算法分析与设计

赵川源

计算机学院

课程概述

- 计算机系统中的任何软件，都是按一个个特定的算法来予以实现的。算法性能的好坏，直接决定了所实现软件性能的优劣。
- 如何判定一个算法的性能、用什么方法来设计算法、所设计的算法需要多少运行时间、多少存储空间，在实现一个软件时，这些都是必须要予以解决的。
- 因此，算法分析与设计是计算机科学与技术的一个核心问题，也是大学计算机专业、软件专业等本科生及研究生的一门重要的专业基础课程。

课程介绍

- 体验算法思想，了解算法和程序设计在解决问题过程中的作用；
- 能从简单问题出发，设计解决问题的算法，并能使用一种程序设计语言编制程序，实现用算法解决问题。

课程目标

1. 掌握分治法、动态规划、贪婪法、回溯法、随机算法等经典算法，能够运用这些知识将实际的计算机工程问题进行建模。

2. 针对实际工程问题的多种解决方案，能够依据需求分析进行功能模块的设计，并利用编程工具进行实现。

3. 能够运用算法设计的基本理论和方法对实际工程问题进行分析，结合文献查阅，分析工程问题的解决方案。

4. 能够针对复杂问题，通过数据分析和解释，建立相应的解决方案，得到合理有效的结论。

课程安排

- **课程安排 （2.5学分，40课时）**
  - ✓理论课时：32学时
  - ✓实　验：8学时，2个实验项目
  - ✓考核方式：平时表现 （课堂表现+作业+笔记）
    课程实验 （实验报告）
    期末考试

课程内容

- **第一部分(1-2章)：基本概念和常用数学工具**
- **第二部分(3-9章)：基本理论和技术，包括排序、递归、分治、贪婪法、动态规划、回溯法、随机算法**
- **第三部分：实验 (8学时)**

课程内容

- 第一章　算法的基本概念
- 第二章　算法的复杂性分析
- 第三章　排序问题和离散集合的操作
- 第四章　递归和分治
- 第五章　贪婪法
- 第六章　动态规划
- 第七章　回溯
- 第九章　随机算法

教材与参考书

- 教材：郑宗汉、 郑晓明：算法设计与分析，清华大学出版社， 第3版.
- 参考书：

1. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms, The MIT Press, 第二版, 2002.
2. 卢开澄. 计算机算法导引（第二版）。清华大学出版社，2006.
3. 王晓东. 计算机算法设计与分析，电子工业出版社，2001

Journals

1. IEEE Transactions on Electronic Computers
2. IEEE Transactions on Software Engineering
3. IEEE Transactions on Data and Knowledge Engineering
4. Acta Informatica
5. SIAM Journal on Computing
6. Journal of Computer and System Sciences
7. Communication of the ACM
8. Journal of the ACM
9. BIT
10. Information and Control
11. ACM Computing Surveys
12. Mathematics of Computation
13. Information Processing Letters
14. Theoretical Computer Science

Conferences

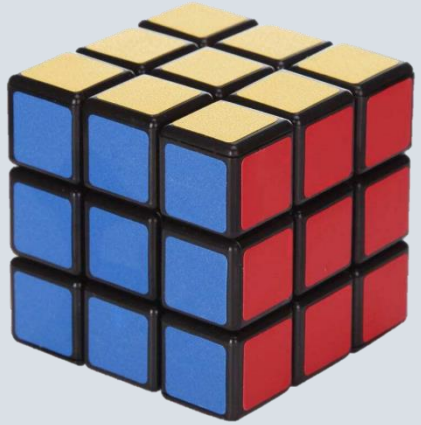
1. Annual ACM Symposium on Theory of Computing
2. Annual IEEE Symposium on Foundations of Computer Science
3. ACM Annual Computer Science Conference
4. Annual Symposium on Computational Geometry
5. ACM Symposium on Parallel Algorithms and Architectures

学习要求

- 深刻理解每一类算法的思想及其实现
- 能熟练运用所学知识解决实际工程问题
- 培养提高计算思维能力
- 提高编程能力

什么是算法？

- 算法：解决特定问题的一组有穷规则的集合，能够产生确定的结果。
- 算法的例子
  - 刘徽割圆术
  - 四则运算
  - 最小生成树
  - 快速排序



一个程序应包括两个方面的内容：

- 对数据的描述：数据结构(data structure)
- 对操作的描述：算法(algorithm)

著名计算机科学家沃思提出一个公式：

数据结构 + 算法 = 程序

完整的程序设计应该是：

数据结构＋算法＋程序设计方法＋语言工具

第1章 算法的基本概念

§1.1 算法的概念与特性

§1.2 简单算法举例

§1.3 算法的表示

§1.4 算法设计

§1.5 算法复杂性分析

§ 1.1 算法的概念与特性

- 定义：算法是解决某一特定问题的一组有穷规则的集合。
  - 算法是程序设计的基础，是计算机科学的核心；
  - 算法是指解决某一问题的运算序列。或者说算法是问题求解过程的运算描述，一个算法由有限条可完全机械地执行的、有确定结果的指令组成。

核心：算法是解决问题的办法和法则，算法必须能够让人一步一步照着执行。

- 一个算法应该具有以下五个特性：

1. 有限性

  - 一个算法须在执行有限个运算步后终止，每一步必须在有限时间内完成。
  - 实际应用中，算法的有限性应该包括执行时间的合理性。

  - **windows操作系统程序是算法吗？**

  ✓ 是一个在无限循环中执行的程序，随时对用户的输入进行响应；不是一个算法。

2. 确定性

算法的每一步骤都有精确的定义，要执行的每一个动作都是清晰的、无歧义的。

例如：计算分段函数 $f(x)=\begin{cases}1 & x>100\\ 0 & x<10\end{cases}$

算法描述：输入变量*x*,

- ✓若*x*大于100的数，输出1;
- ✓若*x*小于10的数，输出0.
- * 输入10<=*x*<=100，则算法在异常情况下，执行结果是不确定的。

3. 可行性

- 算法中的每个步骤是能实现的（如 $x/0$，负数开方）
- 算法的执行结果达到预期目的，正确、有效。

4. 输入

有0个或多个输入项

5. 输出

有1个或多个输出项

算法和程序的区别

- 程序是算法用某种程序设计语言的具体实现。
- 程序可以不满足算法有限性的性质。
- 操作系统是一个在无限循环中执行的程序，因而不是一个算法。
- 操作系统的各种任务可看成是单独的问题，每一个问题由操作系统中的一个子程序通过特定的算法来实现。该子程序得到输出结果后便终止。

例1：求两个整数的最大公因子的欧几里得算法

算法1.1：欧几里得算法

1. 用*n*去除*m*，将余数赋给 *r*；

2. 如果*r*=0, 返回*n*的值作为结果，过程结束;否则执行第3步；

3. 将*n*的值赋给*m*，将*r*的值赋给*n*；

4. 返回第一步。

最大公约数问题：求两个正整数m和n的最大公约数

- 设计：

输入：　正整数*m*和*n*
输出：　*m*和n的最大公约数
第一步：求余数r ← m%n
第二步：if　r == 0?
　　…　　yes　终止(n为答案)
　　…　　no　执行第三步。
第三步：m ← n, n ← *r*,
第四步：返回第一步。

可行性：

算法中所有待实现的运算，都是基本

特性：

注意：
有限性
一个步
有限，
花费的
如果
成无
成，
了。
能确定地执行。

（1）输入
（2）输出
（3）有限性
（4）确定性
（5）可行性

步骤
步骤所
的。
展开
地完
行的

§ 1.3 算法的表示

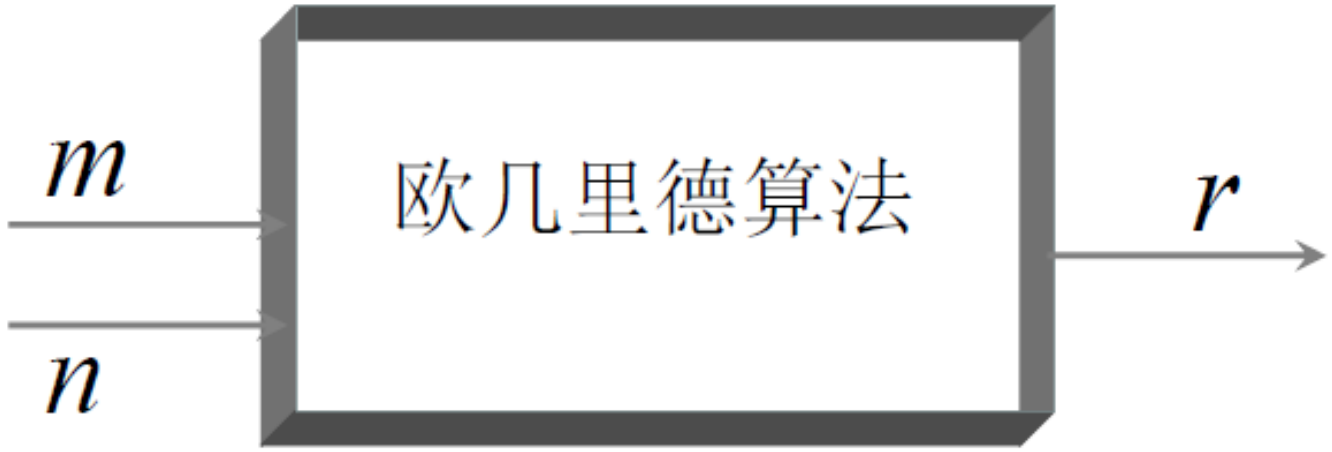
可以用不同的方式表示算法，常用的有：

- 自然语言
- 流程图
- 程序设计语言
- 伪代码

1. 自然语言

- 自然语言就是人们日常使用的语言，可以是汉语或英语或其它语言。用自然语言表示通俗易懂，但文字冗长，容易出现“歧义性”。
- 自然语言表示的含义往往不大严格，要根据上下文才能判断其正确含义，描述包含分支和循环的算法时也不很方便。
- 因此，除了那些很简单的问题外，一般不用自然语言描述算法。

例：欧几里德算法——辗转相除法求两个自然数 m 和 n 的最大公约数



欧几里德算法

① 输入m 和n；

② 求m除以n的余数r；

③ 若r等于0，则n为最大公约数，算法结束；

否则执行第④步；

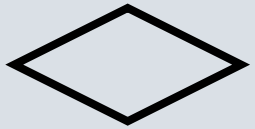
④ 将n的值放在m中，将r的值放在n中；

⑤ 重新执行第②步。

2. 流程图

美国国家标准化协会ANSI(American National Standard Institute)规定了一些常用的流程图符号：

起止框

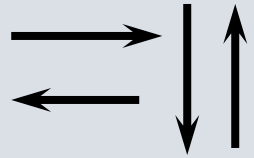


判断框

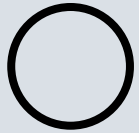
处理框

输入/输出框

注释框

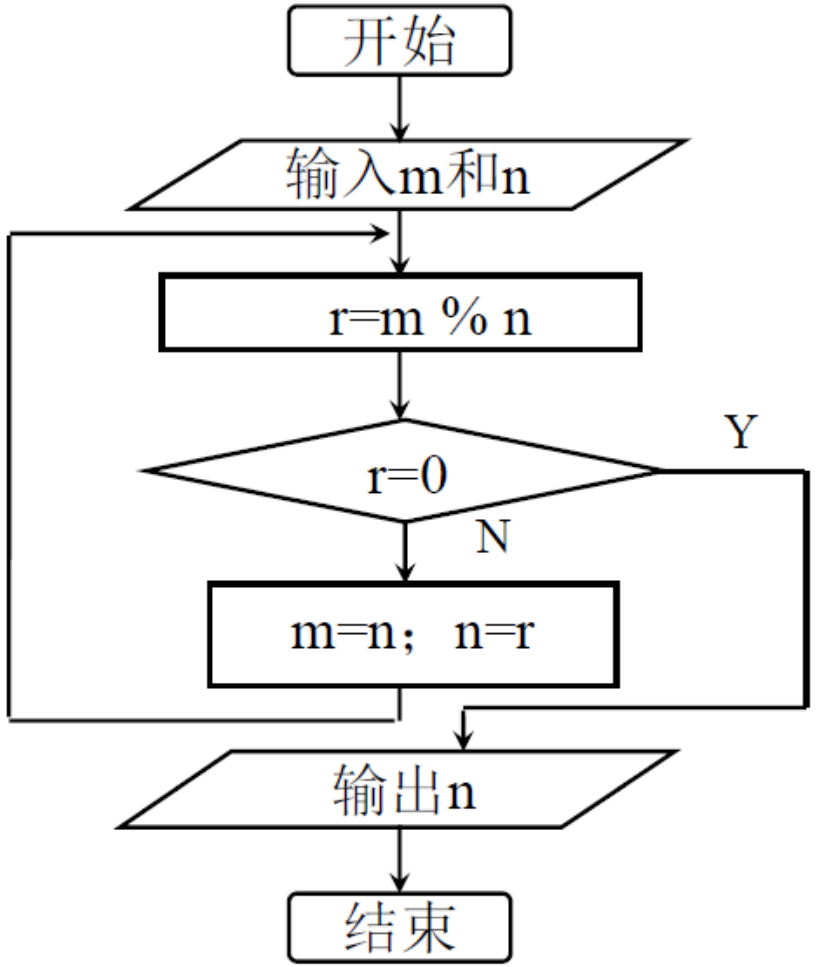


流向线



连接点

欧几里德算法



流程图

优点：流程直观

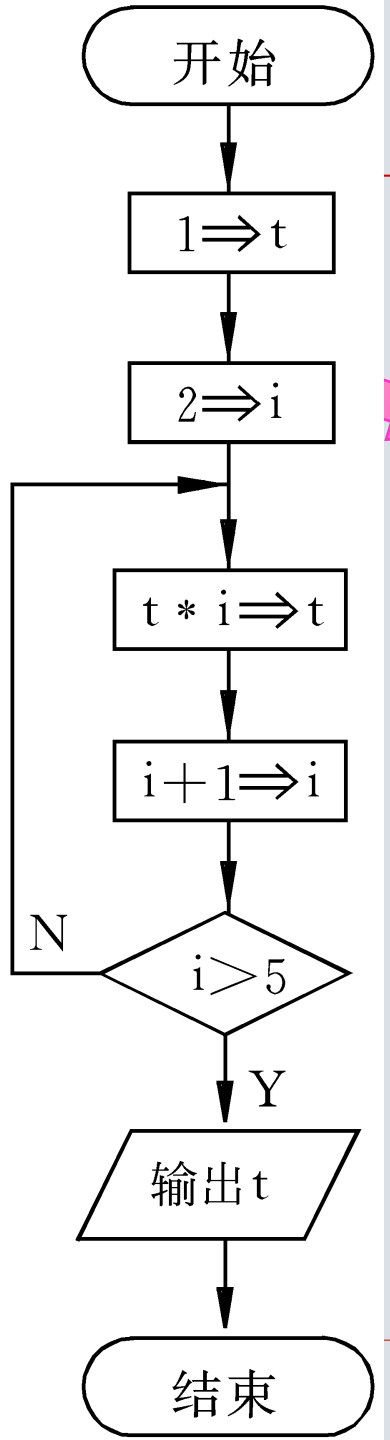
缺点：缺少严密性、灵活性

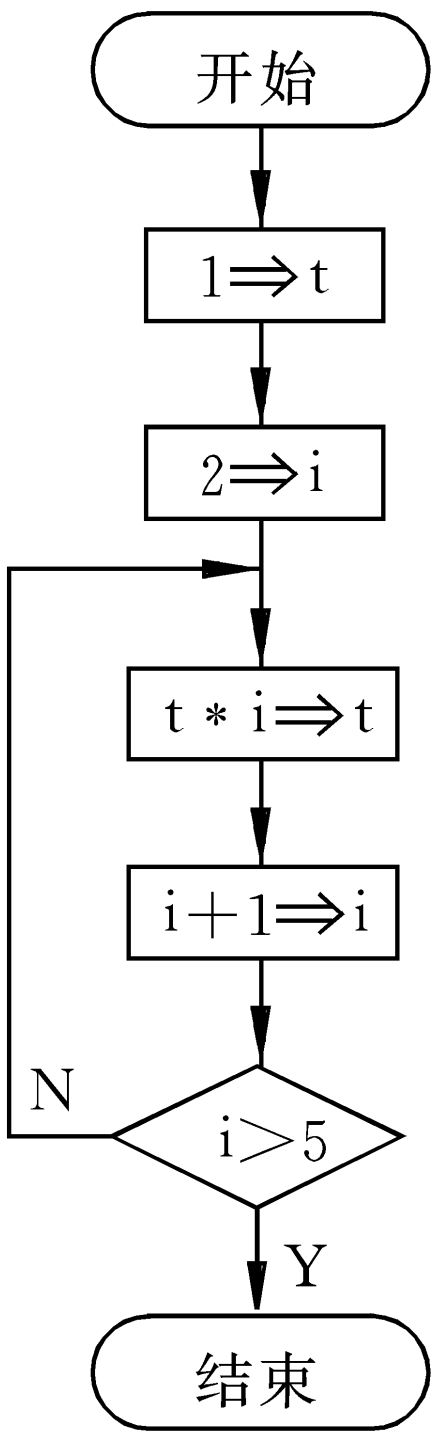
使用方法：描述简单算法

注意事项：注意抽象层次

例2 将求5!的算法用流程图表示

如果需要将最后结果打印出来，可在菱形框的下面加一个输出框。





小结：

- 流程图是表示算法的较好的工具。一个流程图包括以下几部分：

(1) 表示相应操作的框；

(2) 带箭头的流程线；

(3) 框内外必要的文字说明。

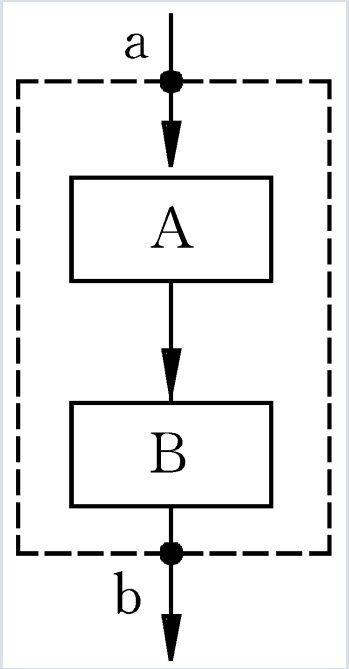
□三种基本结构

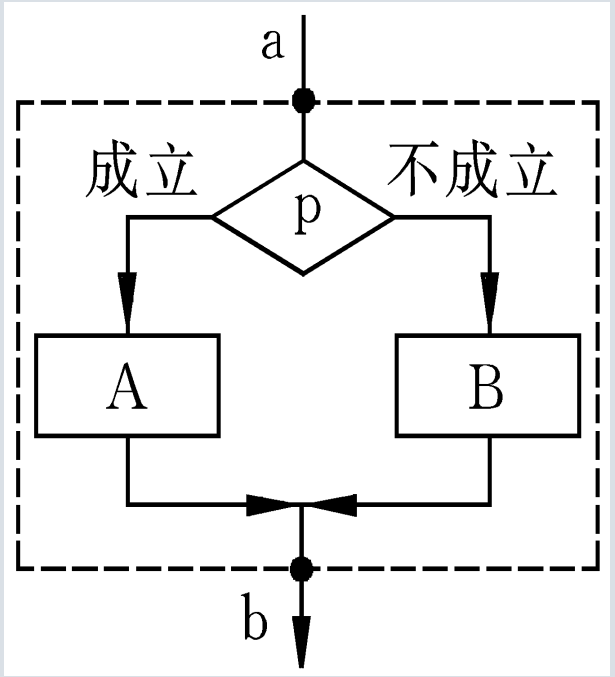
Bohra和Jacopini提出了以下三种基本结构：

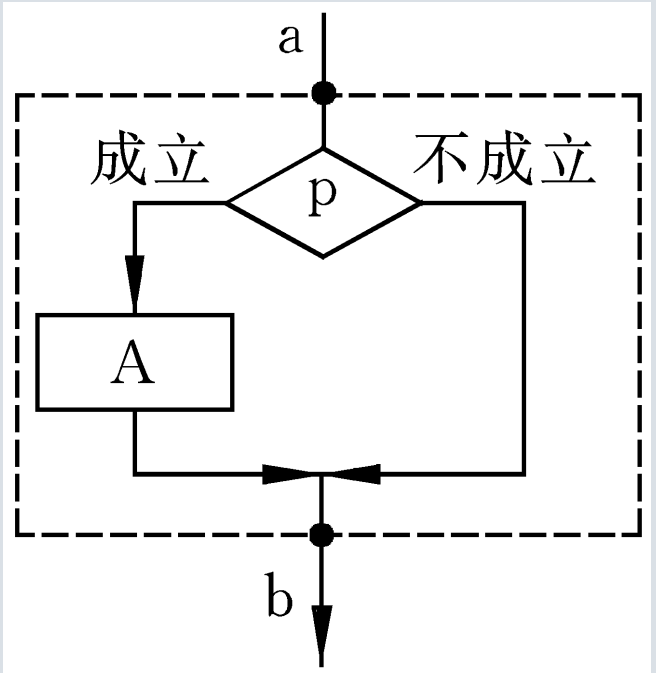
顺序结构、选择结构、循环结构

用这三种基本结构作为表示一个良好算法的基本单元。

三种基本结构的图示：

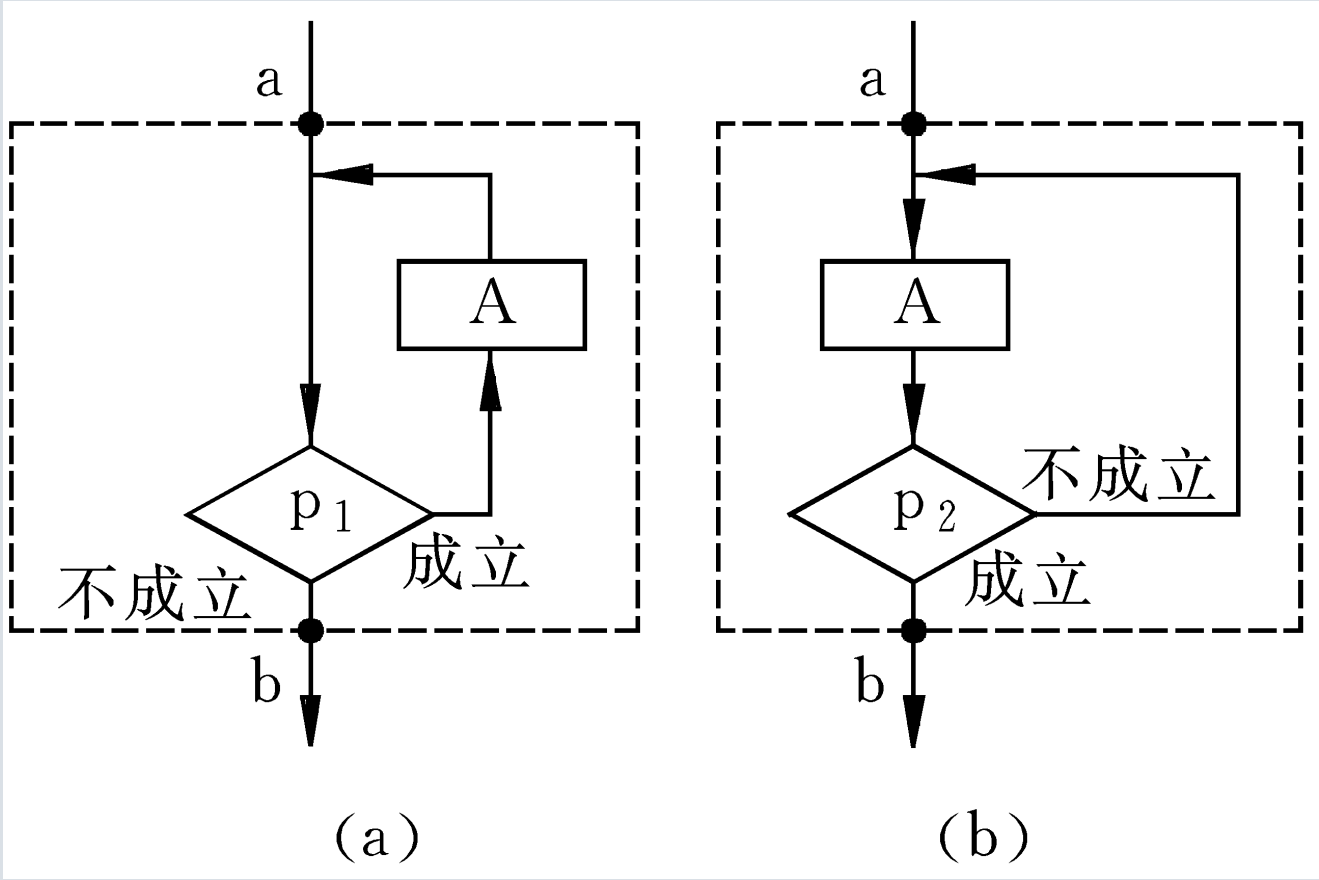






顺序结构　　选择结构

循环结构的图示：



当型(While型)循环结构　直到型(Until型)循环

三种基本结构的共同特点：

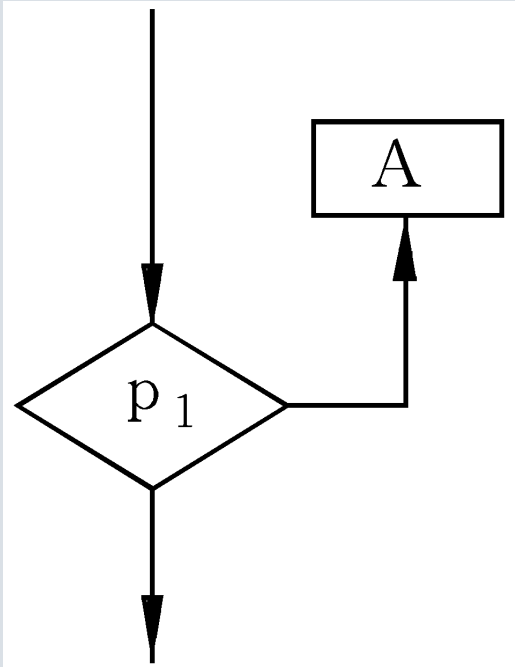
(1)只有一个入口；

(2)只有一个出口；（请注意：一个菱形判断框有两个出口，而一个选择结构只有一个出口。不要将菱形框的出口和选择结构的出口混淆）

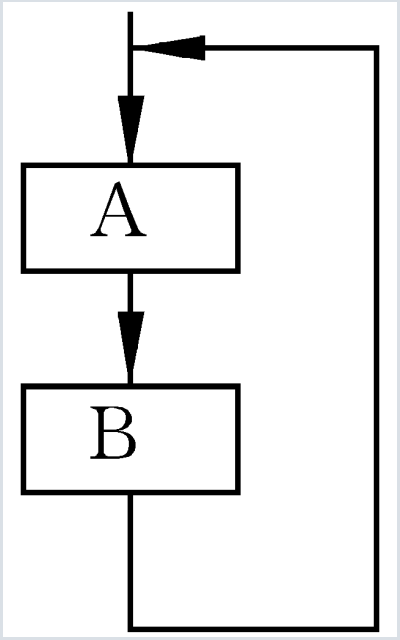
(3)结构内的每一部分都有机会被执行到；

(4)结构内不存在“死循环”(无终止的循环)。

不正确的流程表示：



图中没有一条从入口到出口的路径通过A框。



流程内的死循环

小结：

- 由三种基本结构顺序组成的算法结构，可以解决任何复杂的问题。
- 由基本结构所构成的算法属于“结构化”的算法，它不存在无规律的转向，只在本基本结构内才允许存在分支和向前或向后的跳转。

3. 程序设计语言

- 概念：用计算机实现算法。计算机是无法识别流程图和伪代码的。只有用计算机语言编写的程序才能被计算机执行。因此在用流程图或伪代码描述出一个算法后，还要将它转换成计算机语言程序。
- 特点：用计算机语言表示算法必须严格遵循所用语言的语法规则，这是和伪代码不同的。
- 用处：要完成一件工作，包括设计算法和实现算法两个部分。设计算法的目的是为了实现算法。

欧几里德算法

```
#include <iostream.h>
int CommonFactor(int m, int n)
{
   int r=m % n;
   while (r!=0)
  {
     m=n;
     n=r;
     r=m % n;
  }
  return n;
}
void main( )
{ cout<<CommonFactor(63, 54)<<endl;  }
```

例3：将例2表示的算法（求5!）用C语言表示。

```
#include <stdio.h>
void main( )
 {int i,t;
  t=1;
  i=2;
  while(i<=5)
    {t=t*i;
     i=i+1;
    }
  printf("%d\n",t);
 }
```

- 应当强调说明：写出了C程序，仍然只是描述了算法，并未实现算法。只有运行程序才是实现算法。应该说，用计算机语言表示的算法是计算机能够执行的算法。

4. 伪代码

- 概念：伪代码是用介于自然语言和计算机语言之间的文字和符号来描述算法。
- 特点：它如同一篇文章一样，自上而下地写下来。每一行(或几行)表示一个基本操作。它不用图形符号，因此书写方便、格式紧凑，也比较好懂，也便于向计算机语言算法(即程序)过渡。
- 用处：适用于设计过程中需要反复修改时的流程描述。

欧几里德算法

1. r = m % n;
2. 循环直到 r 等于0
   2.1 m = n;
   2.2 n = r;
   2.3 r = m % n;
3. 输出 n ;

例4：“打印x的绝对值”的算法可以用伪代码表示为：

```
IF x is positive THEN
    print x
ELSE
    print -x
```

也可以用汉字伪代码表示：

```
若 x为正
    打印 x
否则
    打印 -x
```

也可以中英文混用，如：

```
IF x 为正
    print x
ELSE
    print -x
```

例5： 求5!。用伪代码表示

算法：

也可以写成以下形式：

```
BEGIN｛算法开始｝
    1→t
    2 → i
    while i≤5
     {t×i →t
      i+1 → i }
    print t
  END｛算法结束｝
```

开始

置t的初值为1

置i的初值为2

当i<=5，执行下面操作：

使t=t×i

使i=i+1

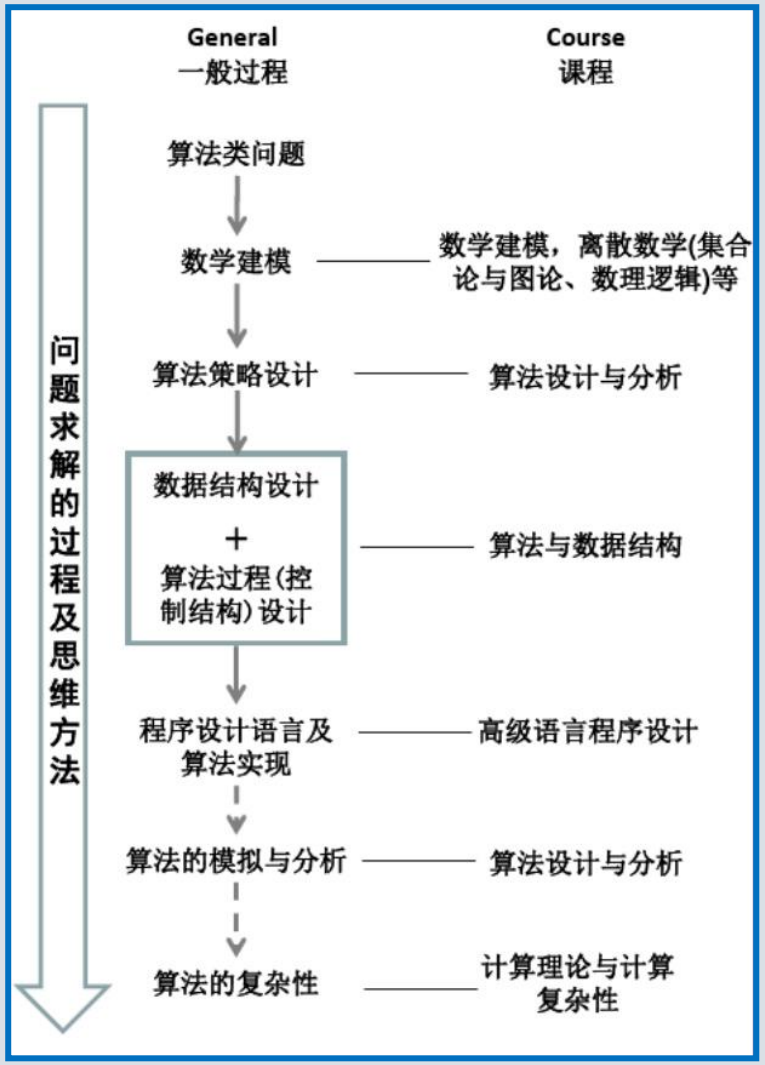
｛循环体到此结束｝

输出t的值

结束

1.4 算法设计

- 问题抽象
- 数学建模
- 算法策略设计
- 算法的数据结构和控制结构设计
- 算法的实现
- 算法的分析



例6 百鸡问题：

- 公鸡每只5元、母鸡每只3元、小鸡3只1元，用100元钱买100只鸡，求公鸡、母鸡、小鸡的只数。
- 令*a*为公鸡数， 为*b*母鸡数，*c* 为小鸡数，则：

$$\begin{cases} a+b+c=100 & (1) \\ 5a+3b+c/3=100 & (2) \\ c\%3=0 & (3) \end{cases}$$

算法1.2 百鸡问题的穷举法

- 输入：所购买的3种鸡的总数目n
- 输出：满足问题的解的数目k,公鸡,母鸡,小鸡的只数g[],m[],s[]

```
1. void chicken_question(int n, int &k, int g[], int m[], int s[])
2. {
3.     int  a,b,c;
4.     k = 0;
5.     for (a = 0; a <= n; a++) {                                  n+1
6.         for ( b = 0; b <= n; b++) {                             (n+1)^2
7.             for (c = 0; c <= n; c++) {                          (n+1)^3
8.                 if ((a + b + c == n) && (5 * a + 3 * b + c / 3 == n) && (c%3 == 0)) {
9.                     g[k] = a;
10.                    m[k] = b;
11.                    s[k] = c;
12.                    k++;
13.                }
14.            }
15.        }
16.    }
17. }
```

分析发现：只能买到n/5只公鸡，n/3只母鸡，将算法进行改进。

n=100时，内循环次数大于100万次

算法1.3 百鸡问题的穷举法改进

- 输入：所购买的3种鸡的总数目n
- 输出：满足问题的解的数目k,公鸡,母鸡,小鸡的只数g[],m[],s[]

$(\frac{n}{5}+1)$

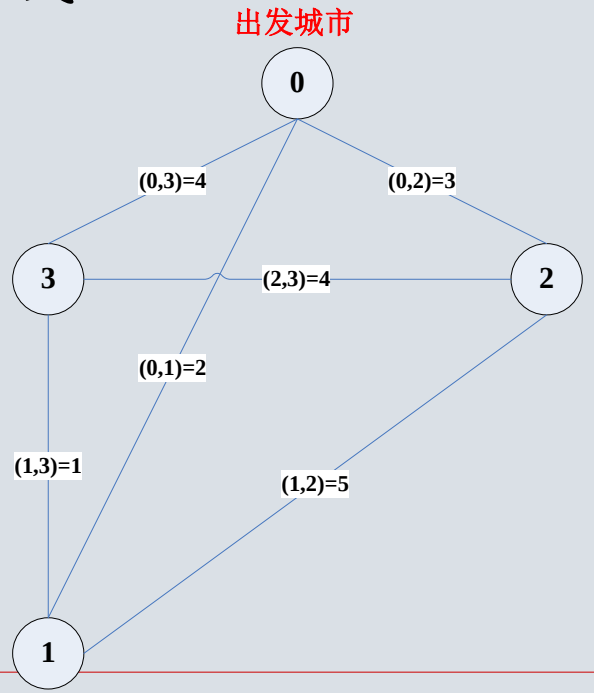
$(\frac{n}{5}+1)(\frac{n}{3}+1)$

```
1. void chicken_question_2(int n, int &k, int g[], int m[], int s[])
2. {
3.     int  a,b,c;
4.     k = 0;
5.     i = n / 5;
6.     j = n / 3;
7.     for (a = 0; a <= i; a++)
8.     for ( b = 0; b <= j; b++)    {
9.         c = n – a - b;     // 原来为 for (c = 0; c <= n; c++)
10.         if ((a + b + c == n) && (5 * a + 3 * b + c / 3 == n) && (c%3 == 0)) {
11.             g[k] = a;
12.             m[k] = b;
13.             s[k] = c;
14.             k++;
15.         }
16. }
```

n=100时，内循环次数大于714次

例7 货郎担问题：

- 售货员到若干个城市去售货，每个城市仅经过一次，最后回到出发点。已知各个城市之间的距离，求一个总路程最短的路线。



- 最短路径的哈密尔顿回路问题，数据结构是无向加权图G = <V, E>，V是顶点集合，E是距离集合。

算法1.4 货郎担问题的穷举法算法

- **输入**：城市个数n，费用(距离)矩阵c[][]
- **输出**：旅行路线t，最小费用min

```
1. void salesman_problem(int n, float &min, int t[], float c[][])
2. {
3.     int p[n], i = 1;
4.     float cost;
5.     min = MAX_FLOAT_NUM;
6.     while (i <= n!) {
7.         产生n个城市的第i个排列于p;
8.         cost =  路线p的费用;
9.         if (cost < min) {
10.            把数组p的内容复制到数组t;
11.            min = cost;
12.        }
13.        i++;
14.    } }
```

n!

货郎担问题的执行时间和问题规模的关系

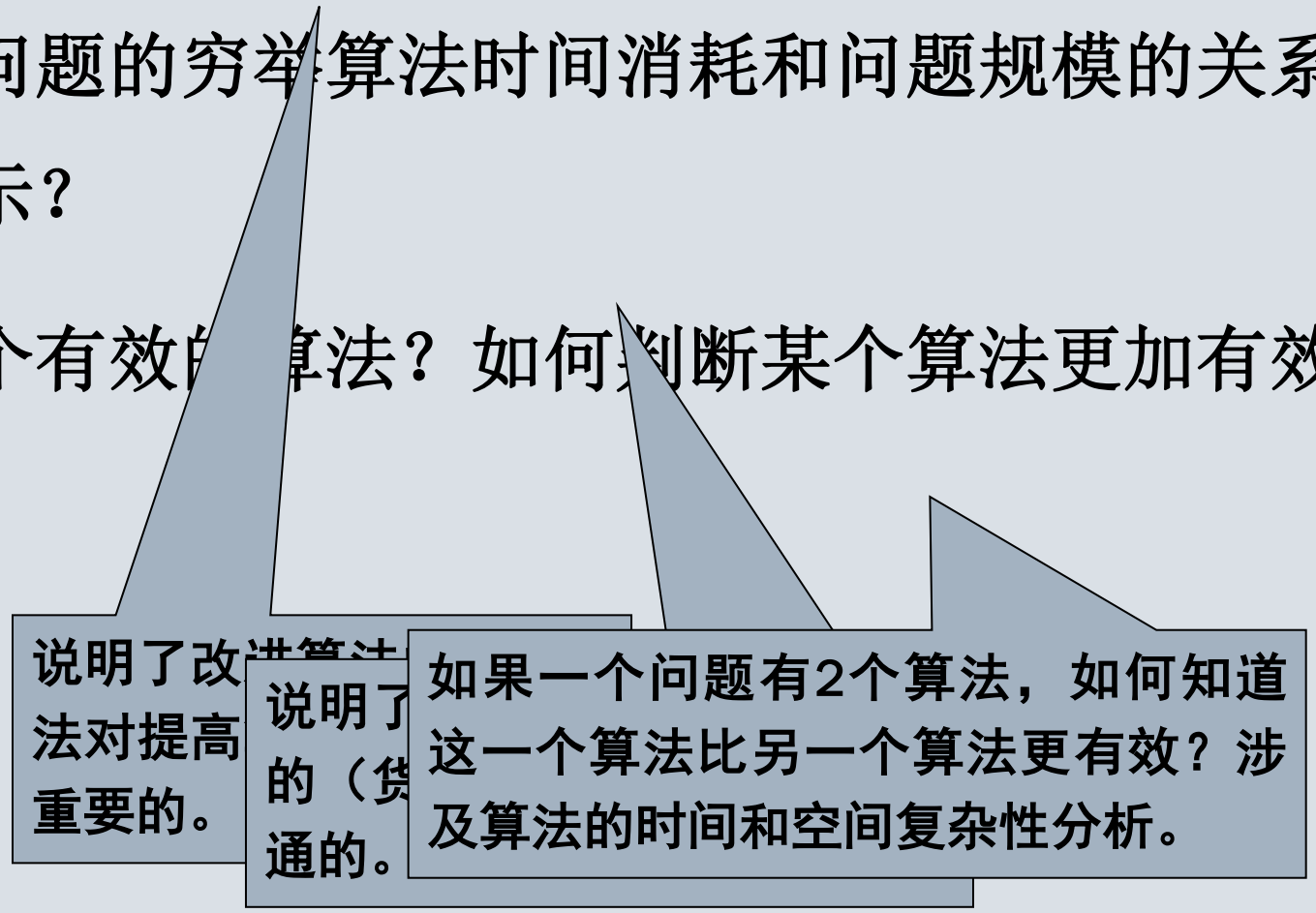
表 1.1　算法 1.4 的执行时间随 n 的增长而增长的情况

n	$n!$	n	$n!$	n	$n!$	n	$n!$
5	120μs	9	362ms	13	1.72h	17	11.27year
6	720μs	10	3.62s	14	24h	18	203year
7	5.04ms	11	39.9s	15	15day	19	3857year
8	40.3ms	12	479.0s	16	242day	20	77146year

（假定循环体每执行一次，需要1μs时间）

思考

- 从百鸡问题的穷举法改进，可以得到什么启示？
- 从货郎担问题的穷举算法时间消耗和问题规模的关系，可以得到什么启示？
- 什么是一个有效的算法？如何判断某个算法更加有效？



§ 1.5 算法复杂性分析

- 算法复杂性的高低体现运行该算法所需计算机资源的多少。
  - 算法的复杂性越高，所需的计算机资源越多；
  - 反之，算法的复杂性越低，所需的计算机资源越少。
- 计算机资源，最重要的是时间资源与空间资源。
- 需要计算机时间资源的量称为时间复杂度，需要计算机空间资源的量成为空间复杂度。
- 算法分析是指对算法的执行时间与所需空间的估算，定量给出运行算法所需的时间数量级与空间数量级。

考虑空间复杂性的理由：

- 在多用户系统中运行时，需指明分给该程序的内存大小；
- 需预先知道计算机系统是否有足够的内存来运行该程序；
- 一个问题可能有若干个不同的内存解决方案，从中择取；
- 用空间复杂性估计一个程序可能解决的问题的最大规模。

考虑时间复杂性的理由

- 某些计算机用户需要提供程序运行时间的上限（用户可接受的）
- 所开发的程序需要提供一个满意的实时反应

§ 1.5.1 算法的时间复杂性

- 一个算法的时间复杂度是指算法运行所需的时间。
- 一个算法的运行时间取决于算法所需执行的语句（运算）的多少。算法的时间复杂度通常用该算法执行的总的语句（运算）的数量级来决定。
- 就算法分析而言，一条语句的数量级即执行它的频数，一个算法的数量级是指它所有语句执行频数之和。

1. 算法的输入规模和运行时间的阶

两点事实：

1. 算法执行时间随问题规模的增大而增大，增长速度随不同的算法而不同；
2. 没有一种方法能准确地计算算法的执行时间。

初等操作：

- ✓ 所有操作数都具有相同的固定字长
- ✓ 所有操作的时间花费都是一个常数时间间隔
- ✓ 算数运算、比较和逻辑运算、赋值运算等

算法1.2 百鸡问题的穷举法

- 输入：所购买的3种鸡的总数目n
- 输出：满足问题的解的数目k,公鸡,母鸡,小鸡的只数g[],m[],s[]

```
1. void chicken_question(int n, int &k, int g[], int m[], int s[])
2. {
3.     int  a,b,c;
4.     k = 0;                                              1
5.     for (a = 0; a <= n; a++) {                          1+2(n+1)
6.         for ( b = 0; b <= n; b++) {                     (n+1)+2(n+1)²
7.             for (c = 0; c <= n; c++) {                  (n+1)²+2(n+1)³
8.14(n+1)³ if ((a + b + c == n) && (5 * a + 3 * b + c / 3 == n) && (c%3 == 0)) {
9.                     g[k] = a;                           ≤4(n+1)³
10.                    m[k] = b;
11.                    s[k] = c;
12.                    k++;
13.                }
14.            }
15.        }
16.    }
17. }
```

算法1.3 百鸡问题的穷举法改进

- 输入：所购买的3种鸡的总数目n
- 输出：满足问题的解的数目k,公鸡,母鸡,小鸡的只数g[],m[],s[]

```
1. void chicken_question_2(int n, int &k, int g[], int m[], int s[])
2. {
3.     int  a,b,c;
4.     k = 0;                                  1
5.     i = n / 5;                              2
6.     j = n / 3;                              2
7.     for (a = 0; a <= i; a++)                1+2(n/5+1)
8.     for ( b = 0; b <= j; b++)    {          (n/5+1)+2(n/5+1)(n/3+1)
9.         c = n – a - b;                      3(n/5+1)(n/3+1)
10.          if ((5 * a + 3 * b + c / 3 == n) && (c%3 == 0)) {   10(n/5+1)(n/3+1)
11.              g[k] = a;
12.              m[k] = b;                      ≤4(n/5+1)(n/3+1)
13.              s[k] = c;
14.              k++;
15.         }
16. }
```

算法1.2 百鸡问题的穷举法

$$T_1(n) \le 1+1+2(n+1)+(n+1)+2(n+1)^2+(n+1)^2+2(n+1)^3$$
$$+14(n+1)^3+4(n+1)^3$$
$$=20n^3+63n^2+69n+28$$

算法1.3 百鸡问题的穷举法改进

$$T2(n) \le 1+2+2+1+2(n/5+1)+(n/5+1)+2(n/5+1)(n/3+1)$$
$$+(3+10+4)(n+1)^2+ 2(n/5+1)(n/3+1)$$
$$=19/15n^2+161/15n+28$$

- 为什么要引入阶？
  - 首先，运行时间是相对的，而不是绝对的。
  - 其次，希望使用度量算法运行时间的准则不依赖于技术的进步。
  - 最后，不仅仅关注小规模输入下的，而且还关注在大规模输入下的情形。
    - ✓ 同一个问题往往可以由不同的算法解决，但是算法不同，效率也不同。

算法1.2 百鸡问题的穷举法

$$T_1(n)\le 1+1+2(n+1)+(n+1)+2(n+1)^2+(n+1)^2+2(n+1)^3$$
$$+14(n+1)^3+4(n+1)^3$$
$$=20n^3+63n^2+69n+28$$

$T_1^*(n) \approx c_1n^3$, $c_1 > 0$, **$T_1^*(n)$ 的阶是n^3**

算法1.3 百鸡问题的穷举法改进

$$T2(n)\le 1+2+2+1+2(n/5+1)+(n/5+1)+2(n/5+1)(n/3+1)$$
$$+(3+10+4)(n+1)^2+ 2(n/5+1)(n/3+1)$$
$$=19/15n^2+161/15n+28$$

$T_2^*(n) \approx c_2n^2$, $c_2 > 0$, **$T_2^*(n)$ 的阶是n^2**

- 当一个问题的输入规模很大时，算法的结构又很复杂时，采用精确分析就显得过于繁琐；
- 为降低算法分析的代价，同时又保证估算的精确度，引入一个简化的计算模型来评估算法的开销，称为渐进分析；
- 渐进分析是对问题的规模充分大的算法开销的估算。

把$T_1^*(n)$和$T_2^*(n)$进行比较，有：

$$\frac{T_1^*(n)}{T_2^*(n)}=\frac{c_1}{c_2}n$$

定义：设算法的执行时间为$T(n)$，如果存在$T^*(n)$，使得

$$\lim_{n\to\infty}\frac{T(n)-T^*(n)}{T(n)}=0$$

就称$T^*(n)$为算法的渐进时间复杂性。

- 在数学上, T(*n*)与$T^*(n)$有相同的最高阶项。可取$T^*(n)$为略去T(*n*)的低阶项后剩余的主项。

  例如 $T(n)=3n^2+4n\log n+7$，则$T^*(n)$可以是$3n^2$

- 当*n*充分大时，用$T^*(n)$代替T(*n*)作为算法复杂性的度量，以简化分析。

- 比较两个算法时，如果他们的阶不同，就可以分出效率高低。故此时只需关心$T^*(n)$最高的阶即可。可忽略最高项系数或最低项。

求渐近复杂性

- $T(n)=3n^2+10n$ ——$T^*(n)=3n^2$
- $T(n)=n^2/10+2^n$ ——$T^*(n)=2^n$
- $T(n)=21+1/n$ ——$T^*(n)=21$

- 当$n \to \infty$时，T(n)渐近于T*(n)，可用T*(n)来替代T(n)作为算法A在$n \to \infty$时的复杂性的度量。
  - ✓ 一种简化
- 渐近复杂性分析只要关心T*(n)的阶就够了，不必关心其中的常数因子。
  - ✓ 进一步简化
- 常用的时间复杂性的阶为：
  - ✓ $\log n$, n, $n\log n$, n^2, n^3, 2^n ,$n!$

表 2-3　各种类别的算法以及它们在一台每秒运行 1 百万次的计算机上的执行时间(1 秒=10^6 微秒=10^3 毫秒)

类　别	复 杂 度	操作次数和执行时间(1 条指令/微秒)					
N		10		10^2		10^3	
常数	O(1)	1	1 微秒	1	1 微秒	1	1 微秒
对数	O(lg n)	3.32	3 微秒	6.64	7 微秒	9.97	10 微秒
线性	O(n)	10	10 微秒	10^2	100 微秒	10^3	1 毫秒
O(n lg n)	O(n lg n)	33.2	33 微秒	664	664 微秒	9970	10 毫秒
平方	O(n^2)	10^2	100 微秒	10^4	10 毫秒	10^6	1 秒
立方	O(n^3)	10^3	1 毫秒	10^6	1 秒	10^9	16.7 分钟
指数	O(2^n)	1024	10 毫秒	10^{30}	3.17*10^7 年	10^{301}	
N		10^4		10^5		10^6	
常数	O(1)	1	1 微秒	1	1 微秒	1	1 微秒
对数	O(lg n)	13.3	13 微秒	16.6	7 微秒	19.93	20 微秒
线性	O(n)	10^4	10 毫秒	10^5	0.1 秒	10^6	1 微秒
O(n lg n)	O(n lg n)	133*1$0^3$	133 毫秒	166*10^4	1.6 秒	199.3*10^5	20 秒
平方	O(n^2)	10^8	1.7 分钟	10^{10}	16.7 分钟	10^{12}	11.6 天
立方	O(n^3)	10^{12}	11.6 天	10^{15}	31.7 年	10^{18}	31,709 年
指数		10^{3010}		10^{30103}		10^{301030}	

结论

- 算法的时间复杂性量级不同，在相同的时间内所能解决的问题的规模不同。
- 选一个好的算法比计算机的速度提高意义更大。
- 把时间复杂性为多项式量级的算法称为有效算法。

- 例：求下列函数的渐进表达式:

（1）$3n^2+10n$

（2）$21+1/n$

- 解：（1） $\lim_{n\to\infty}\frac{(3n^2+10n)-3n^2}{3n^2+10n}=0;$

由渐进表达式的定义可知：$3n^2$是$3n^2+10n$的渐进表达式。

（2） $\frac{21+\frac{1}{n}-21}{21+\frac{1}{n}}\to 0, n\to\infty$

由渐进表达式的定义可知：21是$21+1/n$的渐进表达式。

- ✓ 简化算法时间复杂性分析，只要考察当问题的规模充分大时，算法复杂性在渐近意义下的阶。为了进行渐近性分析，引入以下符号：上界、下界和准确界。

2. 运行时间的上界O

设$f(n)$和 $g\ (n)$ 是定义在正整数集上的正函数,

若存在正常数c和自然数n_0 , 使得当 $n \geq n_0$ 时, 有$f(n) \leq cg\ (n)$, 则称函数$f(n)$在n充分大时有上界, 且$cg\ (n)$是它一个上界，记为$f(n)=O(g\ (n))$, 也称$f(n)$的阶不高于$g\ (n)$的阶。

$O(g(n))$ 的运行时间，是指当n足够大时，该算法的实际运行时间不会超过的某个常数倍时间。

- 算法的执行时间等于算法中各语句执行时间的总和，而某个语句的执行时间等于该语句执行一次所需时间与执行次数的乘积；
- 算法的执行时间通常表示成数量级的形式：$f(n) = O(g(n))$
  - ✓ 含义：当问题规模n足够大时，算法的执行时间$f(n)$和函数$g(n)$成正比，或者说，存在正常数c和n_0，当$n \geq n_0$时，有$|f(n)| \leq c|g(n)|$。

例10: $n \geq n_0$ 时，$f(n) \leq cg(n)$

设$f(n) = n^2 + n$，则

$f(n) = O(n^2)$，取$c = 2, n_0 = 1$即可

$f(n) = O(n^3)$，取$c = 1, n_0 = 2$即可

1. $f(n) = O(g(n))$，$f(n)$的阶不高于$g(n)$的阶；
2. 可能存在多个正数c，只要指出一个即可；
3. 对前面有限个n值可以不满足不等式；
4. 常函数可以写作$O(1)$.

三点注意:

1. 用来作比较的函数 $g(n)$应该尽量接近 $f(n)$.

例如： $3n+2=O(n^2)$ 松散的界限；

$3n+2=O(n)$ 较好的界限

2. 不要产生错误界限。

例如 $n^2+100n+6$，当 $n<3$ 时，$n^2+100n+6<106n$，不能由此就认为 $n^2+100n+6=O(n)$.

3. $f(n)=O(g(n))$不能写成 $g(n)=O(f(n))$

因为两者并不等价。实际上，这里的等号并不是通常相等的含义。

3. 运行时间的下界Ω

若存在正常数c和自然数n_0使得当 $n \geq n_0$ 时，有 $f(n) \geq c\ g(n)$, 则称函数$f(n)$在n充分大时有**下限**，且 $g(n)$是它的一个下界，记为$f(n) = \Omega(g(n))$，也称$f(n)$的阶不低于 $g(n)$的阶。

$n \geq n_0$ 时，$f(n) \geq cg(n)$

例11:

设$f(n) = n^2 + n$，则

$f(n) = \Omega(n^2)$，取$c = 1, n_0 = 1$即可

$f(n) = \Omega(100n)$，取$c = 1/100, n_0 = 1$即可

1.$f(n) = \Omega(g(n))$，$f(n)$的阶不低于$g(n)$的阶；

2.可能存在多个正数c，只要指出一个即可；

3. 对前面有限个n值可以不满足上述不等式。

- 对于一个问题和任意给定的充分大的规模n，下界在该问题的所有算法或某类算法的复杂性中取，这时得到的相应下界，称之为问题的下界或某类算法的下界。
- Ω常常与O配合以证明某问题的一个特定算法是该问题的最优算法或该问题的某算法类中的最优算法。
- 一般来说，说明一个算法好，需要估计算法时间复杂性的上界；说明一个算法坏，需要估计算法时间复杂性的下界。
- 在O和Ω的基础上可定义Θ

4. 运行时间的准确界Θ

$f(n) = \Theta(g(n))$等价于 $f(n)=O(g(n))$ 且 $f(n)=\Omega(g(n))$，

称函数$f(n)$与$g(n)$ 同阶。

为了助于理解，

O类似于≤　　Ω类似于≥　　Θ类似于=

例12:

设$f(n)=n^2+n$，　$g(n)=100n^2$，那么有

$$f(n)=\Theta(g(n))$$

1. $f(n)$的阶与$g(n)$的阶相等；

2. 对前面有限个n值可以不满足上条件。

定理1: 设f和g是定义域为自然数集合的函数，

（1）如果 $\lim_{n\to\infty} f(n)/g(n) \neq \infty$，

那么$f(n) = O(g(n))$　　$f(n)$的增长最多像$g(n)$的增长那么快

（2）如果$\lim_{n\to\infty} f(n)/g(n) \neq 0$，

那么$f(n) = \Omega(g(n))$　算法的运行时间随规模n的增长至少像$g(n)$那么快

（3）如果$\lim_{n\to\infty} f(n)/g(n) = c$，$(c > 0)$，那么$f(n) = \Theta(g(n))$

证明定理（3）： 根据极限定义，对于给定正数ε，存在某个n_0，只要$n \geq n_0$，就有

$$\left|\frac{f(n)}{g(n)} - c\right| < \varepsilon$$

$$c - \varepsilon < \frac{f(n)}{g(n)} < c + \varepsilon$$

取$\varepsilon = c/2$

$$c/2 < f(n)/g(n) < 3c/2 < 2c$$

对所有$n \geq n_0$，$f(n) \leq 2cg(n)$，于是$f(n) = O(g(n))$;

对所有$n \geq n_0$，$f(n) \geq (c/2)g(n)$，于是$f(n) = \Omega(g(n))$

从而 $f(n) = \Theta(g(n))$

例13：设$f(n)=\frac{1}{2}n^2-3n$，证明$f(n)=\Theta(n^2)$

证：因为

$$\lim_{n\to\infty}\frac{f(n)}{n^2}=\lim_{n\to\infty}\frac{\frac{1}{2}n^2-3n}{n^2}=\frac{1}{2}$$

根据定理1，有 $f(n)=\Theta(n^2)$

5. 渐近分析记号的若干性质

- 运算规则

1. $O(f)+O(g)=O(max(f, g))$
2. $O(f)+O(g)=O(f+g)$
3. $O(f)\cdot O(g)=O(f\cdot g)$
4. 如果 $g(n)=O(f(n))$，则 $O(f)+O(g)=O(f)$
5. $f=O(f)$
6. $O(cf(n))=O(f(n))$

（1）传递性：

- ✓ $f(n)= O(g(n))$，$g(n)= O(h(n)) \Rightarrow f(n)=O(h(n))$
- ✓ $f(n)= \Omega(g(n))$，$g(n)= \Omega(h(n)) \Rightarrow f(n)= \Omega(h(n))$
- ✓ $f(n)= \Theta(g(n))$，$g(n)= \Theta(h(n)) \Rightarrow f(n)= \Theta(h(n))$

（2）自反性：

$f(n)= \Theta(f(n))$　　$f(n)= O(f(n))$　　$f(n)= \Omega(f(n))$

（3）对称性：

$f(n)= \Theta(g(n)) \Leftrightarrow g(n)= \Theta(f(n))$

（4）互对称性：

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

例14： 常函数$f(n)=2300$

解：令$n_0=0$, $c=2300$，使得对$g(n)=1$,

对所有的n有：$f(n) \leq cg(n)=2300\times 1=cg(n)$

所以 $f(n)= O(g(n))=O(1)$

同样， $f(n) \geq 2300\times 1=cg(n)$

所以 $f(n) = \Omega(g(n))= \Omega(1)$.

所以 $f(n) = \Theta(1)$

对于常函数，算法的执行时间$T(n)$与问题规模n无关。

例15: (1) $n!= \mathbf{O(n^n)}$是否成立？

解：因为 $0 < \frac{n!}{n^n} = \frac{n(n-1)(n-1)\cdots 1}{nn\cdots n} < \frac{nnn\cdots 1}{nn\cdots n} \le \frac{1}{n}$

所以 $n!= O(n^n)$

(2) $n!= O((n+1)!)$是否成立？

解: $(n+1)!=(n+1)\,n! > n!$

(3) $n\sum_{i=1}^{n}\frac{1}{i} = \Theta(nlogn)$

解：$n\sum_{i=1}^{n}\frac{1}{i} \le n\sum_{i=1}^{n}\frac{1}{1} = n^2 = O(n^2)$ ？

上界要尽可能地小：$\sum_{i=1}^{n}\frac{1}{i} = 1 + \sum_{i=2}^{n}\frac{1}{i} \le 1 + \int_1^n \frac{1}{x}dx = 1 + \ln n$

同样，$\sum_{i=1}^{n}\frac{1}{i} \ge \int_1^{n+1} \frac{1}{x}dx = \ln(n+1)$

也即 $\ln(n+1) \le \sum_{i=1}^{n}\frac{1}{i} \le 1 + \ln n$ 使用换底公式：

$\frac{\log(\boldsymbol{n}+\boldsymbol{1})}{\log \boldsymbol{e}} \le \sum_{i=1}^{n}\frac{1}{i} \le 1 + \frac{\log \boldsymbol{n}}{\log \boldsymbol{e}}$ 也即 $\sum_{i=1}^{n}\frac{1}{i} = \Theta(\log \boldsymbol{n})$

$$n\sum_{i=1}^{n}\frac{1}{i} = \Theta(nlogn)$$

例16： 多项式函数$f(n)= 8n^2+3n+2$

解：令$n_0=2$,当$n \geq n_0$时,有$c_1=12$,使得对$g(n)=n^2$,对所有的n有：

$f(n)\leq 8n^2+3n+n \leq 12n^2=c_1g(n)$

所以 $f(n) = O(g(n))=O(n^2)$.

令$n_0=0$,当$n \geq n_0$时,有$c_2=8$,使得对$g(n)=n^2$,对所有的n有： $f(n) \geq 8n^2=c_2g(n)$

所以 $f(n) = \Omega(g(n))= \Omega(n^2)$.

所以 $f(n) = \Theta(n^2)$

□ 结论：多项式函数

✓ 若 $p(n)= a_0+a_1n+a_2n^2+\ldots+a_dn^d$； $a_d>0$； 则 $p(n) = \Theta(n^d)$;

常见的多项式阶有:

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3)$

常见的指数阶有: $O(2^n) < O(n!) < O(n^n)$

一般当n很大时，在计算机上运行比$O(\log n)$复杂性更高的算法已经很困难了。

最优算法: 时间复杂性达到其下界的算法.

两点说明:

- 对规模较小的问题，决定算法工作效率的可能是算法的简单性而不是算法执行的时间。
- 当比较两个算法的效率时，若两个算法是同阶的，必须进一步考察阶的常数因子才能辨别优劣。

时间复杂度并不是表示一个程序解决问题需要花多少时间，而是当问题规模扩大后，程序需要的时间长度增长得有多快。

程序段所需要的计算时间为：$O(n^2)$

```
   int MaxSum(int n, int *a, int &besti, int
&bestj)
    {
    int sum=0;
    for(int i=1;i<=n;i++){
        int thissum=0;
        for(int j=i;j<=n;j++){
             thissum+=a[j];
            if(thissum>sum)   {
                sum=thissum;
                besti=i;
                bestj=j;
            }
        }
    }
        return sum;
```

§ 1.5.2 算法的空间复杂性

- 算法的空间复杂度是指算法运行的存储空间，是实现算法所需的内存空间的大小。
- 一个程序运行所需的存储空间通常包括固定空间需求与可变空间需求两部分。
  - 固定空间需求包括程序代码、常量与结构变量等所占的空间。
  - 可变空间需求包括数组元素所占的空间与运行递归所需的系统栈空间等。
- 二维或三维数组是空间复杂度高的主要因素之一。
- 在算法设计时，为降低空间复杂度，要注意尽可能少用高维数组。

小结

- 算法的定义和特征
- 算法的复杂性分析
- 估计函数阶的方法：计算极限
- 算法的时间复杂度是各步操作时间之和，在常数步的情况下取最高阶的函数即可。