北京航空航天大學
BEIHANG UNIVERSITY

大学计算机基础

（理科类）

第8讲 计算机求解问题与经典方法

北京航空航天大学

目　录

北京航空航天大學
BEIHANG UNIVERSITY

8.1 计算机求解问题与算法

北京航空航天大学

实际问题到计算机程序的映射

- **描述实际问题**——用**自然语言**描述
- **抽象**——找出问题中的本质
- **建模**——对问题本质采用**数学符号**进行模型描述
- **设计算法**——对模型的实现方法和步骤进行计算机符号描述，如**流程图、伪代码等**
- **编写程序**——对算法用**程序设计语言**进行描述

抽象+自动化

1、什么是算法？

1、什么是算法？

- **计算机求解任何问题都要依赖于算法**
- **算法**是一个**有穷规则**的集合，它用规则规定了解决某一特定类型问题的运算序列。通俗地说，算法规定了**任务执行/问题求解**的一系列**方法和步骤**
- 算法是一个**可终止过程**的一组**有序的**、**无歧义的**、**可执行**的步骤的集合

2、算法的五项基本特征

- **有穷性**：一个算法在执行**有穷步**规则之后必须结束
- **确定性**：算法的每一个步骤必须要确切地定义，不得有歧义性
- **输入**：算法有零个或多个的输入
- **输出**：算法有一个或多个的输出/结果，即与输入有某个特定关系的量
- **可行性**：算法中描述的每一步操作都可以通过**已有的基本操作**执行**有限次**实现

【例8.1】求解两个正整数的最大公因子

【例8.1】欧几里德算法：求解两个正整数的最大公因子（公约数）的算法。

- **最大公约数**（greatest common divisor，简写为gcd；或highest common factor，简写为hcf)：**某几个整数共有因子中最大的一个**

- **欧几里德**：古希腊著名数学家
- 公元前300年，《几何原本》（Elements）：欧几里德算法（也有的写为欧几里得算法，又称辗转相除法）

欧几里德算法思想

- **算法思想**
  - （1）首先用被除数除以除数，得到**余数**；
  - （2）如果余数不为0，则再将除数作为被除数，余数作为除数，进行相除，得到新的余数；
  - （3）重复步骤（2），直到**余数为0**，则最后那个**除数**，就是**最大公约数**。

【例8. 1】欧几里德算法描述

输入：正整数*M*和正整数*N*
输出：*M*和*N*的最大公约数
算法：
Step 1. 将一个数赋给*M*，一个数赋给*N*;
Step 2. ***M*除以*N*，记余数为*R***
Step 3. 当*R*不是*0*时，**将*N*的值赋给*M*，*R*的值赋给*N*，*M*除以*N*，记余数为*R*，**返回Step 3；
否则，跳转到Step4
Step 4.最大公约数是*N*，输出*N*，算法结束

循环

	M	N	R	最大公约数
具体问题	**32**	**24**		**?**
算法计算过程				
1	32	24	8	
2	24	8	0	8
具体问题	**11**	**31**		**?**
算法计算过程				
1	11	31	11	
2	31	11	9	
3	11	9	2	
4	9	2	1	
5	2	1	0	1

当R=0，N是最大公约数

- 该算法同样具有**有穷性**、**确定性**、**输入**、**输出**和**可行性**等算法的基本特征

- **思考：M是否必须大于N？**

【课堂练习1】

- 根据**欧几里德算法**描述，用Python编程实现该算法的处理过程；将最大公约数打印输出
- 假定已经通过键盘输入两个正整数M和N

```
M=int(input())
N=int(input())
```

- 请写出核心算法

输入：正整数*M*和正整数*N*
输出：*M*和*N*的最大公约数
算法：
Step 1. 将一个数赋给*M*，一个数赋给*N*;
Step 2. ***M*除以*N*，记余数为*R***
Step 3. 当*R*不是*0*时，**将*N*的值赋给*M*, *R*的值赋给*N*，*M*除以*N*，记余数为*R*，**返回Step 3；
否则，跳转到Step4
Step 4.最大公约数是*N*，输出*N*，算法结束

【讨论1】

- 下面的写法对吗？为什么？

```
while M%N !=0:
    M=N                 #将N的值赋给M
    N=M%N               #将余数赋给N
```

- M=5、N=20时，会怎样？

3、算法的主要描述方法

问题：*如何表达问题的求解规则与求解步骤？如何描述一个算法？*

- **自然语言**——人们日常生活、工作和学习中使用的通用语言
- **计算机语言**——程序设计语言
- **图形化工具**——图形符号
  - 如**流程图**（FlowChart）、**N-S图**、**PAD图**（Problem Analysis Diagram，问题分析图）、**UML图**（Unified Modeling Language，统一建模语言）
  - **特点**：描述过程简洁、明了，直观；不适于太复杂问题
- **伪代码**（类计算机语言，或称伪语言）
  - 介于自然语言和计算机语言之间的算法描述方法，类Pascal、类C
  - 不拘泥于语言的语法结构，着重以灵活的形式表现被描述对象
  - **建议采用**

语法要求严格，描述过于复杂

只适于描述简单问题

【例8.2】不同的算法描述方法比较

【例8.2】采用不同的方法描述sum=1+2+3+4+…+n求和问题的算法。

- **方法一：**

  采用自然语言描述

Start of the algorithm(算法开始)

(1)输入N的值；

(2)设 i 的值为1；sum的值为0；

(3)如果 i<=N，则执行第(4)步，否则转到第(7)步执行；

(4)计算 sum + i，并将结果赋给sum；

(5)计算 i+1，并将结果赋给i;

(6)返回到第3步继续执行；

(7)输出sum的结果。

End of the algorithm(算法结束)

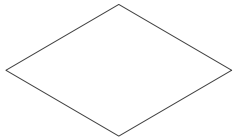
建议采用

流程图

- **流程图**（Flow Chart，框图）是使用最早的算法和程序描述工具，用文字、几何图形和流程线描述算法执行的逻辑顺序
  - **文字**是程序各组成部分的功能说明
  - 不同形状的**几何图形**表示不同性质的操作
  - **流程线**用**箭头**指明算法的执行方向
- **优点**：符号简单，**直观**、易于理解，表现灵活
- **缺点**：对于大型程序或复杂算法，流程图将特别庞大，且难以描述清楚

常见的流程图符号

起止框

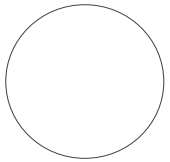


判断框

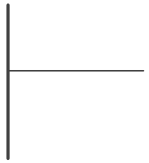
输入输出框

注释

执行框



连接点



流程线

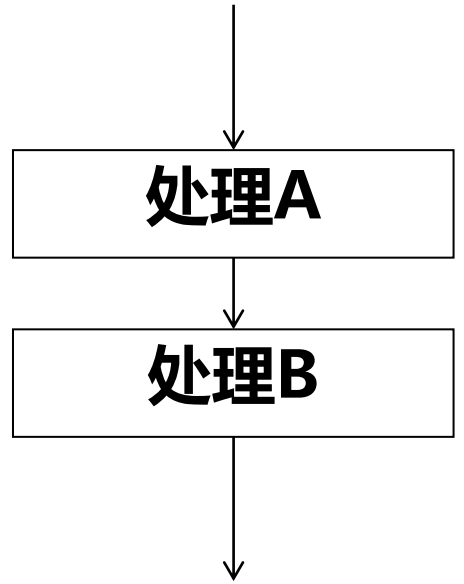
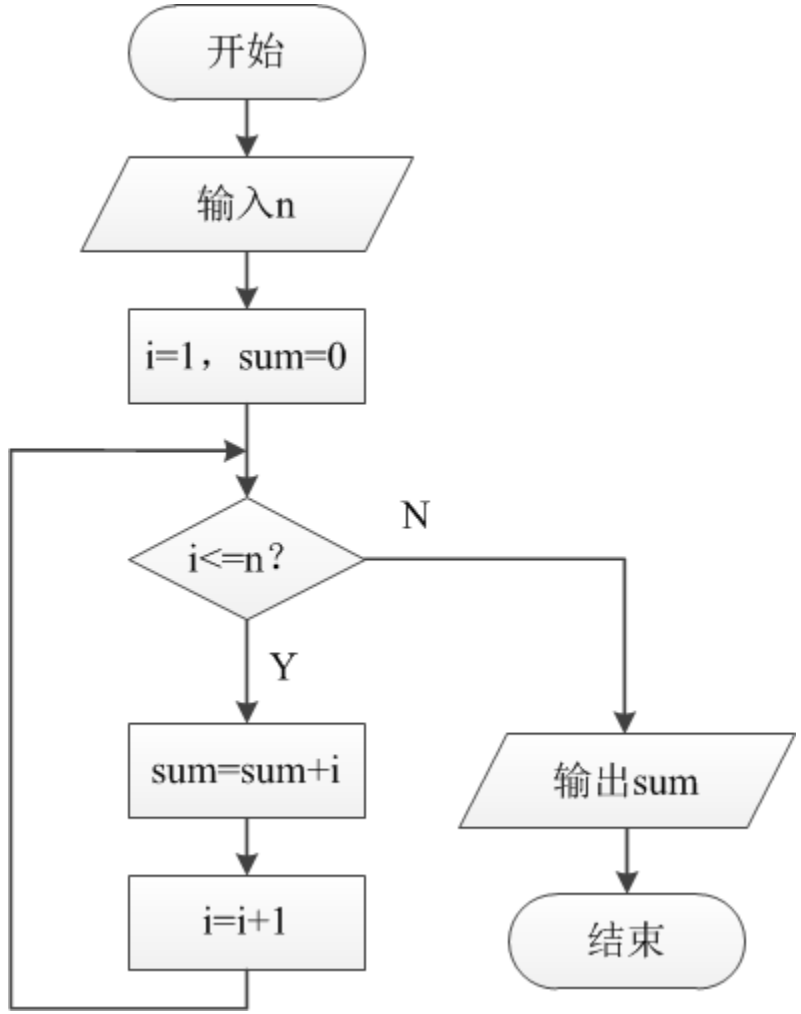


图 形	说 明
圆角矩形	表示“开始”与“结束”
矩形	表示行动及处理方案、普通工作环节，**赋值语句**
菱形	表示问题判断或判定环节，**条件语句**
平行四边形	表示输入输出
连接点	表示流程之间的连接
箭头	表示工作流方向

【例8.2】方法二：采用流程图描述

- **方法二：**

  采用流程图描述



流程图比自然语言描述更直观，简单算法建议采用流程图

伪代码

复杂算法建议采用

- **伪代码**是介于**自然语言**和**计算机语言**之间的文字和符号，它不能被计算机所理解

- 常用的伪代码是用**自然语言**与**类Pascal、类C语言、类Python**相结合的方法来描述算法
- **优点**：
  - 使用伪代码描述的算法易于转变成某种编程语言
  - 尤其适于描述**大型程序**或**复杂算法**

【例8.2】方法三：采用伪代码描述

- **方法三：采用伪代码描述**

```
//用伪代码描述
input n

i ← 1
sum ← 0
while i <= n
        sum ← sum+i
        i ← i+1

输出sum，且算法到此结束。
```

【例8.2】方法四： 采用Python语言描述

- **方法四：**

采用Python语言描述

```
n=int(input("请输入n："))

sum=0        #累加和，初值为0
i=1          #循环变量
while i<=n:
    sum = sum+i
    i=i+1    #循环变量加1

print("最终累加和sum=", sum)
```

例8.2-sum.py

4、从算法到实现

- 算法给人们编程提供思路
- 计算机求解问题时，按照算法描述的方法和步骤机械执行运算和操作
- 算法的实际**执行效率**与所用**操作**、**数据结构**、**控制结构**、**编码**实现的**效率**、机器**硬件**的影响等多种因素有关
- 最重要的两个因素
  - **数据结构**
  - **控制结构**
- **数据结构是算法设计的基础**，也是算法的操作对象
- **设计某个算法的同时**，需**考虑并构建**适合于该算法的**数据结构**
  - 数据结构包括数据的逻辑结构、存储结构和基本操作

控制结构的考虑

- **控制结构是算法设计中需考虑的另一个要素**
- **控制结构**：计算机执行的各操作之间的执行顺序
- 设计算法时，需要考虑计算机的**基本操作**
  - 算术运算（ + 、 - 、×、÷等 ）
  - 逻辑运算（与、或、非等 ）
  - 关系运算（ < 、 > 、 = 、≤、≥、≠等 ）
  - 函数运算
- 还需考虑各操作之间的执行顺序——**控制结构**
- 算法的控制结构给出了算法的框架，决定了各操作的执行次序

任何复杂的算法都可以用顺序结构、选择结构、循环结构这三种控制结构组合而成

【例8.3】括号的匹配问题

- **【例8.3】括号的匹配问题。**在用计算机输入数学公式时，常常遇到括号的匹配问题，尤其是当公式非常复杂时，有时会漏输入括号。
- 请你设计一个程序，选用**合适的数据结构**来解决该问题，判断包含有4种括号**{，[，<，（，），>，]，}**的字符串是否是合法匹配。
- 【输入】：一行，字符串（长度范围： 1～200 ）
- 【输出】：如果字符串中括号匹配是合法的，输出“yes”，不合法则输出“no”

- **假如从左到右扫描字符串，每遇到左括号，就把它暂存起来；**
- **当遇到右括号时，再与暂存的离它最近的左括号相比，看是否匹配**
- **想一想：采用什么数据结构存储扫描到的左括号？**

【例8.3】设计思路

- **设计思路：用栈这种数据结构存储左括号**
  - **合法的匹配：{ [(1+5) * 2 – 9]/3}**
  - **括号运算有何特点？**

最内层括号**先运算**、**最外层**括号**最后运算**

  - 由于每一个右括号，必定是与在其之前的所有未被匹配的左括号中最靠右的一个进行匹配。所以可以按从左到右的顺序读入字符串，并把所有遇到的**左括号**都放入一个**栈**中
  - 若在读入过程中遇到一个右括号，**则比较该右括号是否与栈顶的左括号是一对**，若是，将栈顶的左括号移除，接着往右扫描字符串；否则，该右括号与栈顶的左括号不是一对，则说明该右括号是多余的，不合法

【例8.3】设计思路（续）

- **括号不匹配存在两种特殊情况**

（1）左括号数目少于右括号

- ✓ 例如：[(1+5) * 2 – 9]/3 + 7 }
- ✓ 当**扫描到某个右括号**时，存储左括号的**栈为空**——说明该**右括号**为**多余**的，括号不匹配

（2）左括号数目多余右括号

- ✓ 例如：{ [(1+5) * 2 – 9]/3
- ✓ 当扫描完字符串后，存储左括号的栈**不为空**——说明**左括号**数目**多**余右括号，括号不匹配

【例8.3】采用伪代码描述算法

```
//1、输入和预处理
input string                  //待检测字符串
left = 所有左括号             //列表
right = 所有右括号            //列表（按与左括号匹配的顺序存放）
stack = 空列表                //栈，保存从输入字符串中扫描到的左括号
flag ← 0                      //标志位，若括号匹配，则flag = 0；否则，flag = 1

//2、括号匹配处理
for char in string            //遍历字符串
    if char在left 中          //如果为左括号
        将其压入stack
```

【例8.3】采用伪代码描述算法（续1）

```
elif char 在right中              //如果为右括号
        if stack 为空
            flag = 1             //说明该右括号为多余的，则将flag置为1
        else                     //如果stack不为空
            if char在right列表中索引，与stack栈顶元素在left列表中索引相同
              则左括号出栈 //说明左右括号匹配
            else                 //否则，说明该右括号与stack栈顶元素不匹配
              flag = 1           //则将标志位置为1
              break              //终止循环

if stack不为空                   //说明左括号有多余的，括号不匹配
   flag=1                        //则将flag置为1
```

（1）左括号数目少于右括号

（2）某个右括号与左括号不匹配

（3）左括号数目多于右括号

【例8.3】采用伪代码描述算法（续2）

```
//3、输出
if flag = 0                    //说明右括号都与左括号匹配
    输出yes
else                           //如果flag不为0，说明左右括号不匹配
      输出no
```

【例8.3】Python程序

```
#1、输入和预处理
string = input('请输入待检测字符串：')      #待检测字符串
left = ['{', '[', '<', '(']                  #所有左括号的列表
right = ['}', ']', '>', ')']                 #是所有右括号的列表
stack = []

#2、括号匹配处理
flag = 0                                     #标志位，如果括号匹配，则flag = 0，否则，flag = 1
for char in string:                          #遍历string中每一个字符
    if char in left:                         #如果char为左括号，将其压栈
        stack.append(char)
        print('此时stack为：',stack)
```

栈，保存从字符串中扫描到的左括号

例8.3-括号匹配_方法一.py

【例8.3】Python程序（续1）

```
elif char in right:      #如果char为右括号
    if len(stack) == 0:  #说明该右括号为多余的【（1）左括号数目少于右括号】
        flag = 1         #则将flag置为1
        print('多余的右括号为：',char)
        break            #跳出循环
    else:                #如果stack不为空
        #如果char在right列表中索引与stack栈顶元素在left列表中索引相同
        if right.index(char) ==left.index(stack[-1]) :
            stack.pop()      #说明char与stack栈顶元素匹配，故将左括号删除
            print('匹配一个右括号后stack变为：',stack)
        else:            #否则，不匹配【（2）某个右括号与左括号不匹配】
            flag = 1     #则将flag置为1
            break        #终止循环
```

【例8.3】Python程序（续2）

```
#遍历结束后如果stack不为空，说明左括号有多余的【（3）左括号数目多于右括号】，括号不匹配
if stack != []:
    flag=1                          #则将flag置为1

#3、输出
if flag == 0:                       #说明右括号与左括号匹配
    print("yes")
else:                               #如果flag不为0，则左右括号不匹配
    print("no")
```

输出小技巧

- **技巧**
  - 当有**多种**不同的情况需要处理、但**输出结果只有两种**时，可以采用变量（如flag）来标识，不同情况下，flag=1或flag=0
  - 在最后根据flag的值输出不同的结果

【例8.3】程序运行结果

```
>>>
请输入待检测字符串：{ [ ( 1+5 ) * 2 - 9]/3
此时stack为：  ['{']
此时stack为：  ['{', '[']
此时stack为：  ['{', '[', '(']
匹配一个右括号后stack变为：  ['{', '[']
匹配一个右括号后stack变为：  ['{']
------------------------------------
遍历完字符串后stack最终为：  ['{']
no
>>> ================================ RESTART =
>>>
请输入待检测字符串：[ ( 1+5 ) * 2 - 9]/3 + 7 }
此时stack为：  ['[']
此时stack为：  ['[', '(']
匹配一个右括号后stack变为：  ['[']
匹配一个右括号后stack变为：  []
多余的右括号为：  }
------------------------------------
遍历完字符串后stack最终为：  []
no
>>> ================================ RESTART =
>>>
请输入待检测字符串：{ [ ( 1+5 ) * 2 - 9]/3}
此时stack为：  ['{']
此时stack为：  ['{', '[']
此时stack为：  ['{', '[', '(']
匹配一个右括号后stack变为：  ['{', '[']
匹配一个右括号后stack变为：  ['{']
匹配一个右括号后stack变为：  []
------------------------------------
遍历完字符串后stack最终为：  []
yes
```

左括号数目多于右括号

左括号数目少于右括号

左括号与右括号匹配

【讨论2】

- 4种左括号和右括号还可以采用什么数据结构来存储？

例8.3-括号匹配_方法二.py

例8.3-括号匹配_方法二.py

```
dict1={'(': ')', '<': '>', '[': ']', '{': '}'}

#2、括号匹配处理
flag = 0
for i in range(0,len(string)):          #遍历字符串
    #if string[i] == "(" or string[i] == "<" or string[i] == "[" or string[i] == "{": #若为左括号
    if string[i] in  dict1:             #若为左括号【这样比上句更简单】
        stack.append(string[i])          #则压栈

    #elif string[i] == ")" or string[i] == ">" or string[i] == "]" or string[i] == "}":#若为右括号
    elif string[i] in dict1.values():   #若为右括号【这样比上句更简单】
        #如果此时stack为空，说明该右括号为多余的【（1）左括号数目少于右括号】
        if stack == []:
            flag = 1                    #则将flag置为1
            print('多余的右括号为：',string[i])
            break                       #跳出循环
```

例8.3-括号匹配_方法二.py（续）

```
        else:                          #如果stack不为空
            c = stack.pop()            #则将stack中的最后一个字符弹栈
            #若该右括号与stack中的栈顶对应dict1中的值相等，说明此时右括号与栈顶左括号匹配
            if (string[i] == dict1[c]):
                print('匹配一个右括号后stack变为：',stack)
            else:                      #否则，说明不匹配【（2）某个右括号与左括号不匹配】
                flag = 1
                break

if stack != []:    #遍历结束后stack不为空，说明左括号有多余的【（3）左括号数目多于右括号】
    flag=1                                  #则将flag置为1

#3、输出
#同方法一
```

北京航空航天大學
BEIHANG UNIVERSITY

8.2 枚举法

北京航空航天大学

什么是枚举法？

- **枚举法（穷举法，Enumeration）**就是按问题本身的性质，通过**多重循环**——列举出该问题**所有可能**的解（不能遗漏，也不能重复），并在**逐一列举**的过程中，检验每个可能的解是否是问题的真正解，若是，我们采用这个解，否则抛弃它
- **基本思想**
  - 首先依据题目的部分条件确定答案的**大致范围**
  - 然后在此范围内对所有**可能的解**逐一**验证**，直到全部验证完毕为止
- 枚举法求解问题的核心思路是**暴力破解**，让高速的计算机从事重复运算

设计枚举法的一般方法

设计枚举法的一般方法

- 1. 明确枚举的**对象**：求解变量
- 2. 确定问题**解**的可能搜索的**范围**： 用**循环**或**循环嵌套结构**实现
- 3. 写出**符合问题的解的条件**：采用**if语句**
- 4. **优化程序**，缩小搜索范围，减少程序运行时间

有几个变量就使用几层循环

当明显不满足解的条件时，提前终止循环

枚举法的适用范围

- 求解**方程（组）**、求解**极值**，但数据量不大

（1）求解方程（组）（尤其是不定方程）

① 根据题目建立方程或方程组

② 方程（组）中**有几个变量就建立几层循环**

③ 根据变量特性确定取值范围

④ 以**方程（组）**为**判断条件**逐一检验符合要求的解

（2）求解极值

① 确定求解极值的**范围**

② 确定求解极值的**条件**

③ 常用**一层循环**逐一检验符合要求的解

【课堂练习2】

- **【课堂练习2】考虑用枚举算法解以下问题：**

今有鸡兔同笼，有九十四足，问鸡兔各几何？

- **请列出求解方程公式，并确定变量范围**

【例8.4】枚举法求解阿姆斯特朗数

【例8.4】阿姆斯特朗数。

- 如果一个n位正整数的各位数字的n次方之和等于它本身，则称这个正整数为**阿姆斯特朗数**（也称为自恋性数）。例如153（153=1*1*1+3*3*3+5*5*5）是一个三位数的阿姆斯特朗数，8208则是一个四位数的阿姆斯特朗数。
- 试编程找出所有的**三位数**到**七位数**中的阿姆斯特朗数，打印每个阿姆斯特朗数及其位数。

设计思路

- 采用**枚举法**解决该问题，依次取**100~9999999**以内的各数（设为i），将i的各位数字分解后，据阿姆斯特朗数的性质进行计算和判断
  - ✓ 采用**for循环**确定变量**i**的枚举的空间
  - ✓ **关键**：**如何获得变量i每一位上的数字？**
    - 先采用**str函数**（或repr函数）将变量i转换为一个字符串
    - 再**内嵌for语句**遍历该字符串的每个**字符**，计算各位数字的位数次方之和并累加
  - ✓ 接着采用**if语句**，判断是否符合问题求解条件
    - 如果是阿姆斯特朗数，则输出i的位数以及i，否则继续穷举

采用伪代码描述【例8.4】的算法

指定整数区间[a, b] =[100, 9999999]

for i = a to b

total ← 0　　// i的各位数字的位数次方之和

i转换为字符串string

n ←string的长度

for char in string

每个字符char转换为数字num

total ← total +num^n

if tota >i then break　　//如果 total已经大于 i，则i不可能为阿姆斯特朗数，不必继续计算total，**退出内层循环**

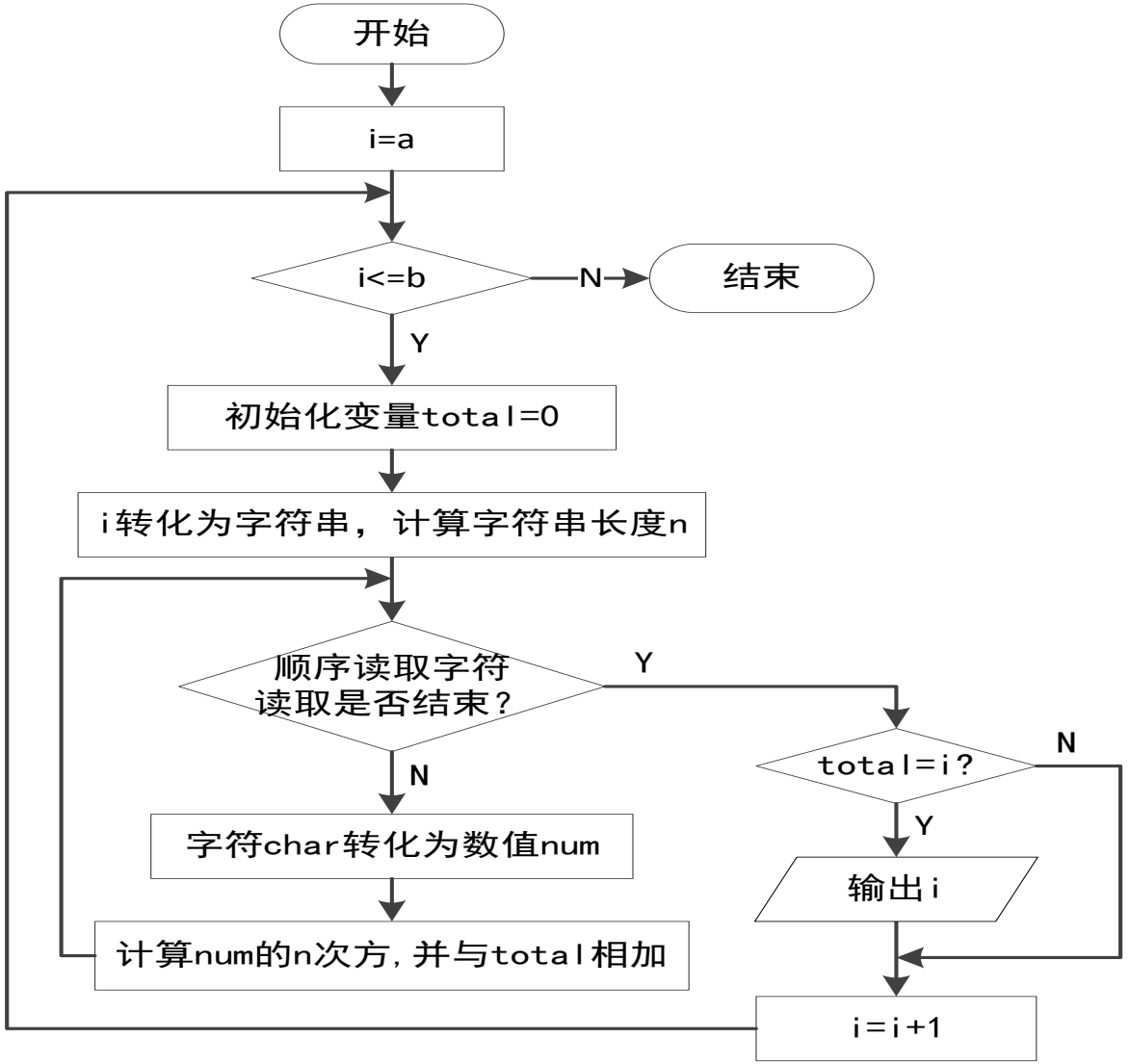
明显不满足题目条件时，可提前跳出循环

if total =i then 输出i　　//是阿姆斯特朗数

算法结束

符合问题的解的条件

采用流程图描述【例8.4】的算法



【例8.4】 Python程序

用**循环结构**确定解的范围

```
for i in range(100, 10000000):                  #穷举100~9999999以内的各数
    total = 0                                   #total为i中各位数字的位数次方之和，初值为0
    string = str(i)                             #将i转换为字符串形式存储在 string 中，便于取各位数字
    #string = repr(i)                           #采用str函数或repr函数都可以
    # (1) 计算各位数字的位数次方之和
    for char in string:                         #对于string中每个字符
        num = int(char)                         #先将其转换为int
        total += num ** len(string)             #然后计算每位数字的位数次方之和（字符串的位数即为len(string)）
        if total > i:                           #如果 total已经大于i，则i不可能为阿姆斯特朗数，退出内层循环
            break

    # (2) 判断是否符合问题求解条件
    if total == i:                              #找到阿姆斯特朗数，则输出i
        print('位数=',len(string),'时，阿姆斯特朗数是：',i)
```

提前终止循环，提高运行效率

例8.4-armstrong.py

【例8.4】的程序运行结果

```
>>>
位数= 3 时, 阿姆斯特朗数是:  153
位数= 3 时, 阿姆斯特朗数是:  370
位数= 3 时, 阿姆斯特朗数是:  371
位数= 3 时, 阿姆斯特朗数是:  407
位数= 4 时, 阿姆斯特朗数是:  1634
位数= 4 时, 阿姆斯特朗数是:  8208
位数= 4 时, 阿姆斯特朗数是:  9474
位数= 5 时, 阿姆斯特朗数是:  54748
位数= 5 时, 阿姆斯特朗数是:  92727
位数= 5 时, 阿姆斯特朗数是:  93084
位数= 6 时, 阿姆斯特朗数是:  548834
位数= 7 时, 阿姆斯特朗数是:  1741725
位数= 7 时, 阿姆斯特朗数是:  4210818
位数= 7 时, 阿姆斯特朗数是:  9800817
位数= 7 时, 阿姆斯特朗数是:  9926315
>>>
```

- 当位数=1-6时，结果很快出来
- 当n=7时，几分钟结果才显示完
- **枚举法只适于问题规模不大的场合**

如何提高枚举法效率？

- **如何提高枚举法效率？**
  - 能否减少循环层数？——减少列举**变量个数**
  - 能否缩小循环范围？——缩小变量**取值范围**

- **执行效率和编写简单**
  - 一个问题可以用不同算法解决
  - 结构简单可能会提高执行效率，但会降低算法的易读性和通用性
  - 选择算法时既要注重**算法的简单易用**，又要兼顾**效率性能**

权衡可读性和执行时间

参见MOOC 8.3节课件【例8-5】填数游戏

北京航空航天大學
BEIHANG UNIVERSITY

8.3 递归法

北京航空航天大学

什么是递归？

- **递归**（recursion）：指函数/过程/子程序在运行过程中直接或间接调用自身而产生的重入现象
- 递归是计算机科学的一个重要概念，递归的方法也是程序设计中有效的方法
- **递归算法**：包含递归过程的算法
  - **优点**：采用递归法编写程序，能使求解的过程变得**简洁**和**清晰**，可读性强
  - **缺点**：**运行效率较低，耗费较多**的计算**时间**；也会占用**较多**的**存储空间**

用递归法求解问题的一般思路

- 一个过程或函数直接或间接地**调用自己本身**称为**递归**
- **用递归法求解问题的一般思路**
  - 把原问题**分解**为更小的子问题，再从子问题里慢慢寻找原问题的解
  - 首先列出**递归表达式**，然后用**程序设计语言**的方式把它表现出来
  - 递归大多可转化为**循环**或者模拟调用**栈**来实现，但递归表达更利于理解（典型的**分治**思想）

【课堂练习3】

- 在教材第3章中，曾经学习了**递归函数**——在函数的定义中包含对函数自身的调用
- 采用递归函数实现求正整数**n的阶乘**
- 请给出下面程序中**横线**上的语句

- **自然数n的阶乘写作n!。** **n! = n×(n-1)×(n-2) ×…×1**
- **阶乘的递归定义**
  - **递归基础**：0!= 1，1!= 1
  - **递归步骤**：n! = n * (n-1)! （n>1）

```
#定义递归函数
def fun(n):
    if n == 0 or n==1:
        return _____
    elif n>1:
        return _____
    else:
        print('请输入正整数')
```

什么样的问题适合于用递归去求解？

- **什么样的问题适合于用递归去求解呢？**

1）问题的定义是递归的

例如：Fibonacci函数，阶乘

2）数据的结构是按递归定义的

例如：树的遍历

3）问题的解法需要使用递归法去实现

例如：问题用**分治法**、**回溯法**等策略建模求解

1）问题的定义是递归的

1）问题的定义是递归的

- 许多**数学公式**、**数列**等的定义是递归的。例如，求n!和Fibonacci数列等
- 这些问题的求解过程可以将其**递归定义**直接转化为对应的**递归算法**

阶乘函数的定义

■ **例如：阶乘函数的递归定义**

递归基础

$$n! = \begin{cases} 1 & \text{当} n = 0 \text{或} 1 \text{ 时} \\ n * (n-1)! & \text{当} n > 1 \text{时} \end{cases}$$

递归步骤

- **递归定义**包括两部分
  - **递归基础**——递归计算的起点
  - **递归步骤**——递归计算的步骤

递归函数

$$f(n) = \begin{cases} 1 & \text{当} n = 0 \text{或} 1 \text{时} \\ n * f(n-1) & \text{当} n > 1 \text{时} \end{cases}$$

- **函数f(n)的定义用到了自己本身f(n - 1)**

Fibonacci数列

- **再如：Fibonacci数列**（斐波那契数列或斐波纳契数列），无穷**数列 1，1，2，3，5，8，13，21，34，55，……**，称为Fibonacci数列

- **递归定义**

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

递归基础

递归步骤

- **该函数对任一自然数的计算过程为：**

F(0)=1; F(1)=1; F(2)=F(1)+F(0)=2;

F(3)=F(2)+F(1)= 3; F(4)=F(3)+F(2)= 3+2=5;

… …

【例8.6】递归法求斐波那契数

【例8.6 】采用递归法求第N项斐波那契数。

- 斐波那契数列是一类典型的可由递归求解的问题
  - **[步骤1] 描述递归关系**
  - **[步骤2] 确定递归边界**（ n=0 或n=1 ）

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

- **递归边界（递归基础，递归出口）**：当到达递归边界时，递归结束（否则会无限次数地递归下去）

【例8.6】Python程序

例8.6-Fibonacci【递归法】.py

```
#递归法求第N项斐波那契数

N=int(input("请输入一个正整数:"))

def Fib(N):
    if N==0 or N==1:                #递归边界
        return 1
    else:
        return Fib(N-1)+Fib(N-2)    #自己调用自己

for i in range(0,N+1):
    print ("第",i,"项斐波那契数：",Fib(i))
```

调用函数Fib(N)

```
请输入一个正整数：10
第 0 项斐波那契数： 1
第 1 项斐波那契数： 1
第 2 项斐波那契数： 2
第 3 项斐波那契数： 3
第 4 项斐波那契数： 5
第 5 项斐波那契数： 8
第 6 项斐波那契数： 13
第 7 项斐波那契数： 21
第 8 项斐波那契数： 34
第 9 项斐波那契数： 55
第 10 项斐波那契数： 89
```

2）数据结构是递归的

自学

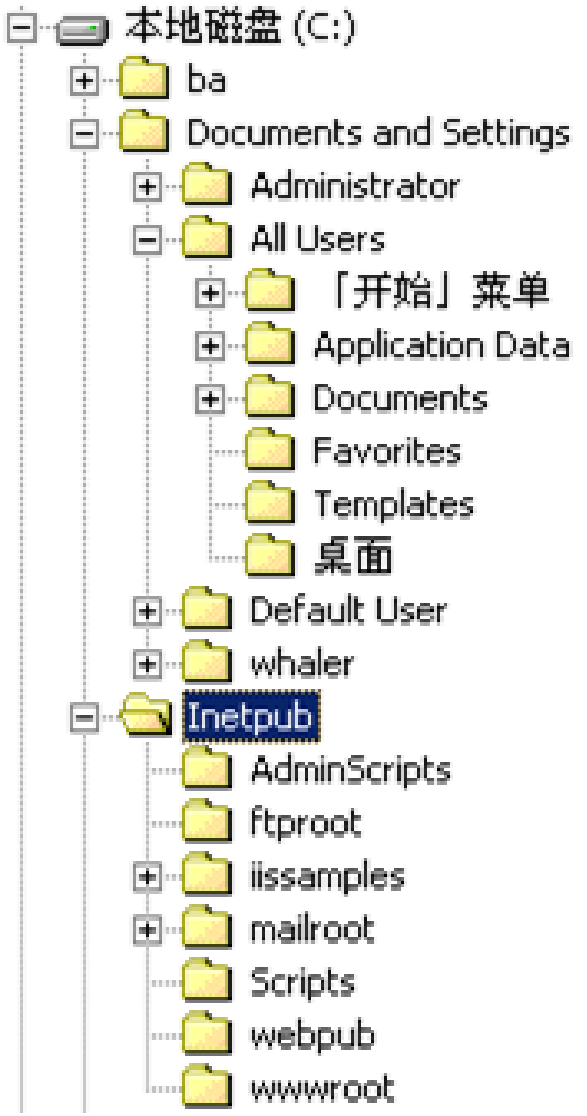
2）数据结构是递归的

- **磁盘根目录系统**就像一棵**倒立的树**
- 数据与数据成一对多的关系，是一种典型的**非线性**结构—**树形结构**
- 树的定义是递归的

◆ **树**（Tree）是n（n≧0）个结点的**有限**集合T，当n=0时称T为**空树**，否则，当T非空时：

（1）有且仅有一个特定的称为**根**的结点；（**递归基础**）

（2）除根结点外的其余结点可被分为k个互不相交的集合$T_1,T_2,\ldots,T_{k(k}\geq 0)$，其中每一个集合$T_i$本身也是一棵树，被称为根的**子树**（Subtree）。（**递归步骤**）

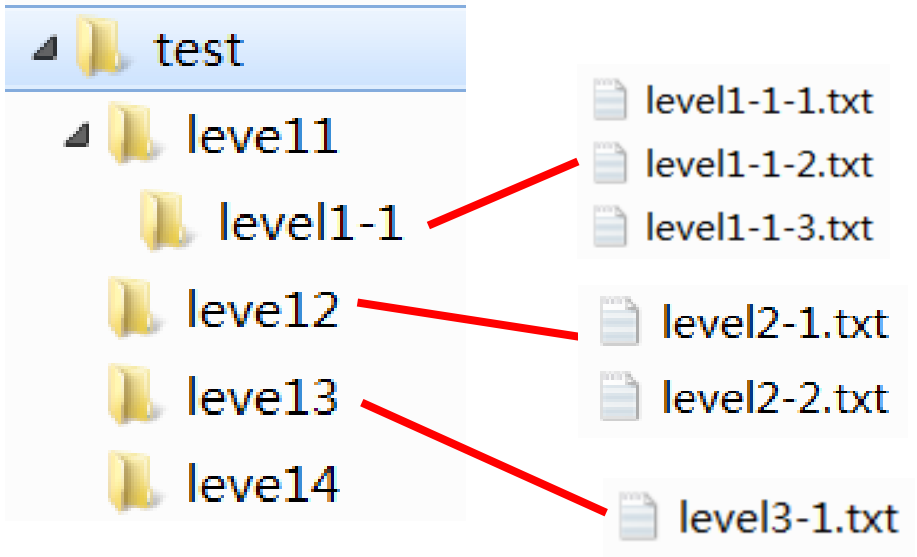


【例8.7】递归法遍历文件夹和文件

自学

【例8.7】采用递归法遍历指定文件夹下的所有子文件夹和所有文件。

- 要求根据目标文件夹的路径（‘d:/test’），输出目标文件夹以及其子文件夹下所有文件的名称。
- 采用**后序遍历**（最后遍历根结点）方式，首先输出的是树形结构的**叶子节点**，即按照**字母排序顺序**找到的第一个子文件夹中、存储在最深层子文件夹中的文件名。



参见MOOC“8.3 递归法”中**【例8-7】**

```
['level1-1-1.txt', 'level1-1-2.txt', 'level1-1-3.txt', 'level1-1', 'level1',
'level2-1.txt', 'level2-2.txt', 'level2', 'level3-1.txt', 'level3', 'level4']
```

3）问题的解法需要使用递归法实现

【例8.8】汉诺塔问题。

- 汉诺塔（也称梵天塔）问题是印度的一个古老传说。
- 开天辟地之神勃拉玛在一个庙里留下了三根金刚石柱，并在第一根柱上从上到下依次串着**由小到大**不同的64片中空的圆型金盘。
- 神要庙里的僧人把这64片金盘全部搬到**第三根**上，搬完后，金盘仍保持原来**由小到大**的顺序。要求**每次只能搬一个**，可利用中间的一根石柱作为中转；在搬运的过程中，不论在哪个石柱上，**大的金盘都不能放在小的金盘上面**。
- 神说，当所有的金盘都从事先穿好的那根石柱上移到另外一根石柱上时，世界就将在一声霹雳中消灭了。

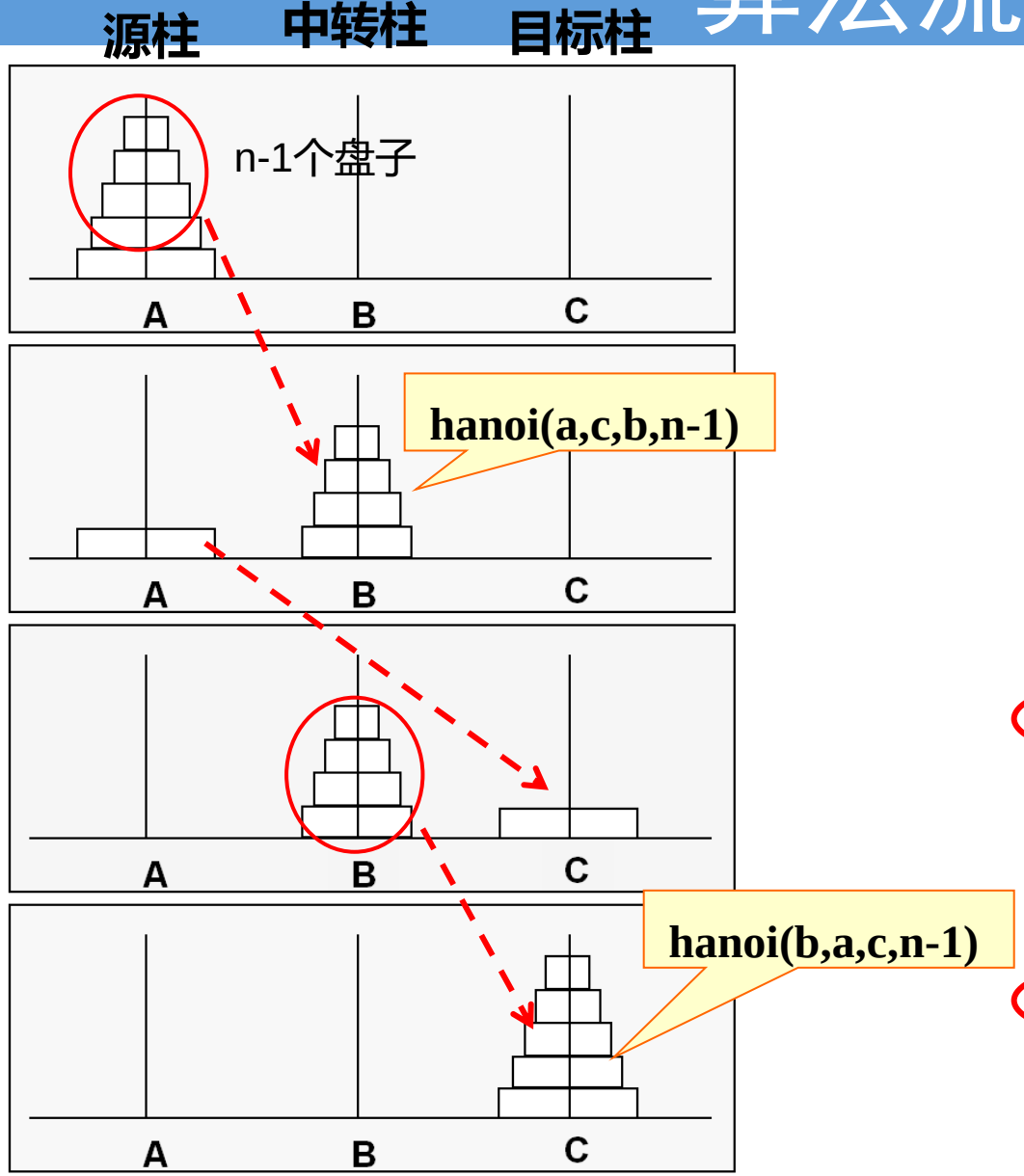
圆盘何时能搬完？

- ***神说的是真的吗？***
- 将三个金刚石柱编号为A、B、C，A为**源柱**，B为**中转柱**，C为**目标柱**
  - 当金盘为**1个**时，只需要移动**1次**；
  - 当金盘有**2个**时，按A→B，A→C，B→C的顺序移动金盘，需要移动**3次**；
  - 当金盘有**3个**时，按A→C，A→B，C→B，A→C，B→A，B→C，A→C的顺序移动金盘，需要移动**7次**。
  - 依次类推，当金盘为**n片**时，需要移动**2^n-1次**。即当金盘为64片时，需要移动$x=2^{64}-1=18446744073709551615$次，假设移动一次需要1秒钟，一个平年365天有31536000 秒，闰年366天有31622400秒，平均每年有$t=31556952$秒，则需要**year=x/t**$=5845.54*10^8=$**5845.54亿年**！

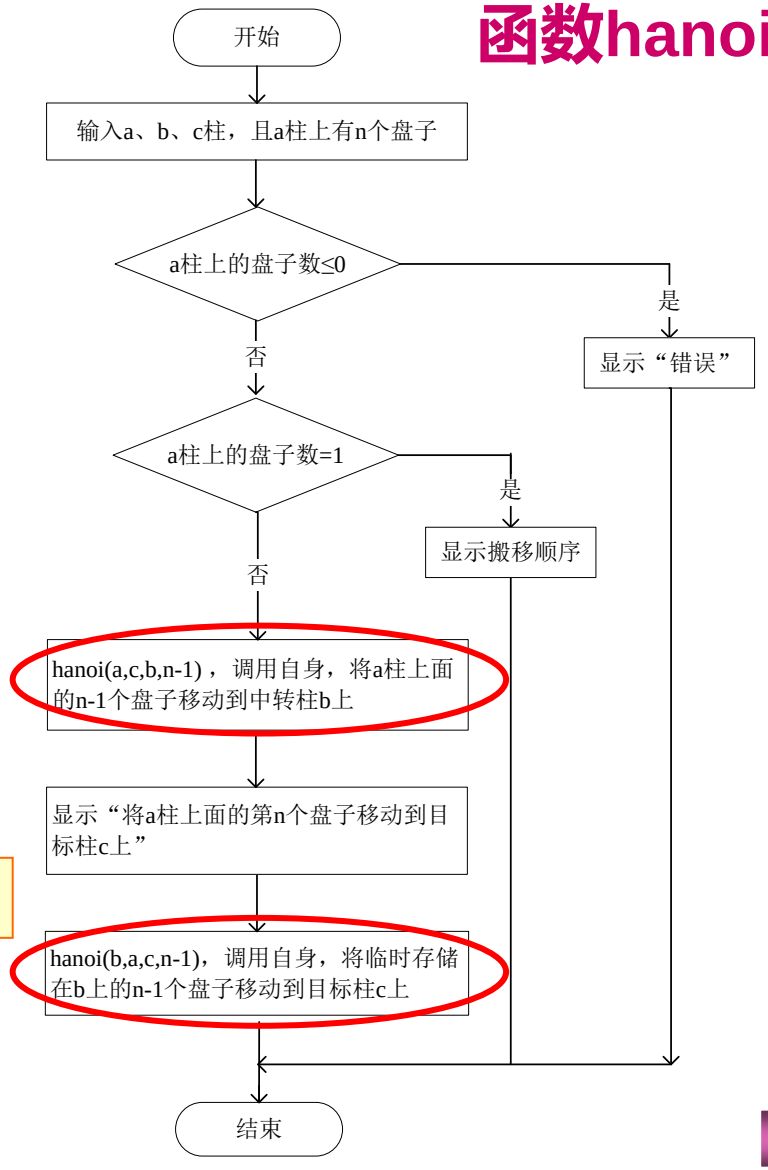
问题分析

- 要将N个盘子从A柱移到目标柱C上，都需要先把A柱上面的N-1个盘子移动到中转柱B上（**子问题1**），再将A上的唯一盘子（最大的盘子）移动到目标柱C上；再把临时存储在B上的N-1个盘子移动到目标柱C上（**子问题2**）；当N为1（边界条件）时，则只需简单地将其移动到目标柱C上即可
- **原问题可分解为子问题1和子问题2**，而子问题1和子问题2与原始问题本质上是等价的，即**问题的本身是递归定义的**。对于这种**递归定义的问题，可使用递归法进行求解**

算法流程



函数hanoi(a,b,c,n)



【例8.8】 Python程序

```
#盘子从上至下标号为1、2、3……、n
import time                                    #导入time模块

#（1）定义解决汉诺塔问题的递归算法
def hanoi(a,b,c,n):
    if n<=0:
        print ("错误！")
    elif n==1:
        print ("将盘子",n,"从",a,"移到",c)
    else:
        hanoi(a,c,b,n-1)                       #子问题1，调用自身
        print ("将盘子",n,"从",a,"移到",c)
        hanoi(b,a,c,n-1)                       #子问题2，调用自身
```

其time()方法获取当前时间

a为源柱，b为中转柱，c为目标柱

把a柱上面的n-1个盘子移动到b上，c为中转柱

把临时存储在b上的n-1个盘子移动到c上

例8.8-hanoi【递归法】.py

【例8-8】的Python程序（续）

```
#（2）输入任意一个自然数n代表盘子的总个数
num =int(input("要搬移的盘子数量是："))
start = time.time()                    #获取自纪元以来的当前时间（以秒为单位）

#（3）调用递归算法
print (num,"个盘子的搬移过程：")
print("共需要移动%d次" % (2**num-1))    #共需移动2^num-1次
hanoi("a","b","c",num)
print()                                #换行
end = time.time()                      #获取程序结束的时间
print("程序执行时间为：%0.3f s" % (end-start))
```

注意：在“def hanoi(a,b,c,n)”中，a,b,c,n都是变量，不是字符串；在hanoi(“a”,“b”,“c”,num) 中，因为a、b、c没有事先赋值，所以，“a”,“b”,“c”是字符串，必须用引号括起来

汉诺塔问题的程序运行结果

```
>>>
要搬移的盘子数量是：0
0 个盘子的搬移过程：
共需要移动0次
错误！请重新输入盘子数量

程序执行时间为：1.680 s
>>> ==================
>>>
要搬移的盘子数量是：1
1 个盘子的搬移过程：
共需要移动1次
将盘子 1 从 a 移到 c

程序执行时间为：1.380 s
>>> ==================
>>>
要搬移的盘子数量是：3
3 个盘子的搬移过程：
共需要移动7次
将盘子 1 从 a 移到 c
将盘子 2 从 a 移到 b
将盘子 1 从 c 移到 b
将盘子 3 从 a 移到 c
将盘子 1 从 b 移到 a
将盘子 2 从 b 移到 c
将盘子 1 从 a 移到 c

程序执行时间为：1.648 s
>>> ==================
```

```
>>>
要搬移的盘子数量是：4
4 个盘子的搬移过程：
共需要移动15次
将盘子 1 从 a 移到 b
将盘子 2 从 a 移到 c
将盘子 1 从 b 移到 c
将盘子 3 从 a 移到 b
将盘子 1 从 c 移到 a
将盘子 2 从 c 移到 b
将盘子 1 从 a 移到 b
将盘子 4 从 a 移到 c
将盘子 1 从 b 移到 c
将盘子 2 从 b 移到 a
将盘子 1 从 c 移到 a
将盘子 3 从 b 移到 c
将盘子 1 从 a 移到 b
将盘子 2 从 a 移到 c
将盘子 1 从 b 移到 c

程序执行时间为：2.101 s
>>>
```

当n=10时，程序执行时间为：46.646 s

问题规模越大，程序运行时间越长

总结：递归算法的执行过程

■ 递归算法的执行过程

① 从**最后一项**（**hanoi(a,b,c,n)**）开始自调用前面的几项（ hanoi(a,c,b,n-1)、 hanoi(b,a,c,n-1)），不断地**自调用**，直到到达**递归出口**（ **hanoi(a,b,c,1)** ）才**结束自调用**过程；

② 到达递归出口（ **hanoi(a,b,c,1)**）后，递归算法开始按**最后调用的过程**（ **hanoi(b,a,c,2)**）**最先返回结果值**的次序返回值；

③ 返回到最外层的调用语句（ **hanoi(a,b,c,n)** ）时，递归算法执行过程结束。

递归法的优点和缺点

- **优点：清晰和简单**。一些问题，特别是人工智能问题，依赖于递归提供解决方案
- **缺点：**解题的**运行效率较低**
  - ✓ **时间耗费大**：递归计算实际上是两个过程：**函数调用**和**逐层返回（值）**。当递归越深，它返回的时间就越长。递归的时间复杂度是**指数级**增长的
  - ✓ **空间耗费大**：递归在计算时要存储**所有中间结果**。这些中间结果被在逐层返回时使用。中间结果都会被存储在栈中，当计算规模很大时，常会导致**栈溢出（MLE）**，即没有足够的空间存储中间结果

【课后练习】

- **【课后练习】考虑用枚举法解以下问题：**

1、1XX47能被57或67整除，求这样的数字

2、班级30人聚餐总共花费500元，收费时出席的男生30元，出席的女生25元，未出席的学生10元，请问男生、女生、未出席的各多少？

3、 10枚硬币中有一个是伪币，且伪币比真币要轻。如何用一台天平找到该枚伪币？假定可以称多次。

- **1~2题请列出求解方程公式，并确定变量范围**
- **第3题给出思路**
- **尝试编程实现**

总结：如何思考编程解决一个问题？

- **第一步：分析和构思**
  - **1、输入是什么？求什么？它们之间能否建立一个数学模型（公式）？**
  - **2、要处理的数据的逻辑结构是什么？**
    - ✓ 线性结构（线性表，栈，队列）？非线性结构（树状结构，图状结构)?
  - **3、采用什么数据结构存储数据？**
    - ✓ **线性表**采用**列表**、**字符串**（如果本身是字符串）存储
    - ✓ **栈**或**队列**采用**列表**存储，采用列表的append()方法、pop()方法模拟栈或队列的操作
    - ✓ 具有**特殊映射关系（名和值）**的数据采用**字典**存储

总结：如何思考编程解决一个问题？（续1）

- **4、采用什么程序控制结构？**
  - ✓ 如果一种操作需要重复多次，宜采用**循环结构**
    - 当循环次数已知时，宜采用**for**语句
    - 当循环次数未知时，宜采用**while**语句（注意在循环体中一定要有一条改变循环条件表达式的语句）或**while True**语句（注意当循环条件不满足时，采用break终止循环）
  - ✓ 如果是在不同的条件下执行不同的操作，宜采用**分支结构**
    - 当有两个以上条件时，采用**if-elif-else**条件语句
    - 当某个True代码块或者False代码块含有条件语句时，采用**if语句的嵌套**

总结：如何思考编程解决一个问题？（续2）

- **第二步：设计算法**
  - 结合选定的数据存储结构和程序控制结构，设计算法
    - ✓ 在纸上画算法**流程图**，或用**伪代码**描述算法
- **第三步：Python编程**
  - **1、确定输入的数据如何存储**
    - ✓ **单个**输入： a=input()　#输入a为字符串　n=int(input())　#输入n为整数
    - ✓ 一行输入**多个**数据，每个数据之间用一个**空格**隔开
      - 数据**较少**时， a,b,c = input().split()
      - 数据**很多**时，s = input().split()　#s为列表
    - ✓ 一行输入**多个**数据，每个数据之间用一个空格隔开，并转换成**整型**或**浮点型**
      - 数据**较少**时， a,b,c=map(float, input().split())
      - 数据**很多**时， nums = list(map(int, input().split()))　# nums为列表

总结：如何思考编程解决一个问题？（续3）

- **第三步：Python编程**
  - **2、处理**
    - ✓ 根据设计的算法，采用确定的Python数据结构（列表、元组、字符串、字典）和赋值语句、循环语句或条件语句编程实现算法
  - **3、确定处理结果以什么形式输出**
    - ✓ **采用print函数打印处理结果**
    - ✓ 直接输出：print(ans)
    - ✓ 根据**标志位**的不同输出不同的值
    - ✓ **格式化字符串**输出
      - – 当需要打印的变量值为不同类型（如浮点数）时，可以采用格式化字符串的方法，来指明变量值的**类型**、**位数**

```
print(' %0.3f ' % ans)
```

```
if flag == 0:
    print("yes")
else
    print("no")
```