析: Fibonacci Number

- 问题：
- 一对兔子饲养到第二个月进入成年，第三个月生一对小兔，以后每个月生一对小兔(一雌一雄)，所生小兔全部都能存活并且也是第二个月成年，第三个月生一对小兔，以后每个月生一对小兔。问这样子下去到年底应该有**多少对**小兔?
- f_n = ?

(image of Leonardo Fibonacci from http://www.math.ethz.ch/fibonacci)

月份	大兔	出生的小兔	兔子总数
1月	1	0	F1=1
2月	1	0	F2=1
3月	1	1	F3=2
4月	2	1	F4=3
5月	3	2	F5=5
6月	5	3	F6=8

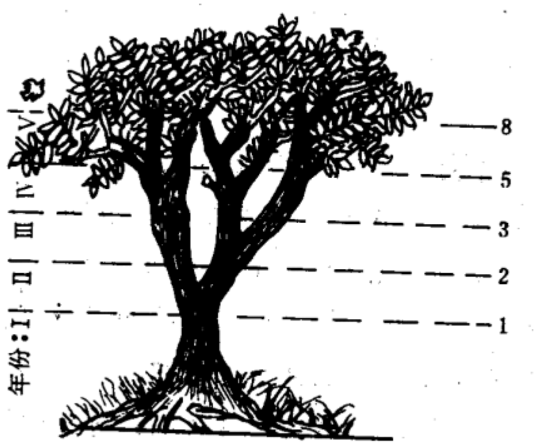
- ***分析：***
- $f_1 = 1$, $f_2 = 1$
- For all integers n > 2:
  $f_n = f_{n-1} + f_{n-2}$
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, **144**,...

算法分析: Fibonacci Number

- Fibonacci数列
  - 无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。它可以递归地定义为：

$$F(n)=\begin{cases} 1 & n=0 \\ 1 & n=1 \\ F(n-1)+F(n-2) & n>1 \end{cases}$$

鲁德维格定律



算法分析: Fibonacci

- 面试题目：求斐波那契数列的第n项
- 解法1: 直接用递归函数来实现。

Algorithm 1 F(n)
Input：n
Output：F(n)
if $n \leq 1$ **then**
    return (1)
else
    return (F(n−1) + F(n−2))

正确否？

复杂程度？

能改进吗？

算法分析: Fibonacci

Algorithm 1 F(n)

if $n \leq 1$ **then**

  return (1)

else

  return (F($n-1$) + F($n-2$))

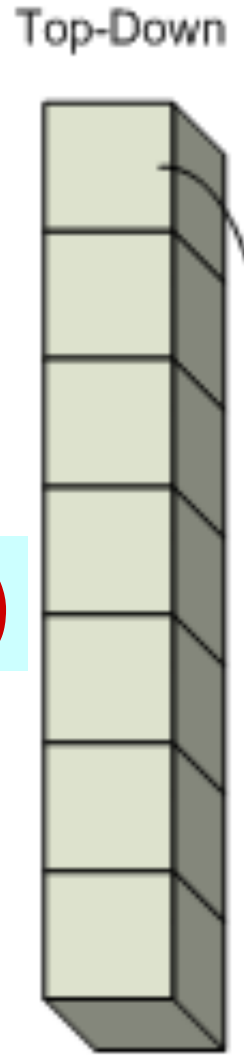
T(n)

O(1)

T(n-1)+T(n-2)+1

- 分析
  - 基本操作：检查n值和加法
  - 设T(n)是计算F(n)的算法时间复杂度
  - 有T(n)>T(n-1)+T(n-2)
  - F(n)/F(n-1)=(1+sqrt(5))/2=1.618
  - 有T(n) > Fn $\approx 1.6^n = 2^{0.694n}$
- 指数级时间（Exponential time）

$O(\phi^n)$



$T(200)>2^{138}$

算法分析: Fibonacci

- 面试题目：求斐波那契数列的第n项。
- 解法1: 直接用递归函数来实现。

Algorithm 1 F(n)

Input：n

Output：F(n)

if $n \le 1$ **then**

  return (1)

else

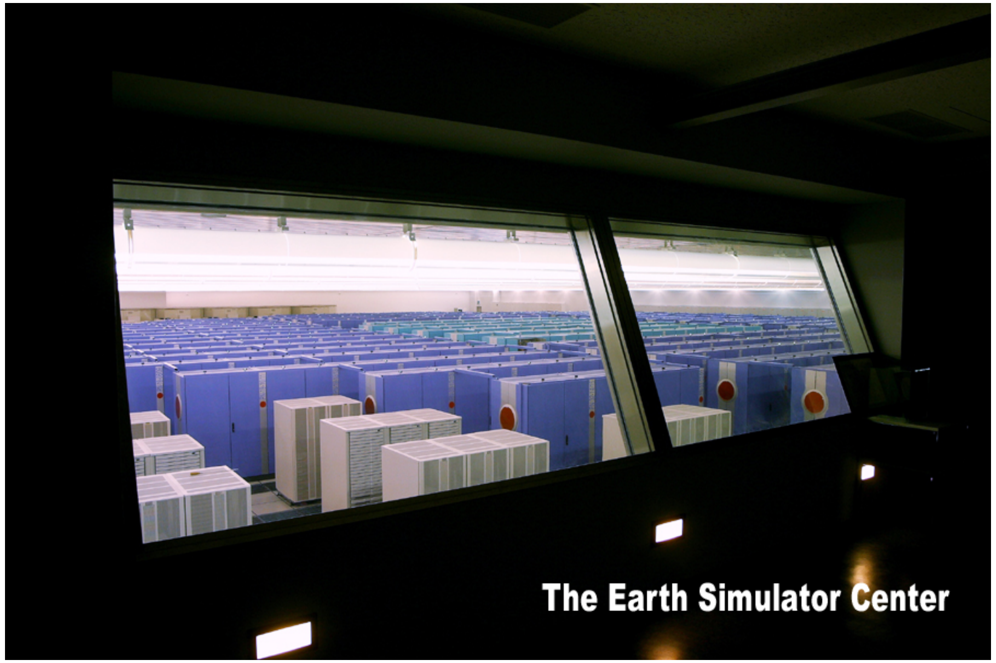
  return (F($n-1$) + F($n-2$))

- 存在问题：
  - 计算量随着n的增大而急剧增大
  - What is f_{200}?

算法分析: Fibonacci

- What is f_{200}?
  - Need $2^{0.694n}$ operations to compute Fn.
  - Eg. Computing F_{200} needs about 2^{140} operations.
- How long does this take on a fast computer?

NEC Earth Simulator



40万亿

Can perform up to 40 trillion operations per second.

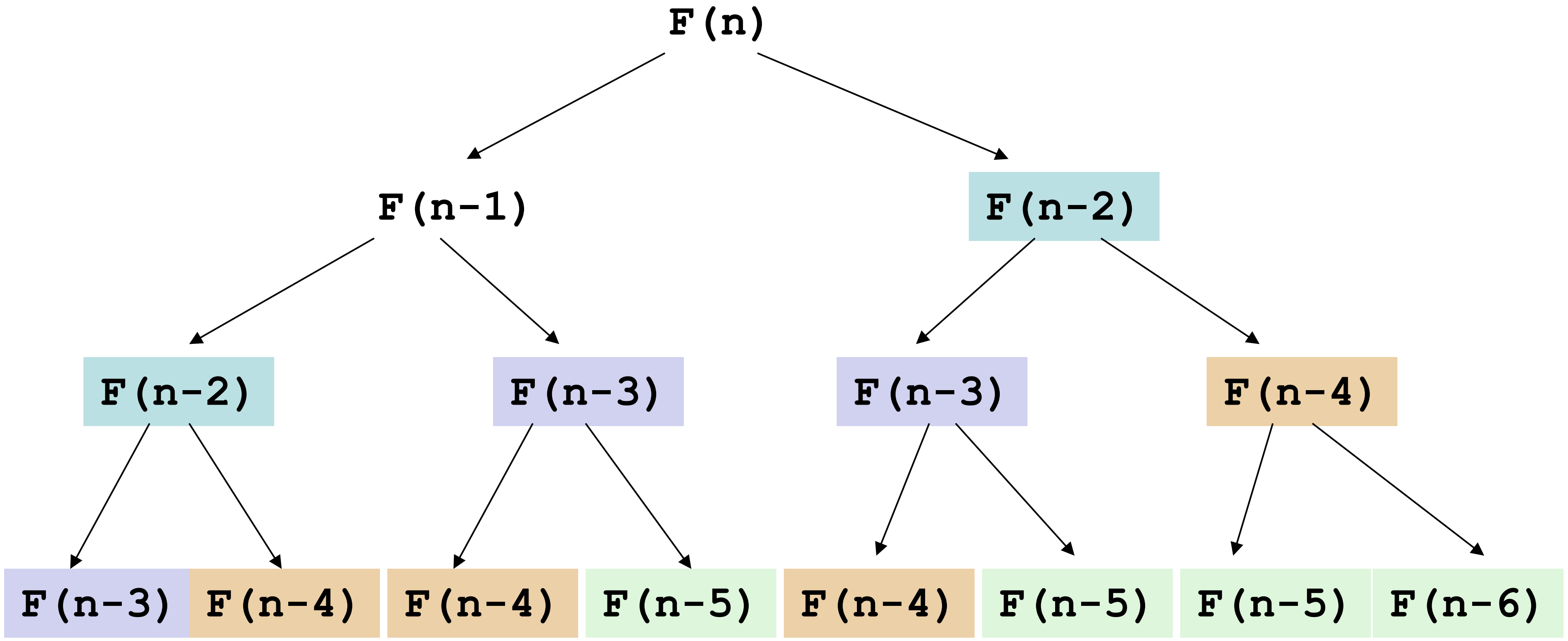
算法分析: Fibonacci

- The Earth simulator need 2^{95} seconds for f_{200}.
- Time in seconds — Interpretation

Time in seconds	Interpretation
2^{10}	17 minutes
2^{20}	12 days
2^{30}	32 years
2^{40}	…

算法分析: Fibonacci

- 为什么复杂度那么高？ Let’ s unravel the recursion…



相同的子问题被重复计算!

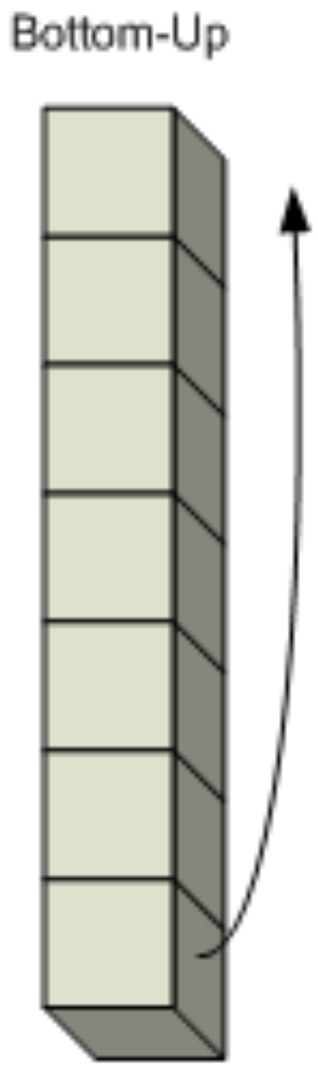
启示：优秀就是一种习惯 （勿以恶小而为之）

算法分析: Fibonacci

- 面试官期待的解法
  - 改进算法：保存已经得到的中间项

```
Algorithm 2  F(n)
//Initially we create an array F[1: n]
F[1] ← 1, F[2] ← 1
for i = 3 to n do
    F[i] ← F[i−1] + F[i−2]
return (F[n])
```

O(n)



分析： $T(n)=O(n)$

算法分析: Fibonacci

- 实现的两种算法

```
Algorithm 1  F(n)
Input： n
Output： F(n)
if n ≤ 2 then
  return (1)
else
  return (F(n−1) + F(n−2))
```

```
Algorithm 2  F(n)
Input： n
Output： F(n)
//Initially we create an array F[1: n]
F[1] ← 1, F[2] ← 1
for i = 3 to n do
  F[i] ← F[i−1] + F[i−2]
return (F[n])
```

算法分析: Fibonacci

还能改进吗？

- 提示：
  - O(logn)

实践能力训练

- 青蛙跳台阶问题
  - 一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个n级的台阶总共有多少种跳法？

实践能力训练： Fibonacci扩展题

- 输入：frog, n
- 输出：跳法种类数jumpN(n)
- 青蛙跳台阶问题分析
  - 如果只有1级台阶，那显然只有一种跳法 jumpN(1)=1
  - 如果有2级台阶，那么就有2种跳法: jumpN(2)=2
    - 一种是分2次跳, 每次跳1级
    - 另一种就是一次跳2级
  - 如果台阶级数大于2，设为n的话，这时我们把n级台阶时的跳法看成n的函数，记为jumpN(n)
    - 第一次跳的时候有2种不同的选择：一是第一次跳一级，此时跳法的数目等于后面剩下的n-1级台阶的跳法数目，即为jumpN(n-1), 二是第一次跳二级，此时跳法的数目等于后面剩下的n-2级台阶的跳法数目，即为jumpN(n-2)
    - 因此n级台阶的不同跳法的总数为jumpN(n-1)+jumpN(n-2)，就是斐波那契数列

多项式级Polynomial vs. 指数级exponential

- Running times like

  n, n^2, n^3, 是多项式级
- Running times like

  $2^n, e^n, 2^{\sqrt{n}}$ 是指数级

- 基本常识:

  多项式级（polynomial） is reasonable

  指数级（exponential） is not reasonable

算法分析方法 - 案例

- 例1：顺序搜索算法

输入：a , n, k ;

输出：值k的位置；//寻找k值

```
int seqSearch(Type *a, int n, Type k)
{
    for(int i=0;i<n; i++)
        if (a[i]==k) return i;
    return -1;
}
```

分析：

规模

基本操作

三种情况

算法分析方法 - 案例

- 最坏情况：$T_{max}(n)$ = max{ T(I) | size(I)=n }=O(n)
- 最好情况：$T_{min}(n)$ = min{ T(I) | size(I)=n }=O(1)
- 在平均情况下，假设：
  - (a) 搜索成功的概率为p (0 ≤ p ≤ 1)；
  - (b) 在数组的每个位置i (0 ≤ i < n)搜索成功的概率相同，均为 p/n。

基本操作：
比较k值

$$T_{avg}(n) = \sum_{size(I)=n} p(I)T(I)$$

$$= \left(1\cdot\frac{p}{n} + 2\cdot\frac{p}{n} + 3\cdot\frac{p}{n} + \cdots + n\cdot\frac{p}{n}\right) + n\cdot(1-p)$$

$$= \frac{p}{n}\sum_{i=1}^{n} i + n(1-p) = \frac{p(n+1)}{2} + n(1-p)$$

算法分析的基本法则

- 算法的运行时间：指在特定输入时,所执行的基本操作数,这是独立于具体机器的
- 非递归算法：
  - for / while 循环
    - 循环体内计算时间*循环次数
  - 嵌套循环
    - 循环体内计算时间*所有循环次数
  - 顺序语句
    - 各语句计算时间相加
  - if-else语句
    - if语句计算时间和else语句计算时间的较大者

算法分析方法 - 案例

- 例2：插入排序

问题:将一列数按非递减顺序排列

输入: n个数$< a_1, a_2, \dots, a_n >$

输出: 输入序列的一个排列（即重新排序）$< a'_1, a'_2, \dots, a'_n >$,使得

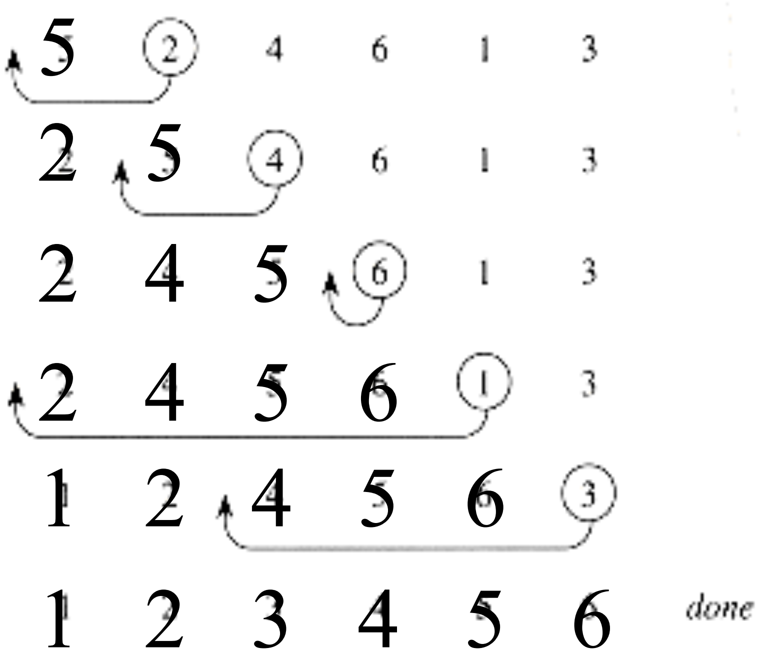
$$a'_1 \le a'_2 \le \cdots \le a'_n$$

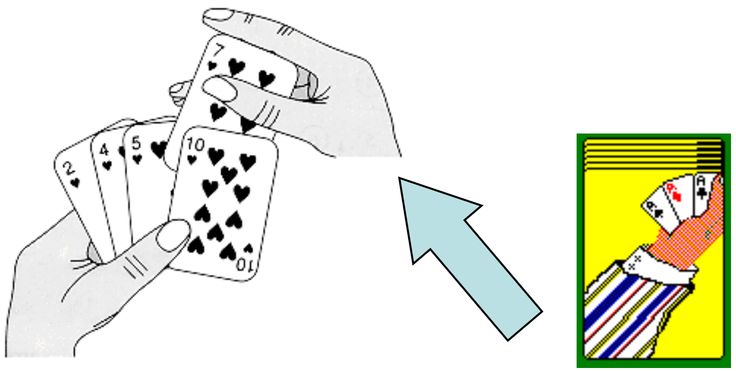
基本思想：将一个记录插入到**已经排好顺序的有序表**中，从而得到一个新的有序表。

算法分析方法 - 案例

- 例2：插入排序

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





算法分析方法 - 案例

- 基本思想：
  - 将一个记录插入到已经排好顺序的有序表中，从而得到一个新的有序表。

抓牌

移动：
找位置

插入

```
void insertion_sort(Type *a, int n){
    Type key;     //当前处理对象          //  cost      times
    for (int i = 1; i < n; i++){          //  c1        n
        key=a[i];    /*复制到临时区*///    c2        n-1
        int j=i-1;                        //  c3        n-1
        while( j>0 && a[j]>key ){         //  c4        sum of ti
            a[j+1]=a[j];  /*记录后移*/    //  c5        sum of (ti-1)
            j--;                          //  c6        sum of (ti-1)
        }
        a[j+1]=key;                       //  c7        n-1
    }          /*插入到正确的位置*/
}
```

分析：
输入规模
基本操作
三种情况？

算法分析方法 - 案例

$$T(n)=c_1n+c_2(n-1)+c_3(n-1)+c_4\sum_{i=1}^{n-1}t_i+c_5\sum_{i=1}^{n-1}(t_i-1)+c_6\sum_{i=1}^{n-1}(t_i-1)+c_7(n-1)$$

- **在最好情况下**，$t_i=1$, for $1 \le i < n$;

无需移动牌

$$T_{\min}(n)=c_1n+c_2(n-1)+c_3(n-1)+c_4(n-1)+c_7(n-1)$$

$$=(c_1+c_2+c_3+c_4+c_7)n-(c_2+c_3+c_4+c_7)=\boxed{O(n)}$$

- **在最坏情况下**，$t_i = i+1$, for $1 \le i < n$;

输入数组按逆序排列，全部移动

（即输入数据a ‘[i]=n-i, i=0,1,…,n-1时，达到其最坏情形

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

$$T_{\max}(n)\le c_1n+c_2(n-1)+c_3(n-1)+$$

$$c_4\left(\frac{n(n+1)}{2}-1\right)+c_5\left(\frac{n(n-1)}{2}\right)+c_6\left(\frac{n(n-1)}{2}\right)+c_7(n-1)$$

$$=\frac{c_4+c_5+c_6}{2}n^2+\left(c_1+c_2+c_3+\frac{c_4-c_5-c_6}{2}+c_7\right)n-(c_2+c_3+c_4+c_7)$$

$$=\boxed{O(n^2)}$$

分析非递归算法的通用方案

- 步骤1 决定用哪个参数作为输入规模的度量
- 步骤2 找出算法的基本操作

规律：总是位于算法的最内层循环中

- 步骤3 检查基本操作的执行次数是否只依赖输入规模
  - 如果还依赖一些其他的特性，则最坏效率、平均效率以及最好效率需要分别研究
- 步骤4 建立一个算法基本操作执行次数的求和表达式（或者是**递推表达式**）
- 步骤5 利用求和运算的标准公式和法则来建立一个操作次数的公式

递归算法复杂性分析

- 阶乘问题

```
int factorial(int n){
    if (n == 0) return 1;
    return n*factorial(n-1);
}
```

T(n)

T(n-1)+1

边界(初始)条件
递推关系

分析：

1 输入规模: n

2 基本操作：乘法

3 检查是否需要三种情况分析

4 列求和公式

5 计算

$$T(n)=\begin{cases}0 & n=0\\ T(n-1)+1 & n>0\end{cases}$$

不做乘法

计算factorial(n-1)

将n乘以f(n-1)

$$T(n)=n$$

分析递归算法的通用方案

- 步骤1 决定用哪个参数作为输入规模的度量
- 步骤2 找出算法的基本操作
- 步骤3 检查一下，对于相同规模的不同输入，基本操作的执行次数是否不同。如果不同，则必须对最坏情况、最好情况以及平均情况做单独研究
- 步骤4 对于算法基本操作的执行次数，建立一个递推关系以及相应的初始条件
- 步骤5 解递推式，或者至少确定它的解的增长次数

思考

- 目前案例中的分析都是计算出程序执行的次数
  - 如何表示出来？
  - 单位是什么

- 如何从大局观念上进行分析
  - 渐近计法：使用一组由希腊字母构成的记号体系

算法分析

- 分析一个简单的算法也可能是一个挑战问题
  - 数学功底好
    - 需要的数学: combinatorics, probability theory, algebraic dexterity
  - 洞察能力强，能抓住最重要的部分
  - 抽象思维强，能将算法的本质特征归纳成简单形式

小结

- 主要内容
  - 算法分析
    - 正确性分析
    - 复杂性分析
  - 算法的计算复杂性
    - 计算公式
    - 案例分析

- 重点
  - 算法复杂性的概念、表示和计算公式
  - 算法分析的基本方法：计数就是算法分析的核心