Algorithms
Lecture 1: 算法分析基本概念

姜丽红／jianglh@sjtu.edu.cn

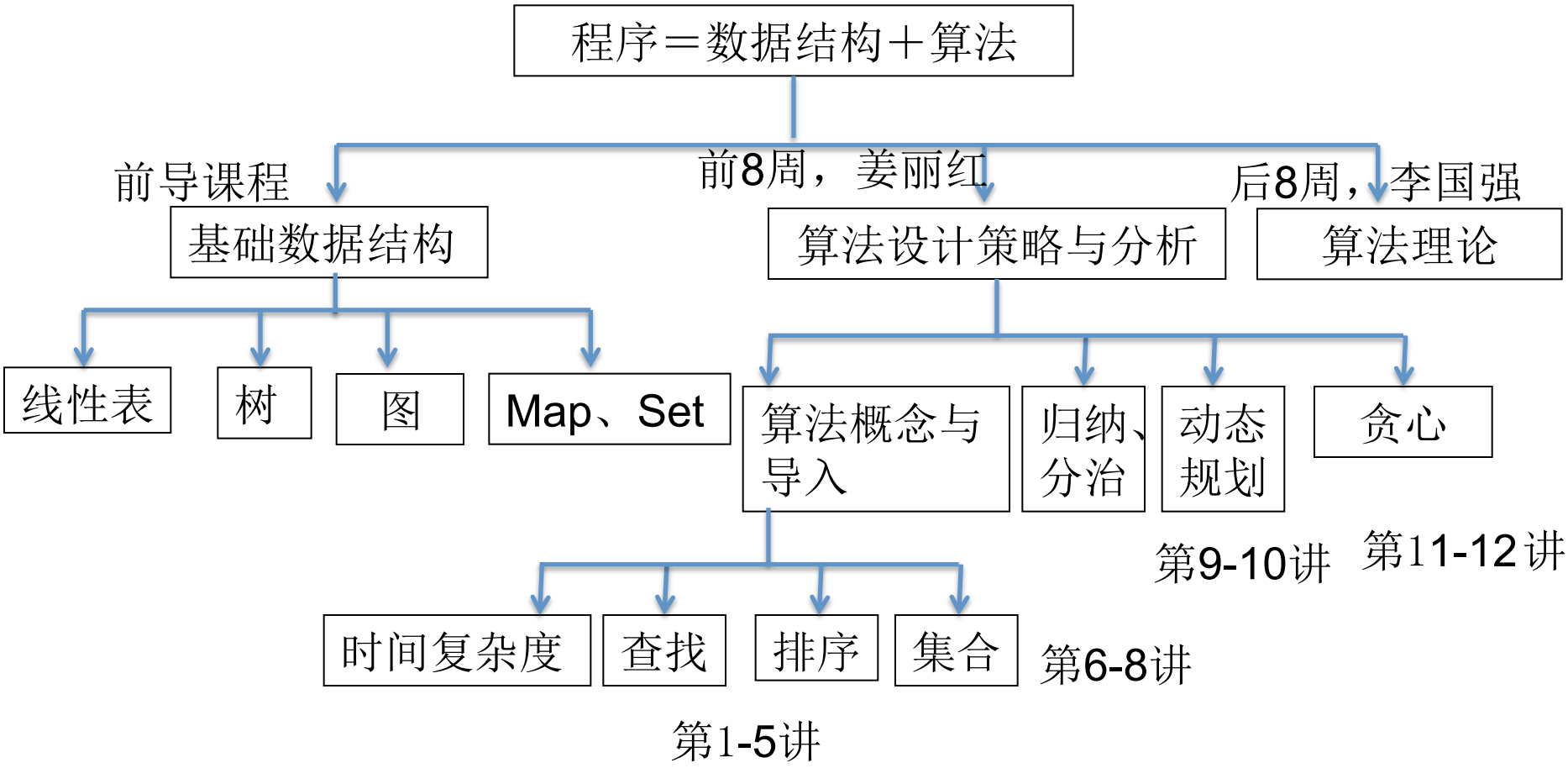
李国强

助教：张翼飞（前8周）

z71737173@126.com

软件大楼5310信息系统实验室

课程介绍


程序＝数据结构＋算法
前导课程
前8周，姜丽红
后8周，李国强
基础数据结构
算法设计策略与分析
算法理论
线性表
树
图
Map、Set
算法概念与导入
归纳、分治
动态规划
贪心
第9-10讲
第11-12讲
时间复杂度
查找
排序
集合
第6-8讲
第1-5讲

课程介绍

- **Textbook:**
  - *教材：算法设计技巧与分析* [沙特] M.H.Alsuwaiyel 电子工业出版社
  - 主要参考教材：算法概论，王沛等，清华大学出版社
- **Grading Policy:**
  - 考勤&Homework: 30%
  - Final Exam: 70%

本章内容

- 1.1 引言：算法概念
- 1.2 背景：算法的重要性
- 1.3 二分查找
- 1.4Merge算法
- 1.5 选择排序
- 1.6 插入排序
- 1.7 归并排序
- 1.8 时间复杂度概念
- 1.9 空间复杂度概念
- 1.10 最优算法概念
- 1.11 时间复杂度分析方法
- 1.12 最坏情况和平均情况分析
- 1.13 平摊分析

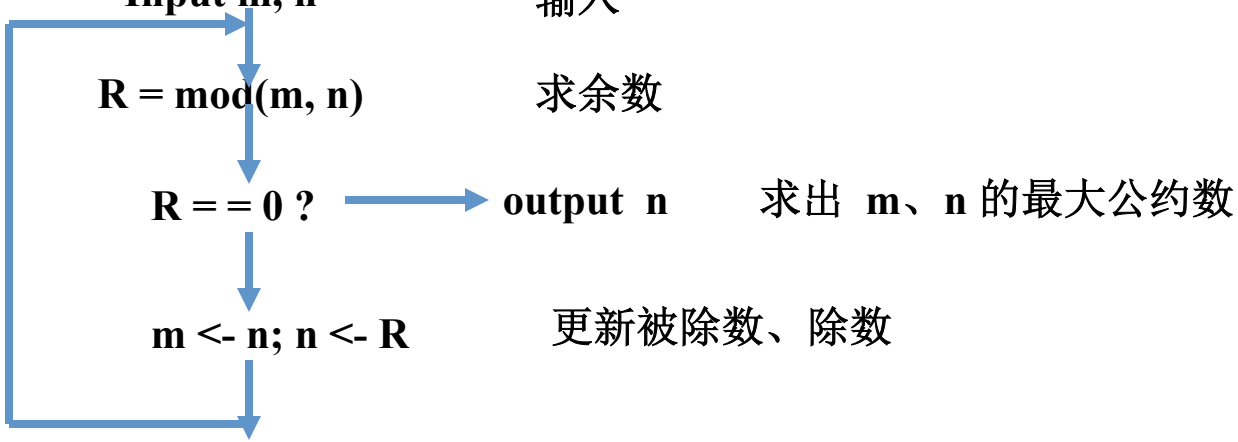
算法概念

算法：一个算法就是有穷规则的集合，其中的规则规定了一个解决某一个特定问题的运算序列。

欧几里德算法：如果 a、b 都是正整数，且 a >b 。 则 a ＝ bq + r , 0 < r < b, 此处 r,q 都是正整数。那么，（a,b) = (b, r) ; 符号（a,b) 为 a、b之间的最大公约数。即：设 m、n 都是正整数，且 m > n; 那么 m、n 之间的最大公约数可以计算如下：

$$g(m,n) = \begin{cases} n & \text{if } mod(m,n) = 0 \\ g(n, mod(m,n)) & \text{if } mod(m,n) != 0 \end{cases}$$

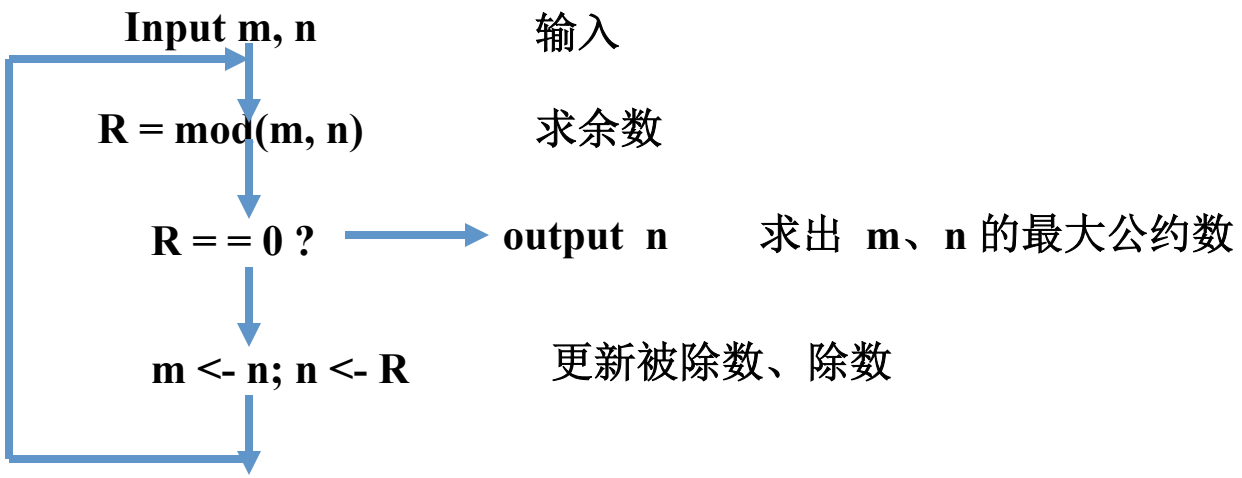
程序框图：



算法概念

算法：一个算法就是有穷规则的集合，其中的规则规定了一个解决某一个特定问题的运算序列。

程序框图：



·特征：1、有穷性：执行有穷步后结束。
2、确定性：每一步有确定的含义。
3、能行性：原则上能精确的进行，用纸和笔有限次完成。
4、输入
5、输出

算法概念

算法1.1：求m、n 之间的最大公约数

```
// 输入两个正整数m、n, 满足关系 m > n 。现在要求m 和n 之间的最大公约数。
int maximal_common_divisor( ) {
int m,n; int r; // r用于保存 m % n 之值。
cout << "Please enter two positive intergers m n ,note: m > n !" << endl;
cin >> m >> n;  // 输入m 和 n 之值。
Exception( m < n, “m <n is ERROR.” );  //若 m> n,继续；否则出错，程序中断。
cout << " m is " << m << "!" << " n is " << n << endl; //显示m、n。
while ( 1 ){
        r = m % n;      // 得到 m/n 之后的余数。
        if ( r == 0 ) return n; // 若余数为0，那么原 m和n之间的最大公约数为 n。
        else { m = n;  // 更新被除数 m。
               n = r;  // 更新除数 n。
             }
}
} // 在 m > n 时，函数返回正整数 m 和n之间的最大公约数。
```

1.1 引言:程序=数据结构+算法

* 例子:26个英文字母全排列，它的排列数为:
* $26！\approx 4\times 10^{26}$
* 以每年365天计算，共有
* $365\times 24\times 3600=3.1536\times 10^{7}$秒
* 以每秒能完成10^{7}个排列的超高速电子计算机来做这项工作，需要
* $4\times 10^{26}/(3.1536\times 10^{14})\approx 1.2\times 10^{12}$年
* 即使计算机运行速度随着技术的提高，恐怕也还是不可能实现的。因计算机的速度再提高也有它的极限。

引例:程序=数据结构+算法

百钱买百鸡问题

100元钱买100只鸡，母鸡每只5元，公鸡每只3元，小鸡3只1元，问共可以买多少只母鸡、多少只公鸡、多少只小鸡？

求解：设母鸡、公鸡、小鸡各为x, y, z只。则有：

$$x + y + z = 100$$

$$5x + 3y + z/3 = 100$$

只需要解出本方程就可以得到答案。

1.1 引言:程序=数据结构+算法

方法1：用三重循环：

```
for(I=0; I<=100; I++)
        for(j=0; j<=100; j++)
                for(k=0; k<=100;k++)
        {       if(k%3 = = 0 && I+j+k= =100          \
                                && 5*I+3*j+k/3 = =100)
                        printf( “%d,%d,%d\n” , I,j,k);
        }
```

1.1 引言:程序=数据结构+算法

- 方法2：用两重循环：
- 因总共买100只鸡，所以小鸡的数目可以由母鸡数和公鸡数
- 得到。

```
For(l=0;l<100;l++)
 for(j=0;j<100;j++)
 {   k=100 –i –j ;
     if(k%3==0 && 5*l+3*j+k/3==100)
         printf(“%d,%d,%d\n”,l,j,k);
 }
```

1.1 引言:程序=数据结构+算法

方法3：用两重循环：

钱共100元，而母鸡5元1只，所以母鸡数不可能超过20只，同样，公鸡数也不超过33只。

```
For(I=0;I<=20;I++)
        for(j=0;j<33;j++)
        {       k=100 –i –j ;
                if(k%3==0 && 5*I+3*j+k/3==100)
                        printf(“%d,%d,%d\n”,I,j,k);
        }
```

1.1 引言:程序=数据结构+算法

- 方法4：用一重循环
- 由x+y+z = 100和5*x+3*y+z/3=100可以合并为一个方程：14*x+8*y = 200, 进一步简化为:7*x+4*y=100
- 从上面方程可见，x不超过14，并可以进一步判断x必为4的倍数，于是有：
- For(l=0;l<=14;l+=4)
  { j = (100 –7*l)/4;
- k=100 –i –j;
  if(k%3==0 && 5*l+3*j+k/3==100)
- printf("%d,%d,%d\n", l,j,k);
- }

1.1 引言:程序=数据结构+算法

上面四个方法中，第一个方法的循环次数为：　100*100*100即一百万次；第二个方法的循环次数为：100*100，1万次；第三个方法为：20*34，680次，第四个方法为：4次，由此可见，算法的设计至关重要。

返回

1.2历史背景 研究算法的重要性

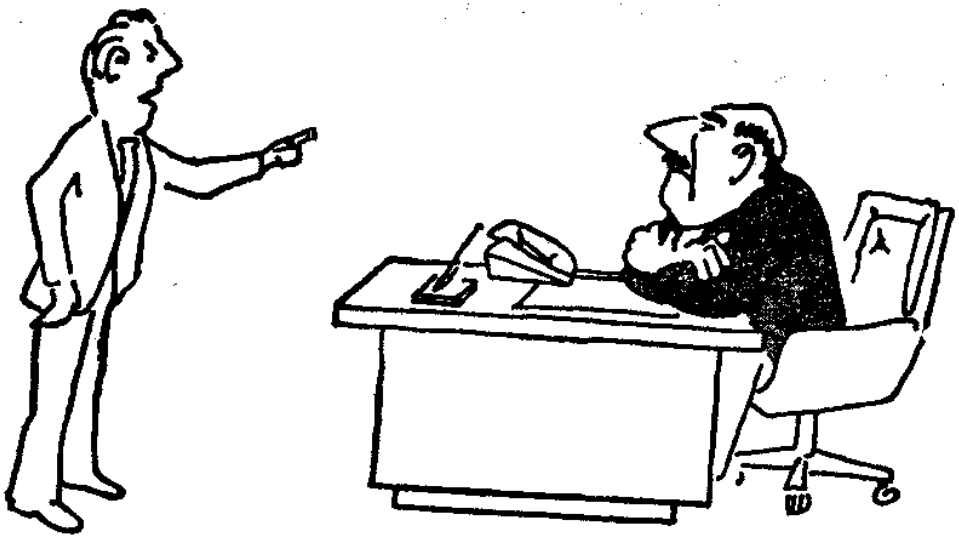
问题能解决吗(20世纪30年代，可计算性理论)

假设某一负责人交给你一个很难的任务，几天后……

失败！

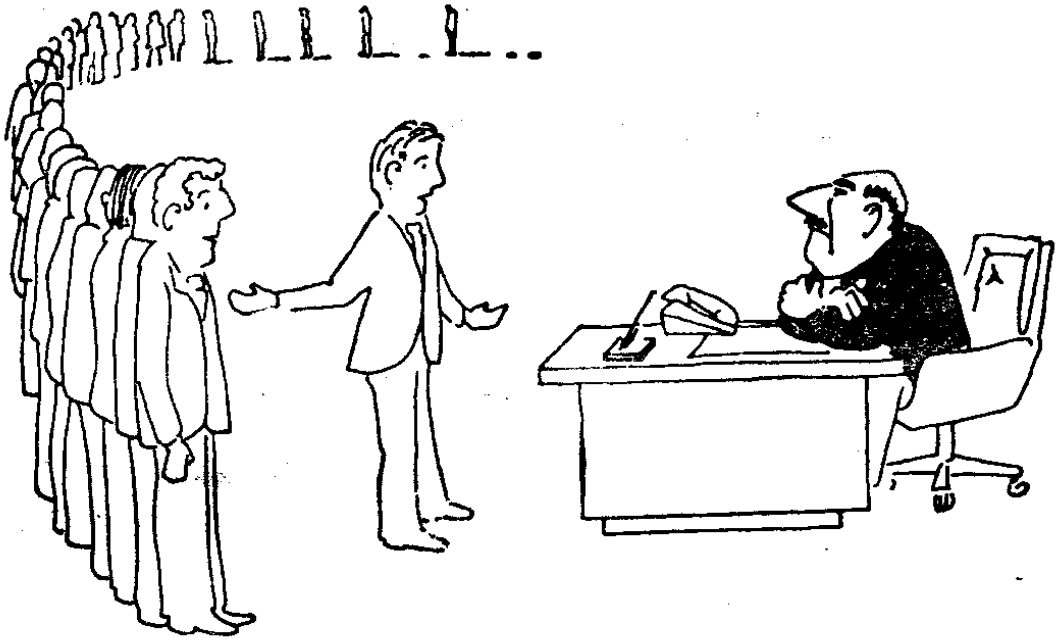
1.2历史背景 研究算法的重要性

Done!



问：“交给你的问题，解决方案设计出来了吗？”
答：“我找不到一个有效的算法来解决它，因为这样的算法是不存在的。”
(不过，要证明一个问题不存在有效算法，往往跟寻找有效算法一样难。)

1.2历史背景 研究算法的重要性



问：“交给你的问题，解决方案设计出来了吗？”
答：“我找不到一个有效的算法来解决它，但不是我不行，因为所有这些名人也都找不到解决它的有效算法。”

如果是你的话，你愿意是哪种结果？

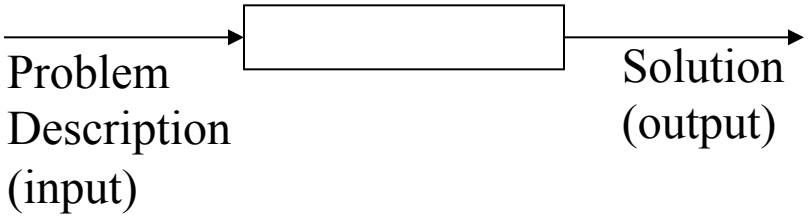
问题解决的好吗？（算法的效率）

设计出解决问题的算法后，要知道算法的优劣好坏，是好算法则不必对其怀疑而再浪费时间进行研究。不是好算法则应再进行研究改进。而如何知道算法的优劣好坏，需要学习分析算法的方法。

要设计出好算法，需要学习算法设计的常用方法。

1.2历史背景 研究算法的重要性

- 有效算法的标准：
- （1）正确性
- （2）有效性
- （3）鲁棒性
- （4）可读性



有效算法的重要性

算法的重要性：

并非所有的算法都有实际意义。

举一个例子加以说明。假定时间复杂性函数的时间单位为 us。

函数	n=20	n=50	n=100	n=500	
1000n	.02s	.05s	.15s	.5s	易性算法
1000nlogn	.09s	.3s	.6s	4.5s	
$100n^2$	.04s	.25s	1s	25s	
$10n^3$	.02s	1s	10s	21分	
n^{logn}	.4s	1.1小时	220天	无穷	
$2^{n/3}$	.0001s	0.1s	2.7小时	5×10^8世纪	顽性算法
2^n	58分	2×10^9世纪			

有效算法的重要性

算法的重要性：

·硬件速度的提高和问题的求解规模之间的关系：硬件速度提高 10 倍，并不意味着求解问题的规模同样提高 10 倍。由于算法不同，差别很大。设时间复杂性函数的时间单位为ms。

提速之前	在 1 秒内	在 1 分钟内	在 1 小时
A1：n	1000	$6 * 10^4$	$3.6 * 10^6$
A2：nlogn	140	4893	$2 * 10^5$
A3：n^2	31	244	1897
A4：n^3	10	39	153
A5：2^n	9	15	21

例如：设 1 分钟内算法 A5的求解规模为 s5:

1秒内算法的运行次数为1000

则：$2^{s5} = 60 * 10^3$

解之：s5 = 15

有效算法的重要性

有效算法的重要性：

·硬件速度的提高和问题的求解规模之间的关系：硬件速度提高 10 倍，并不意味着求解问题的规模同样提高 10 倍。由于算法不同，差别很大。设时间复杂性函数的时间单位为ms。

提速10倍后	提速10倍前求解规模	提速10倍后求解规模
A1：n	s1	10s1
A2：nlogn	s2	10s2
A4：n^2	s3	3.16s3
A3：n^3	s4	2.15s4
A5：2^n	s5	s5+3.3

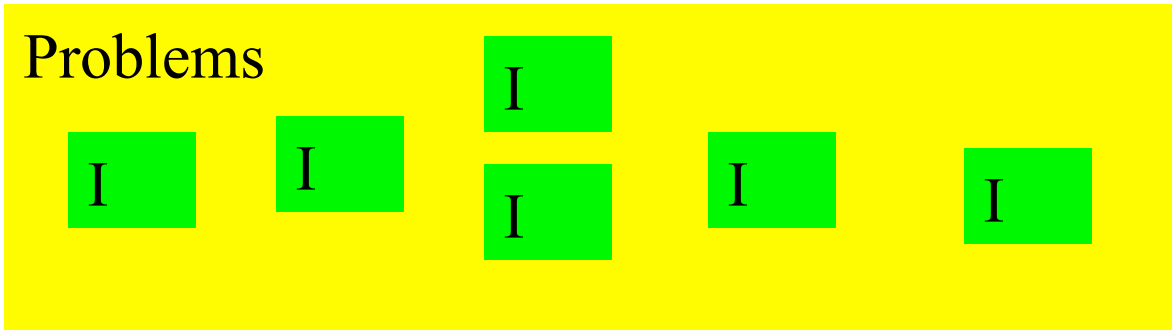
例如：设提速 10 倍后在时间 t 秒内，A5 的求解规模为 z5:

$$\frac{2^{s5}}{2^{z5}} = \frac{t * 1000}{t * 1000*10}$$

解之：z5 = s5 +3.3

1.3 Getting Started:几个例子
Problem and Instance

- Problem: well-defined statement specifies in general terms the desired input/output relationship
- Instance (of a problem): input values needed to compute a solution to the problem



- An algorithm should solve a problem, i.e. all the instances of a problem(例如：排序问题n有n!个实例)

算法入门例子

- 查找
- 排序

顺序查找

Linear search（顺序查找） (p3)(又称线性查找)

Algorithm 1.1 LINEARSEARCH

Input: An array A[1..n] of n elements and an element x

Output: j if x=A[j], 1≤j≤n, and 0 otherwise

1. j←1
2. **while (j<n) and (x≠A[j])**
3. j←j+1
4. end while
5. if x=A[j] then return j else return 0

元素比较次数：最少1次，最多n次

二分查找Binary search

Algorithm 1.2 BINARYSEARCH(p4)

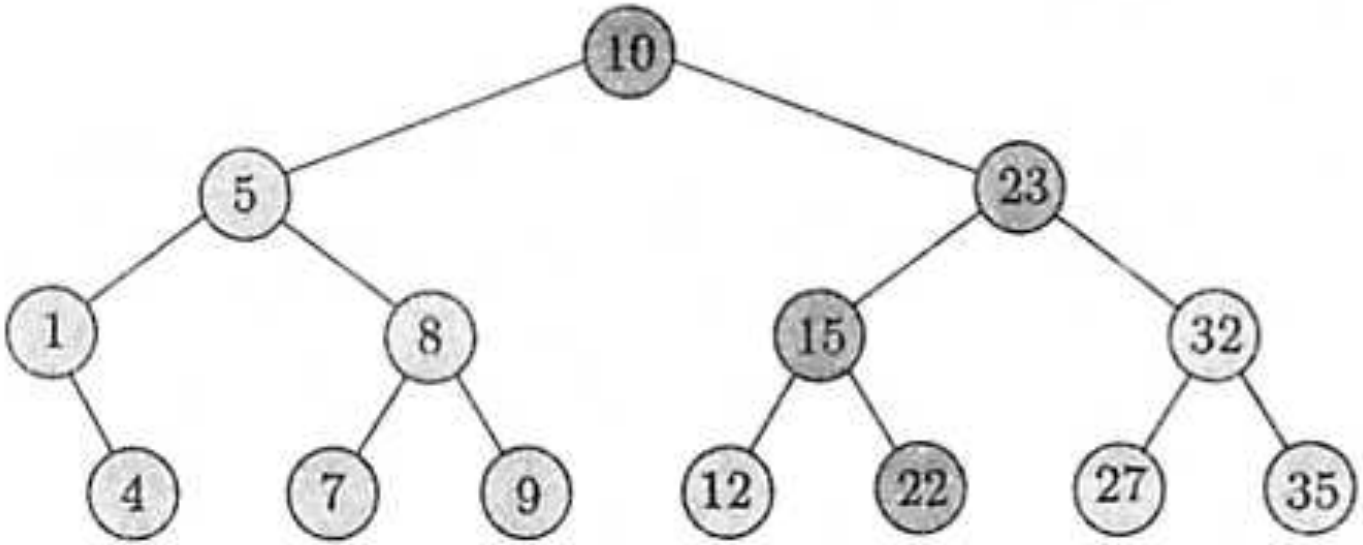
Input: An array A[1..n] of n elements sorted in nondecreasing order and an element x

Output: j if x=A[j], $1 \le j \le n$, and 0 otherwise

1. low←1, high←n, j←0
2. while (low≤high) and (j=0)
3. mid←$\lfloor$(low+high)/2$\rfloor$
4. **if x=A[mid] then j←mid**
5. **else if x<A[mid] then high←mid-1**
6. **else low←mid+1**
7. end while
8. return j

元素比较次数：最少1次，最多$\lfloor \log n \rfloor$+1次

二分查找性能分析
Decision tree(p5 例1.2）



A[1..14]=1, 4, 5, 7, 8, 9, 10, 12, 15, 22, 23, 27, 32, 35

1.4Merge

Algorithm 1.3 MERGE

Input: An array A[1..m] of elements and three indices p, q and r, with $1 \le p \le q < r \le m$, such that both the subarrays A[p..q] and A[q+1..r] are sorted individually in nondecreasing order

Output: A[p..r] contains the result of merging the two subarrays A[p..q] and A[q+1..r]

Merge

1. comment: B[p..r] is an auxiliary array
2. s←p, t←q+1, k←p
3. while s≤q and t≤r
4. if A[s]≤A[t] then
5. B[k]←A[s]
6. s←s+1
7. else
8. B[k]←A[t]
9. t←t+1
10. end if
11. k←k+1
12. endwhile
13. if s=q+1 then B[k..r]←A[t..r]
14. else B[k..r]←A[s..q]
15. endif
16. A[p..r]←B[p..r]

Merge 算法分析

假设：Two array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$

元素比较次数最少：$\lfloor n/2 \rfloor$，最大：n-1

元素赋值次数： 2n

1.5 Selection sort

对于数组A[]：45，33，24，45，12，12，24，12

Algorithm 1.4 SELECTIONSORT(p8)

Input: An array A[1..n] of n elements

Output: A[1..n] sorted in nondecreasing order

1. for i←1 to n-1
2. k←i
3. for j←i+1 to n {Find the ith smallest element}
4. if A[j]<A[k] then k←j
5. end for
6. if k≠i then interchange A[i] and A[k]
7. end for

Selection sort 分析

元素比较次数为n(n-1)/2.

元素赋值次数介于 0 and 3(n-1)

分析前面的例子算法执行次数分析

数组A[]：45，33，24，45，12，12，24，12

1.6插入排序Insertion sort

Algorithm 1.5 INSERTIONSORT
Input: An array A[1..n] of n elements
Output: A[1..n] sorted in nondecreasing order

```
1. for i←2 to n
2.     x←A[i]
3.     j←i-1
4.     while (j>0) and (A[j]>x)
5.         A[j+1]←A[j]
6.         j←j-1
7.     end while
8.     A[j+1]←x
9. endfor
```

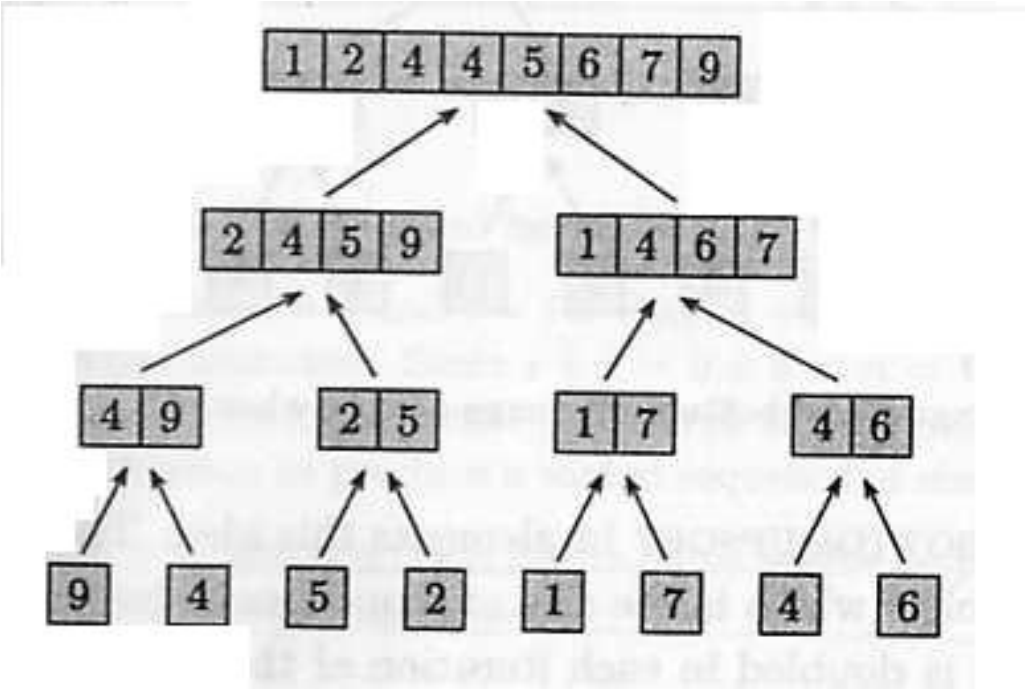
Insertion sort

元素比较次数最少为n-1，最多为n(n-1)/2.

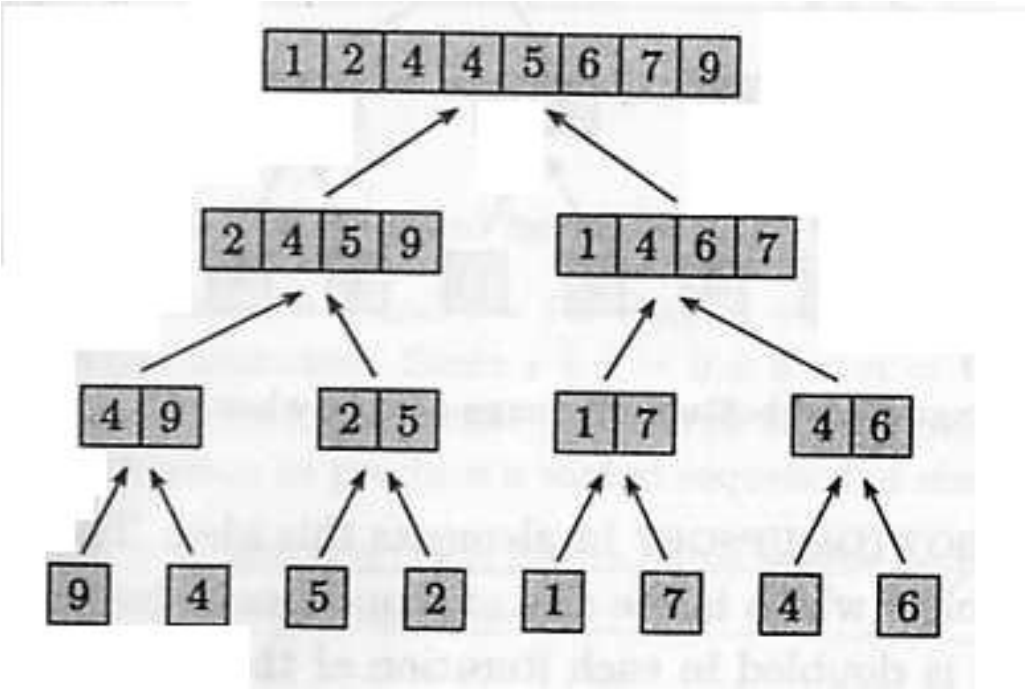
元素赋值次数为元素比较次数＋n-1

30,12,13,13,44,12,25,13

1.7 自底向上合并排序
Bottom-up merge sorting



1.7 自底向上合并排序
Bottom-up merge sorting



1.7 Bottom-up Merge Sorting

Algorithm 1.6 BOTTOMUPSORT
Input: An array A[1..n] of n elements
Output: A[1..n] sorted in nondecreasing order

```
1. t←1
2. while t<n
3.       s←t, t←2s, i←0
4.       while i+t≤n
5.           MERGE(A, i+1, i+s, i+t)
6.           i←i+t
7.       end while
8.       if i+s<n then MERGE(A, i+1, i+s, n)
9. endwhile
```

Analysis of BottomUpSort

Suppose that n is a power of 2, say $n = 2^k$.

- The outer **while** loop is executed $k = \log n$ times.
- Step 8 is never invoked.
- In the j-th iteration of the outer **while** loop, there are $2^{k-j} = n/2^j$ pairs of arrays of size 2^{j-1}.
- The number of comparisons needed in the merge of two sorted arrays in the j-th iteration is at least 2^{j-1} and at most $2^j - 1$.
- The number of comparisons in BottomUpSort is at least

$$\sum_{j=1}^{k}(\frac{n}{2^j})2^{j-1} = \sum_{j=1}^{k}\frac{n}{2} = \frac{n\log n}{2}$$

- The number of comparisons in BottomUpSort is at most

$$\sum_{j=1}^{k}(\frac{n}{2^j})(2^j - 1) = \sum_{j=1}^{k}(n - \frac{n}{2^j}) = n\log n - n + 1$$

元素赋值次数2nlogn

1.8.1阶的增长：时间复杂度Time complexity（例1.5-1.15）

Running time of a program is determined by:

1) input size（问题规模）
2) quality of the code（代码质量）
3) quality of the computer system（硬件速度）
4) time complexity of the algorithm（时间复杂度）

We are mostly concerned with the behavior of the algorithm under investigation on large input instances.（大规模输入情况下的算法性能）

抽象时间衡量，非实际运行时间

独立于运行环境的时间衡量

So we may talk about the rate of growth or the order of growth of the running time（渐进性时间复杂度）

算法效率(时间复杂度)的衡量方法和准则

通常有两种衡量算法效率的方法:

事后统计法

缺点：1。必须执行程序

2。其它因素掩盖算法本质

事前分析估算法

算法 = 控制结构 + 原操作（或元运算）

（固有数据类型的操作）

算法的执行时间 =

$\sum$原操作(i)的执行次数×原操作(i)的执行时间

算法的执行时间

与

原操作执行次数之和　成正比

一个特定算法的“运行工作量”的大小，只依赖于问题的规模（通常用整数量n表示），或者说，它是问题规模的函数f(n)。

1.8.2 O-notation

Definition1.2: Let f(n) and g(n) be two functions from the set of natural numbers to the set of nonnegative real numbers. f(n) is said to be O(g(n)) if there exists a natural number n0 and a constant c>0 such that

$\forall n \geq n0$, $f(n) \leq cg(n)$

Consequently, if limf(n)/g(n) exists, then

$\lim f(n)/g(n) \neq \infty$ implies $f(n)=O(g(n))$

Informally, this definition says that f grows no faster than some constant times g.

1.8.3 Ω-notation

Definition: Let f(n) and g(n) be two functions from the set of natural numbers to the set of nonnegative real numbers. f(n) is said to be Ω(g(n)) if there exists a natural number n0 and a constant c>0 such that

$\forall n \geq n0$, $f(n) \geq cg(n)$

Consequently, if limf(n)/g(n) exists, then

limf(n)/g(n)≠0 implies f(n)=Ω(g(n))

Informally, this definition says that f grows at least as fast as some constant times g.

f(n)=Ω(g(n)) if and only if g(n)=O(f(n))

1.8.4 Θ-notation

Definition: Let f(n) and g(n) be two functions from the set of natural numbers to the set of nonnegative real numbers. f(n) is said to be Θ(g(n)) if there exists a natural number n0 and two positive constants c1 and c2 such that

$$\forall n \geq n0,\ c1g(n) \leq f(n) \leq c2g(n)$$

Consequently, if limf(n)/g(n) exists, then

$$\lim f(n)/g(n) = c \text{ implies } f(n) = \Theta(g(n))$$

where c is a constant strictly greater than 0

f(n)=Θ(g(n)) if and only if f(n)=Ω(g(n)) and f(n)=O(g(n))

Examples

- $10n^2 + 20n = O(n^2)$.
- $\log n^2 = O(n)$.
- $\log n^k = \Omega(\log n)$.
- $n! = O((n+1)!)$.

Examples

Consider the series $\sum_{j=1}^{n} \log j$. Clearly,

$$\sum_{j=1}^{n} \log j \leq \sum_{j=1}^{n} \log n = n \log n$$

On the other hand,

$$\sum_{j=1}^{n} \log j \geq \sum_{j=1}^{\lfloor n/2 \rfloor} \log(\frac{n}{2}) = \lfloor n/2 \rfloor \log(\frac{n}{2}) = \lfloor n/2 \rfloor \log n - \lfloor n/2 \rfloor$$

o-notation

Definition: Let f(n) and g(n) be two functions from the set of natural numbers to the set of nonnegative real numbers. f(n) is said to be o(g(n)) if for every constant c>0 there exists a positive integer n0 such that f(n)<cg(n) for all n≥n0

Consequently, if limf(n)/g(n) exists, then

limf(n)/g(n)=0 implies f(n)=o(g(n))

Informally, this definition says that f(n) becomes insignificant relative to g(n) as n approaches infinity.

f(n)=o(g(n)) if and only if f(n)=O(g(n)) but g(n)≠O(f(n))

$$1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{3/4} \prec n \prec n\log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}$$

ω-notation

Definition: Let f(n) and g(n) be two functions from the set of natural numbers to the set of nonnegative real numbers. f(n) is said to be $\Omega(g(n))$ if there exists a natural number n0 and a constant c>0 such that

$\forall n \geq n0$, $f(n) > cg(n)$

Definition in Terms of Limits

Suppose $\lim_{n\to\infty} f(n)/g(n)$ **exists**.

- $\lim_{n\to\infty} f(n)/g(n) \neq \infty$ implies $f(n) = O(g(n))$.
- $\lim_{n\to\infty} f(n)/g(n) \neq 0$ implies $f(n) = \Omega(g(n))$.
- $\lim_{n\to\infty} f(n)/g(n) = c$ implies $f(n) = \Theta(g(n))$.
- $\lim_{n\to\infty} f(n)/g(n) = 0$ implies $f(n) = o(g(n))$.
- $\lim_{n\to\infty} f(n)/g(n) = \infty$ implies $f(n) = \omega(g(n))$.

A Helpful Analogy

- $f(n) = O(g(n))$ is similar to $f(n) \leq g(n)$.
- $f(n) = o(g(n))$ is similar to $f(n) < g(n)$.
- $f(n) = \Theta(g(n))$ is similar to $f(n) = g(n)$.
- $f(n) = \Omega(g(n))$ is similar to $f(n) \geq g(n)$.
- $f(n) = \omega(g(n))$ is similar to $f(n) > g(n)$.

复杂性函数的性质

A 传递性：

例1 $f(n)=\theta(g(n))\wedge g(n)=\theta(h(n))\Rightarrow f(n)=\theta(h(n))$

例2 $f(n)=\mathrm{O}(g(n))\wedge g(n)=\mathrm{O}(h(n))\Rightarrow f(n)=\mathrm{O}(h(n))$

例3 $f(n)=\Omega(g(n))\wedge g(n)=\Omega(h(n))\Rightarrow f(n)=\Omega(h(n))$

例4 $f(n)=o(g(n))\wedge g(n)=o(h(n))\Rightarrow f(n)=o(h(n))$

例5 $f(n)=w(g(n))\wedge g(n)=w(h(n))\Rightarrow f(n)=w(h(n))$.

B 自反性：

例2 $f(n)=\theta(f(n))$，

例3 $f(n)=\mathrm{O}(f(n))$，

例4 $f(n)=\Omega(f(n))$.

C 对称性

$f(n)=\theta(g(n))\ iff\ g(n)=\theta(f(n))$.

D 反对称性：

$f(n)=O(g(n))\ \ iff\ \ g(n)=\Omega(f(n))$

$f(n)=o(g(n))\ \ iff\ \ g(n)=w(f(n))$

*并非所有函数都是可比的，即对于函数 $f(n)$ 和 $g(n)$，可能 $f(n)\neq\mathrm{O}(g(n)),f(n)\neq\Omega(g(n))$.

1。O符号的加法准则

$$T(n) = T_1(n) + T_2(n) = O(f_1(n)) + O(f_2(n))$$
$$= O(\max(f_1(n), f_2(n)))$$

证明：根据定义，对于某些常数c1、n1、及c2、n2，由已知可得：

在 n ≥n1 时，T1(n) ≤ c1 f(n) 成立。

在 n ≥n2 时，T2(n) ≤ c2 g(n) 成立。

设n1和n2 之间的最大值为n0，即n0 = MAX(n1，n2).
那么，在 n ≥ n0；T1(n) ＋ T2(n) ≤ c1 f(n) ＋c2 g(n)。所以，T1(n) ＋ T2(n) ≤ (c1＋c2) MAX(f(n)，g(n))。 于是，命题得证。

2。O符号的乘法准则

$T(n) = T1(n) \times T2(n) = O(f1(n)) \times O(f2(n)) = O(f1(n) \times f2(n))$

证明：根据已知：在 n ≥n1 时，T1(n) ≤ c1 f(n) 成立。在 n ≥n2 时，T2(n) ≤ c2 g(n) 成立。

其中 c1、n1、及c2、n2 都是常数。

所以，在n ≥ MAX(n1，n2) 时，T1(n)×T2(n) ≤c1c2f(n)g(n)，所以：T1(n)×T2(n)是O(f(n)×g(n))的。

1.9 空间复杂性Space complexity

The work space cannot exceed the running time of an algorithm.

$$S(n)=O(T(n))$$

In many problems there is a time-space tradeoff: The more space we allocate for the algorithm the faster it runs, and vice versa.

增加空间，速度增加有限度;

减少空间，一定导致算法速度降低。

1.10 最优算法Optimal algorithm

In general, if we can prove that any algorithm to solve problem II must be $\Omega(f(n))$, then we call any algorithm to solve problem II in time $O(f(n))$ an optimal algorithm for problem II

基于比较的排序算法的下界(nlogn)

1.11How to estimate the running time of an algorithm

1. Counting the number of iterations
2. Counting the frequency of basic operations

   Definition: An elementary operation in an algorithm is called a basic if it is of highest frequency to within a constant factor among all other elementary operations
3. Using recurrence relations
4. 平摊分析Amortized analysis

Counting the Iterations

Algorithm 1.7 Count1
Input: $n = 2^k$, for some positive integer k.
Output: *count* = number of times Step 4 is executed.

1. $count \leftarrow 0$;
2. **while** $n \geq 1$
3. **for** $j \leftarrow 1$ **to** n
4. $count \leftarrow count + 1$
5. **end for**
6. $n \leftarrow n/2$
7. **end while**
8. **return** *count*

while is executed $k + 1$ times; **for** is executed $n, n/2, \ldots, 1$ times

$$\sum_{j=0}^{k} \frac{n}{2^j} = n \sum_{j=0}^{k} \frac{1}{2^j} = n(2 - \frac{1}{2^k}) = 2n - 1 = \Theta(n)$$

Counting the Iterations

Algorithm 1.8 Count2
Input: A positive integer n.
Output: $count$ = number of times Step 5 is executed.

1. $count \leftarrow 0$;
2. **for** $i \leftarrow 1$ **to** n
3. $\quad m \leftarrow \lfloor n/i \rfloor$
4. $\quad$ **for** $j \leftarrow 1$ **to** m
5. $\quad\quad count \leftarrow count + 1$
6. $\quad$ **end for**
7. **end for**
8. **return** $count$

The inner **for** is executed $n, \lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots, \lfloor n/n \rfloor$ times

$$\Theta(n \log n) = \sum_{i=1}^{n} (\frac{n}{i} - 1) \leq \sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \leq \sum_{i=1}^{n} \frac{n}{i} = \Theta(n \log n)$$

1.11.3 使用递推关系

在递归算法中，一个界定运行时间的函数常常以递推关系的形式给出，即一个函数的定义包含了函数本身，例如 $T(n)=2T(n/2)+n$。寻找递推式的解已经得到了很好地研究，甚至可以机械地得到它的解（见 2.8 节对递推关系的讨论）。推导出一个递推关系，它界定一个非递归算法中基本运算的数目是可能的。例如，在算法 BINARYSEARCH 中，如果令 $C(n)$为一个大小是 n 的实例中执行比较的次数，则可以用递推式表示算法所做的比较次数：

$$C(n)\leqslant\begin{cases}1 & \text{若 } n=1\\ C(\lfloor n/2\rfloor)+1 & \text{若 } n\geqslant 2\end{cases}$$

这个递推式的解简化成如下的和：

$$\begin{aligned}C(n)&\leqslant C(\lfloor n/2\rfloor)+1\\&=C(\lfloor\lfloor n/2\rfloor/2\rfloor)+1+1\\&=C(\lfloor n/4\rfloor)+1+1 \quad \text{（等式 2.3）}\\&\vdots\\&=\lfloor\log n\rfloor+1\end{aligned}$$

也就是 $C(n)\leqslant\lfloor\log n\rfloor+1$，因此 $C(n)=O(\log n)$。由于在算法 BINARYSEARCH 中元素比较运算是基本运算，因此它的时间复杂性是 $O(\log n)$。

1.12 最好情况、最坏情况、平均情况分析

* 有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。例如：

```
* Void bubble-sort(int a[]，int n)
*     for(i=n-1,change=TURE;i>1 && change;--i)
*     {
*          change=false;
*           for(j=0;j<i;++j)
*              if (a[j]>a[j+1]) {
*                  a[j] ←→a[j+1];
*                   change=TURE}
*        }
```

* 最好情况下元素交换次数：0次
* 最好情况下元素比较次数：n-1次

Average Case Analysis

Take Algorithm InsertionSort for instance. Two assumptions:

- $A[1..n]$ contains the numbers 1 through n.
- All $n!$ permutations are equally likely.

The number of comparisons for inserting element $A[i]$ in its proper position, say j, is *on average* the following

$$\frac{i-1}{i} + \sum_{j=2}^{i} \frac{i-j+1}{i} = \frac{i-1}{i} + \sum_{j=1}^{i-1} \frac{j}{i} = \frac{i}{2} - \frac{1}{i} + \frac{1}{2}$$

The *average* number of comparisons performed by Algorithm InsertionSort is

$$\sum_{i=2}^{n} (\frac{i}{2} - \frac{1}{i} + \frac{1}{2}) = \frac{n^2}{4} + \frac{3n}{4} - \sum_{i=1}^{n} \frac{1}{i}$$

1.13 Amortized analysis（平摊分析）

In amortized analysis, we average out the time taken by the operation throughout the execution of the algorithm, and refer to this average as the amortized running time of that operation.

Amortized analysis guarantees the average cost of the operation, and thus the algorithm, in the worst case.

平摊分析求出的是平均值：总时间 / 总次数
属于平均时间复杂度估算
与最坏情况下的总操作时间相同

双向链表，初始时由一个节点组成，输入X，如果X是奇数，将X添加到表中。如果X是偶数，将X添加到表上，然后移去表中所有在X之前的奇数元素

- for j<-1 to n
- x<-A[j]
- 将x加入表中
- if x为偶数then
- while pred(x)为奇数
- 删除pred(x)
- end while
- end if
- end for

O(n) 还是$O(n^2)$?

Input Size and Problem Instance

Suppose that the following integer

$$2^{1024} - 1$$

is a legitimate input of an algorithm. What is the *size* of the input?

Input Size and Problem Instance

Algorithm 1.9 FIRST
Input: A positive integer n and an array $A[1..n]$ with $A[j] = j$ for $1 \leq j \leq n$.
Output: $\sum_{j=1}^{n} A[j]$.

1. $sum \leftarrow 0$;
2. **for** $j \leftarrow 1$ **to** n
3. $\quad sum \leftarrow sum + A[j]$
4. **end for**
5. **return** sum

The input size is n. The time complexity is $O(n)$. It is linear time.

Input Size and Problem Instance

Algorithm 1.10 SECOND
Input: A positive integer n.
Output: $\sum_{j=1}^{n} j$.

1. $sum \leftarrow 0$;
2. **for** $j \leftarrow 1$ **to** n
3. $\quad sum \leftarrow sum + j$
4. **end for**
5. **return** sum

The input size is $k = \lfloor \log n \rfloor + 1$. The time complexity is $O(2^k)$. It is exponential time.

Review of this lecture

- What is an algorithm, a problem, an instance?
- Why study algorithm design and analysis?
- Asymptotic notations and their properties :例1.5-1.16
- Growth rate of functions: polynomials vs. exponentials
- 时间复杂度的分析方法(迭代\语句执行频度\递归\平摊)

- 1.5, 1.9, 1.13, 1.16, 1.17, 1.25, 1.31, 1.32, 1.33, 1.37
- 注： 红色题号为难度偏大的选做题目