0.3 算法描述与分析

Descr. & Analysis of Algorithms

郝家胜

hao@uestc.edu.cn

自动化工程学院

内容回顾

- 计算机如何解题
  - 分析问题
  - 设计算法
  - 编写程序
- 什么是算法
  - 利用计算机作为求解工具
  - 可计算问题
  - 算法（一系列明确的计算指令）

内容提要

- 算法的概念
- 算法的描述
- 算法的效率

计算机如何解题

分析问题：

对问题进行详细地分析，通过分析，弄清楚已知条件下的初始状态及要达到的目标，找出求解问题的方法和过程，并抽取出一个数学模型，形成算法；

设计算法：

将这个数学模型连同它要处理的数据用计算机能识别的方式描述出来，使之成为计算机能处理的对象；

编写程序：

用程序设计语言设计出具体的问题求解过程，形成计算机程序。

计算机如何解题

- *计算机程序*(Computer Program)

  指示计算机如何去解决问题或完成任务的一组可执行的指令

- *程序设计*(Program Design)

  寻求解决问题的方法，并将其实现步骤编写成计算机可以执行的程序的过程

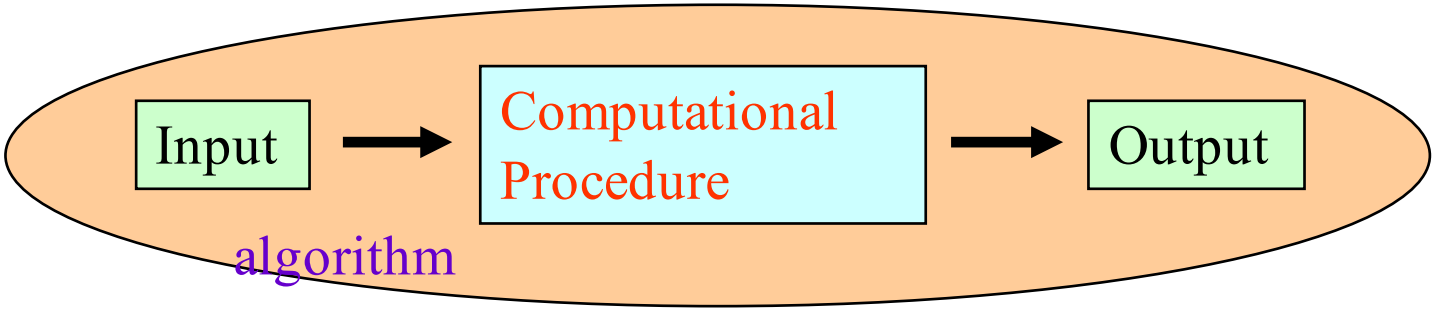
- *Wirth公式*

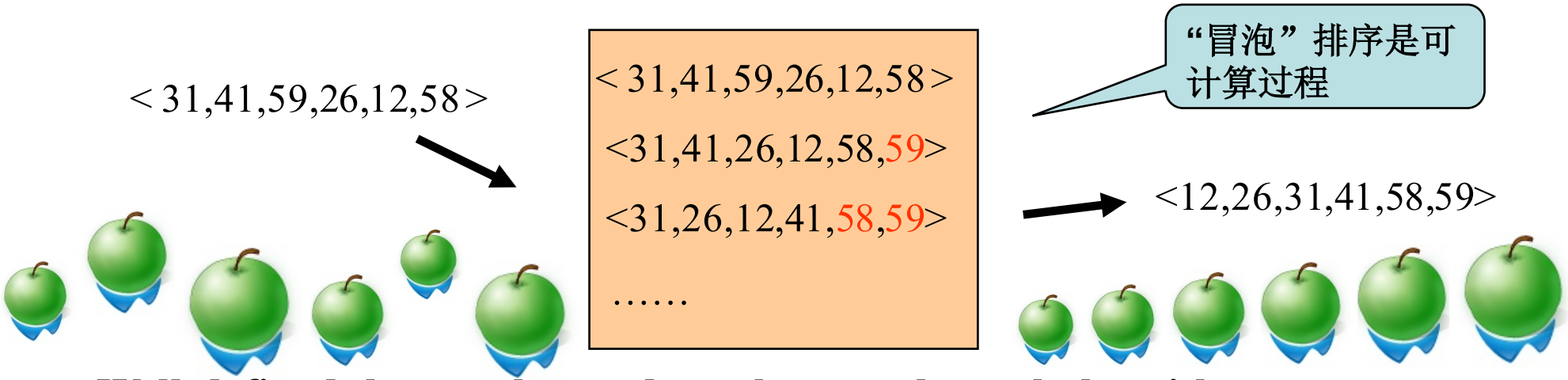
  程序 = 算法 + 数据结构

什么是算法

1. Algorithm is any well-defined computational procedure

（算法：一个定义明确的是可计算过程）





- **Well-defined: know what to do each step; always halts with correct answer .**
- **Efficiency: good or bad?**

Algorithms

- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output
- An algorithm is thus a sequence of computational steps that transform the input into the output
- Program: a series of coded instructions to control the operation of a computer or other machine by using some algorithms

算法的特点

- 算法是在有限步骤内求解某一问题所使用的一组定义明确的规则
- 一种思维方式
  - 从具体的操作规范入手，通过操作过程的构造与实施来解决给定问题的思维方法
  - 描述人类解决问题的操作过程
  - 描述计算机解决问题的计算过程
- 基本特征
  - 有穷性，确定性，输入，输出，可行性

算法的范畴

- 时空有限
  - 解决结果的正确性
  - 解决步骤的有穷性
  - 易于实现的方式
- 算法设计
  - 穷举法，迭代法，递推法，递归法，分治法
  - 贪心法，动态规划法，回溯法
- 算法分析
- 算法实现

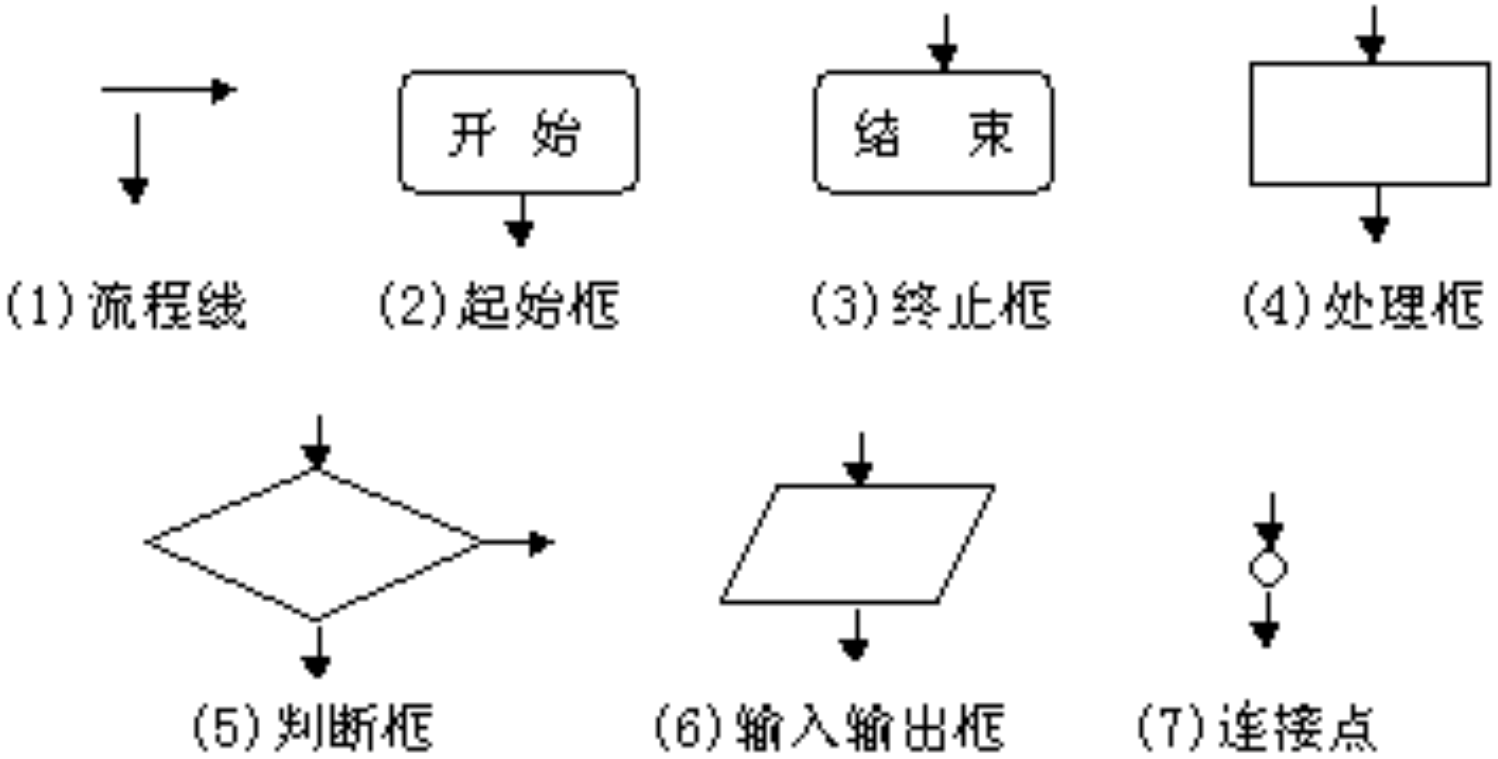
算法的用处

- 计算机导航
- 电子交易
  - 保持银行帐号、密码的私密性，订单的完整性，交易的不可否认性
- *Google's mission*
  - to organize the world's information and make it universally accessible and useful.
- 人类基因项目
  - 找出人类DNA中的所有100 000中基因，确定构成人类DNA的30亿种化学基对的各种序列

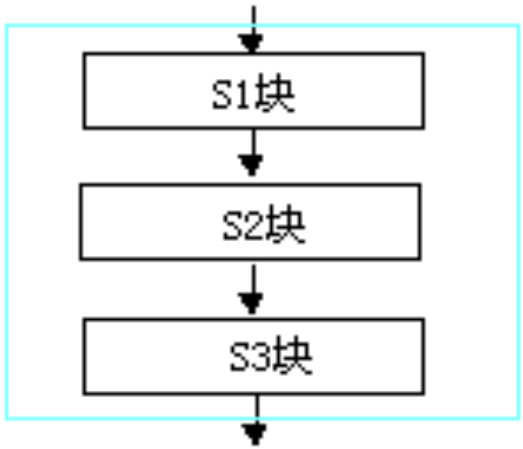
算法的描述

- 自然语言
- 流程图
- 伪代码
- 计算机语言

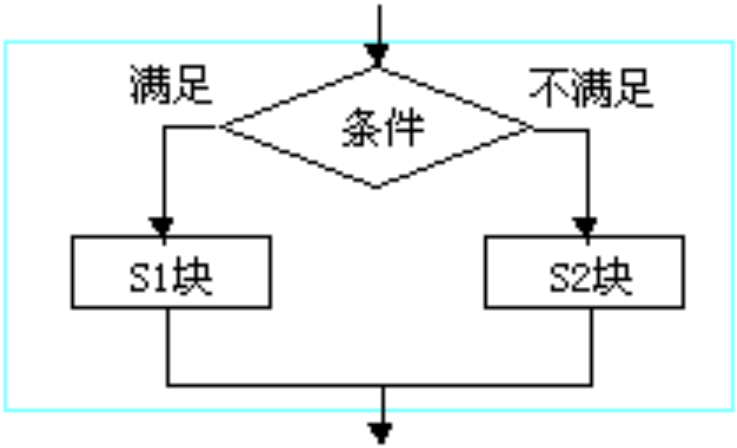
算法的流程图表示法



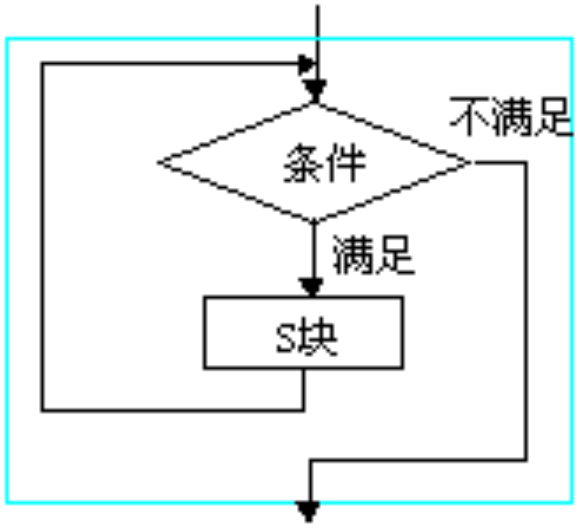
顺序结构、选择结构和循环结构流程图



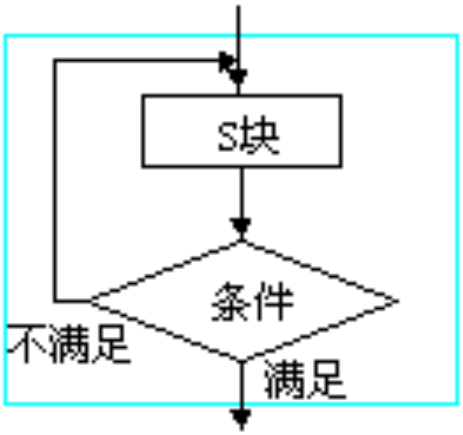
顺序结构



选择结构



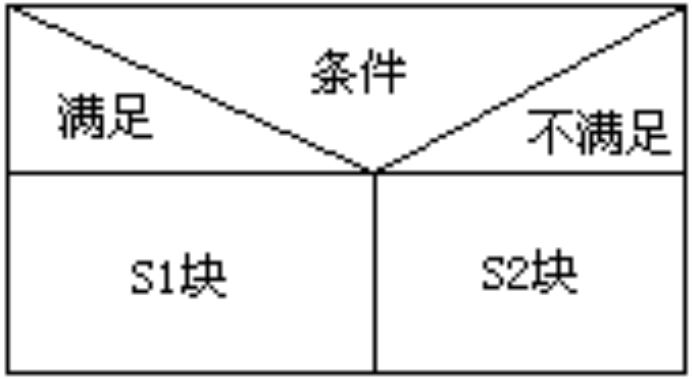
“当型”循环结构



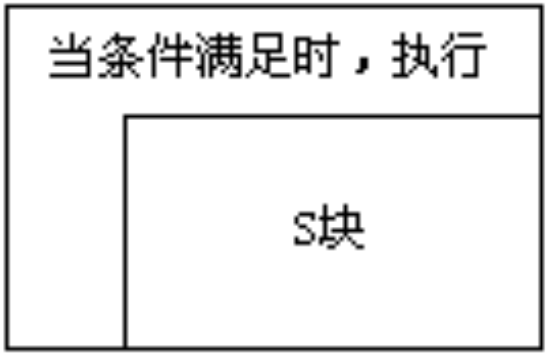
“直到型”循环结构

N-S图

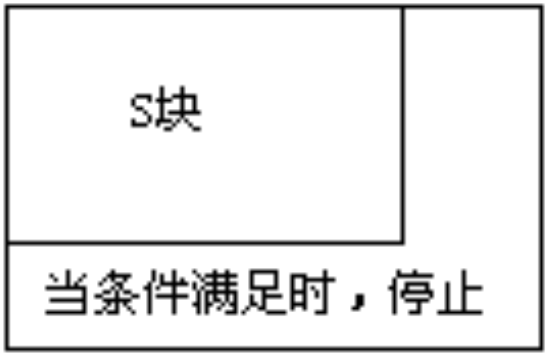
顺序结构



选择结构



“当型”循环结构



“直到型”循环结构

伪代码

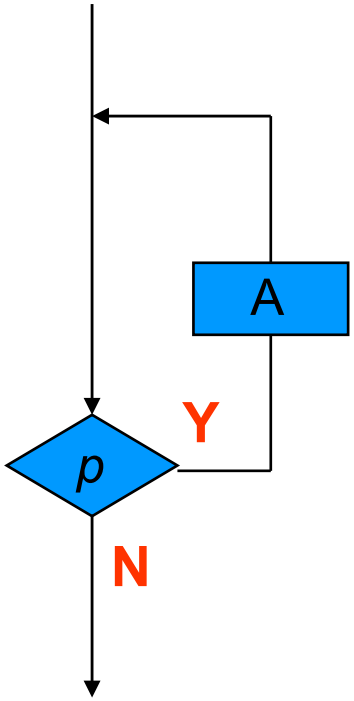
- 伪代码(Pseudocode)是一种算法描述语言
- 使用为代码的目的是为了使被描述的算法可以容易地以任何一种编程语言(Pascal, C, Java, etc)实现。
- 伪代码必须结构清晰，代码简单，可读性好。
- 类似自然语言，但使用伪代码描述算法没有严格的语法限制，它更侧重于对算法本身的描述。
- 例如：

```
x←y
x←20*(y+1)
x←y←30
```

```
x = y;
x = 20*(y+1);
x = y = 30;
```

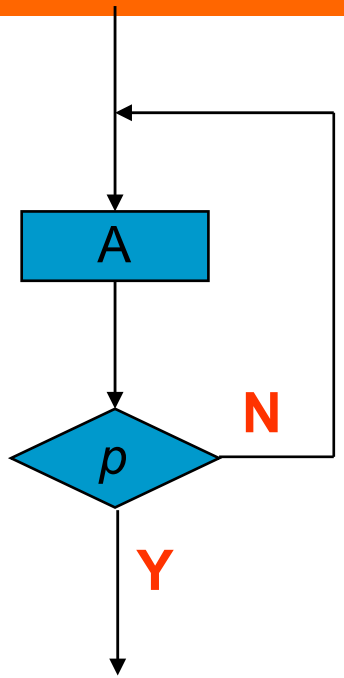
```
x := y;
x := 20*(y+1);
x := 30; y := 30;
```

伪代码描述的循环



当型语句：

```
While   p
    循环体
End
while
```

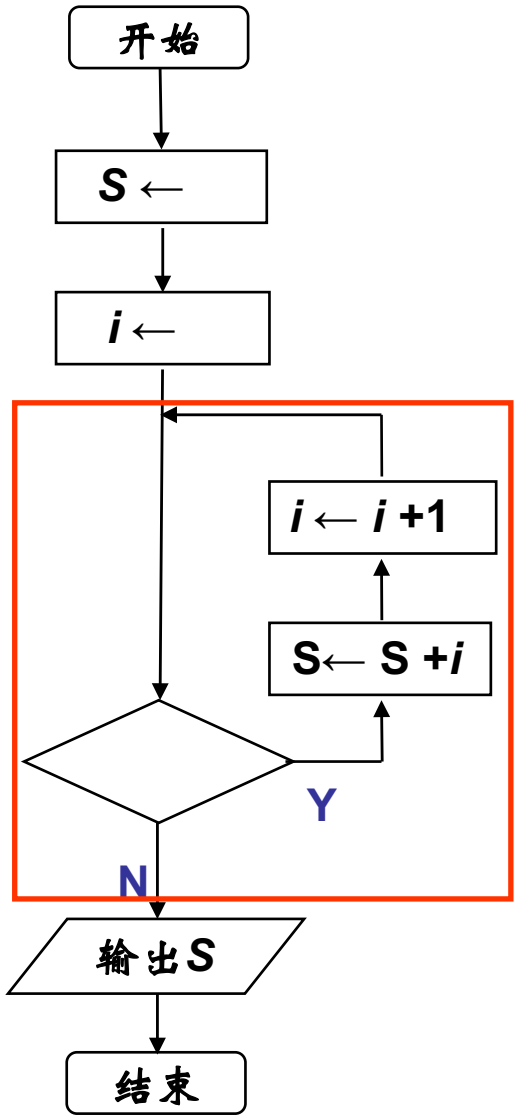


直到型语句：

```
Do
    循环体
Until
    p
End
Do
```

伪代码中的： **循环语句**

$S=1+2+3+\cdots+100$



自然语言—当型循环，先累加后计数：

S1 S←0 ；

S2 *i* ←1 ；

S3 当i≤100时，

S←*S*＋*i*;

i←*i*＋1 ;

转S3;

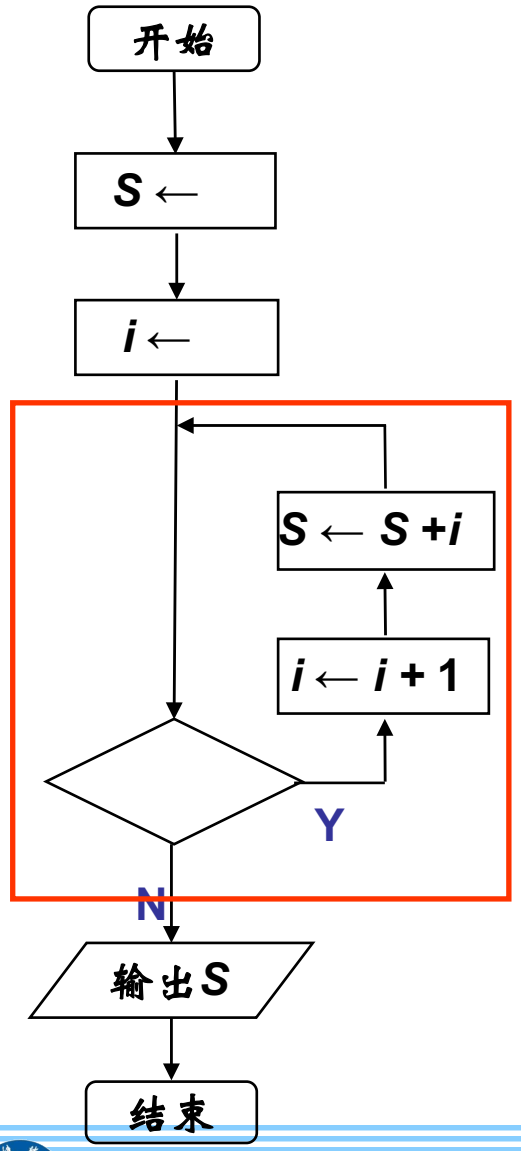
S4 输出S.

当型循环语句伪代码格式：

```
While P
    循环体
End While
```

```
S←0
i ←1 ;
While i≤100
    S←S＋i
    i←i＋1
End While
Print S
```

$S=1+2+3+\cdots+100$



自然语言—当型循环, 先计数后累加：

S1　S←0　;

S2　*i* ←0　;

S3　当i≤99时，

　　i←*i*＋1 ;

　　S←*S*＋*i*;

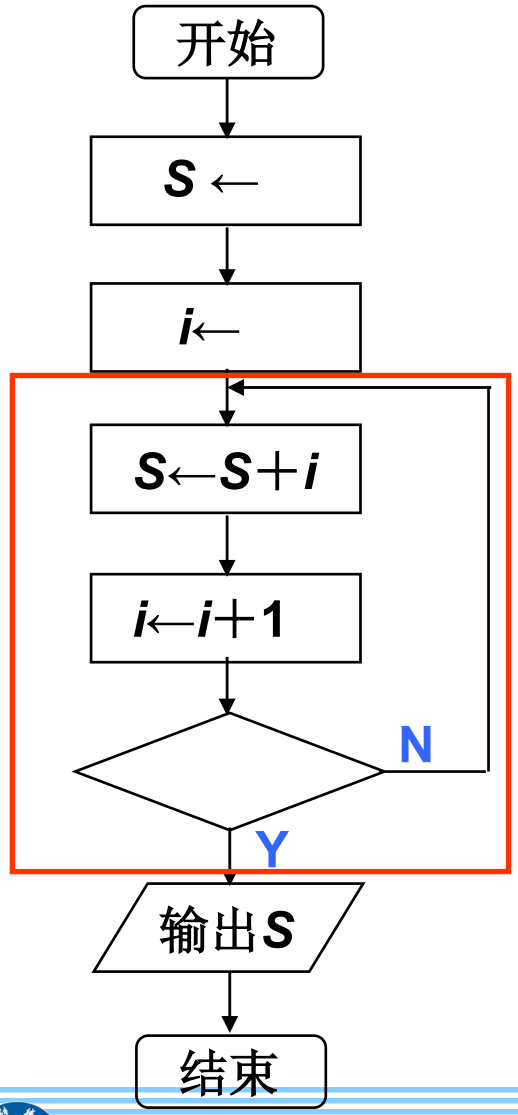
　转S3;

S4　输出S.

当型循环流程图和伪代码条件的一致性.

```
S←0
i ←0 ;
While i≤99
    i←i＋1
    S←S＋i
End while
Print S
```

$S=1+2+3+\cdots+100$



自然语言—直到型循环先累加后计数：

S1 S←0 ；

S2 *i* ←1 ；

S3 S←S+*i*;

S4 *i*←*i*+1 ;

S5 如果i不大于100，

转S3;

S6 输出S.

直到型循环语句伪代码格式：

Do

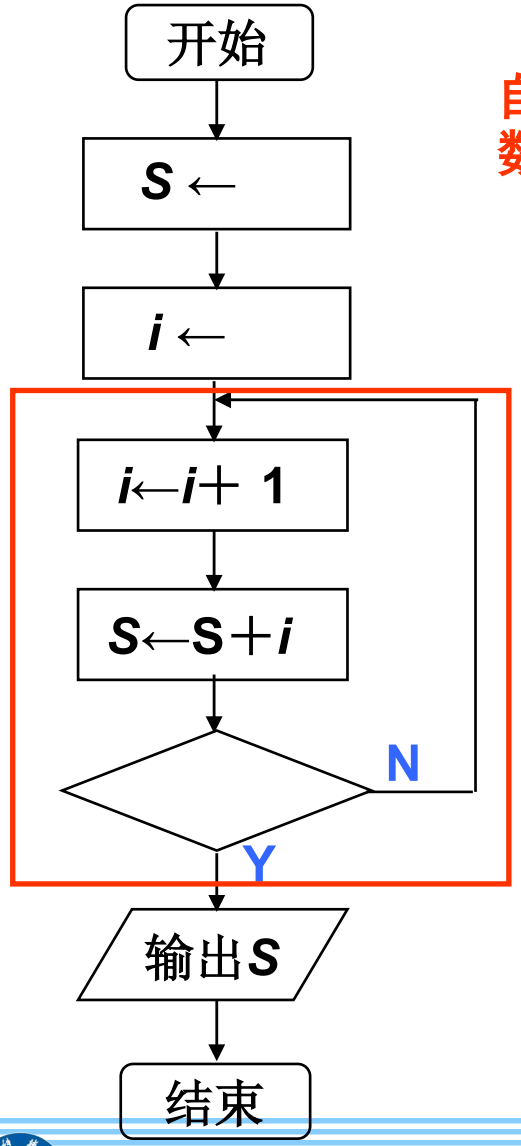
循环体

Until P

End Do

```
S←0
i ←1 ;
Do
    S←S+i
    i←i+1
Until  i >100
End Do
Print S
```

$S=1+2+3+\cdots+100$



自然语言—直到型循环先计数后累加：

S1 S←0 ;

S2 *i* ←1 ;

S3 *i*←*i*+1 ;

S4 *S*←*S*+*i;*

S5 如果i不大于99，

转S3;

S6 输出S.

直到型循环流程图和伪代码条件的一致性.

```
S←0
i ←1 ;
Do
    i←i+1
    S←S+i
Until i >99
End Do
Print S
```

设计计算1×3×5×7×…×99的一个算法，并写出伪代码

解： 算法如下：

S1 T←1；

S2 I←1；

S3 若I≤50，则转S4，否则转S6；

S4 T←T×(2I-1)；

S5 I←I+1，转S3；

S6 输出T.

当型循环：

当型语句如下：

T←1

I←1

While I≤50

T←*T*×(2*I*-1)

I←*I*＋1

End while

Print *T*

例下列伪代码实现的什么算法?

```
S←0
a←1
i←1
While i≤101
    S←S+a×i
    a←a×(−1)
    i ← i+2
End While
Print  S
```

1−3+5−7+9−…−101

请大家仔细观察上面写算法的几个问题，

他们的结构有什么特点？

他们的循环的次数已经确定。

当循环的次数已经确定，可用“For”语句表示.

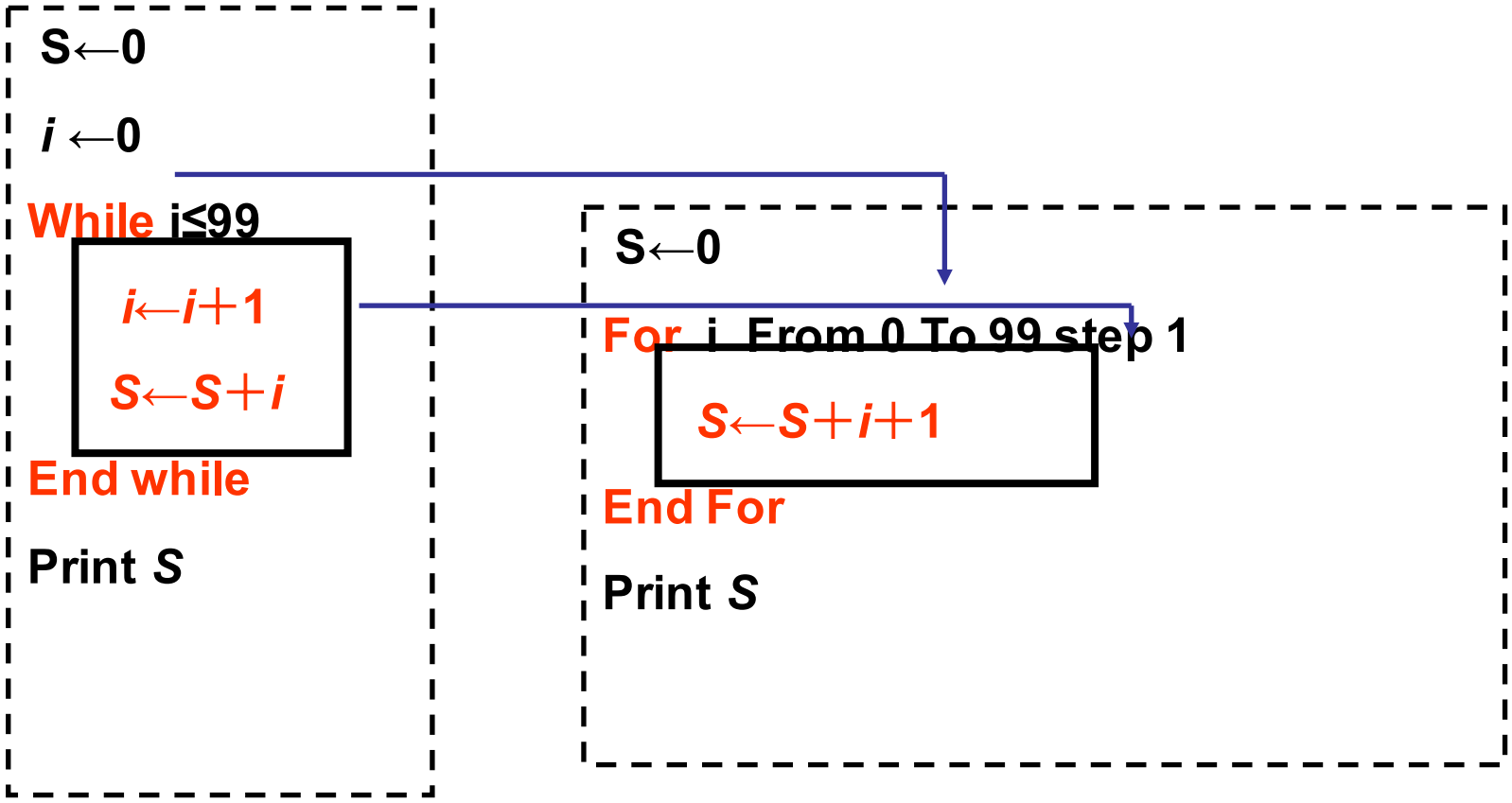
“For”语句伪代码格式：

For I From “初值” To “终值” step “步长”

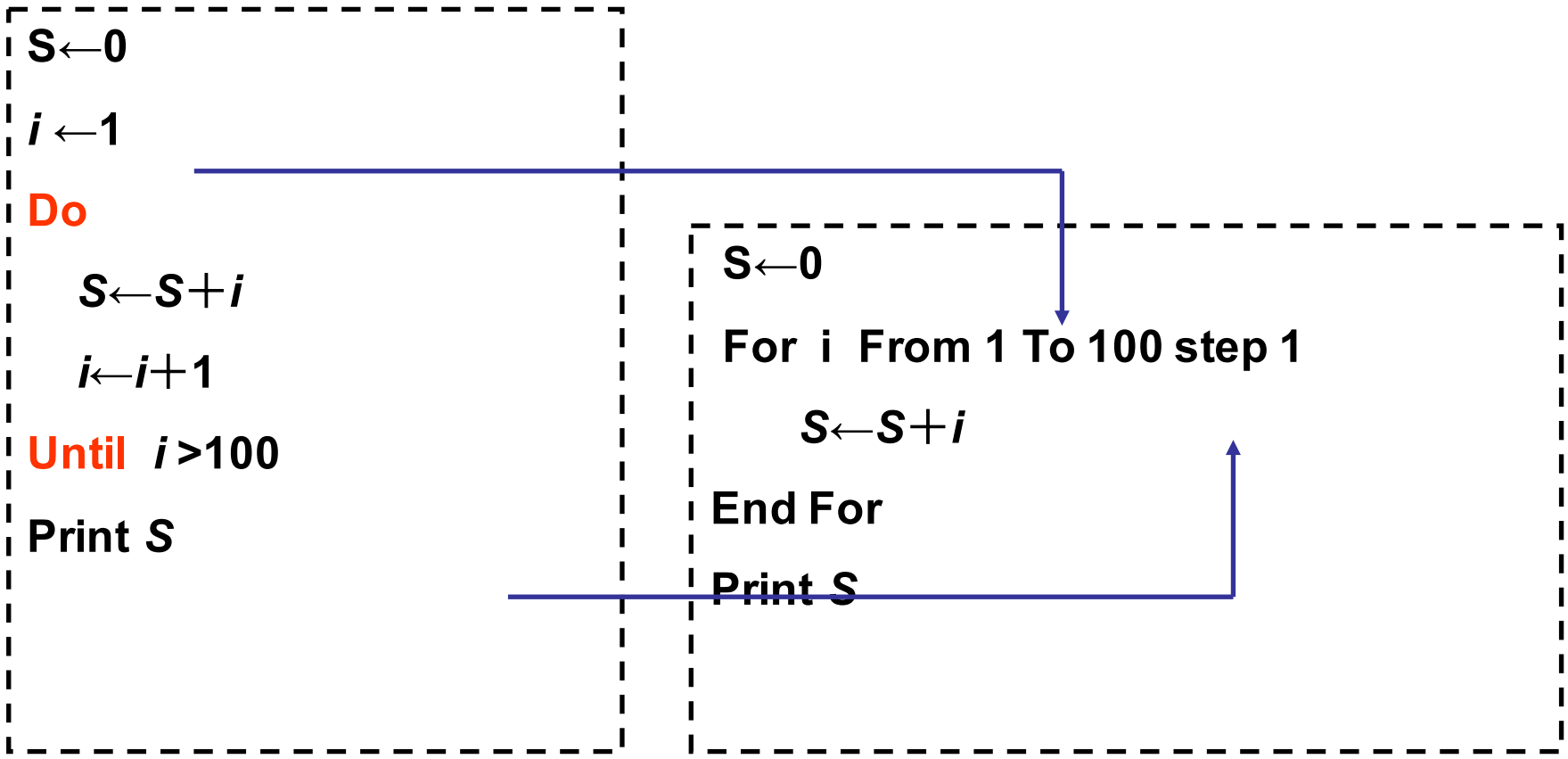
……

End For

例下列伪代码实现的什么算法?如何用For语句改写该算法？

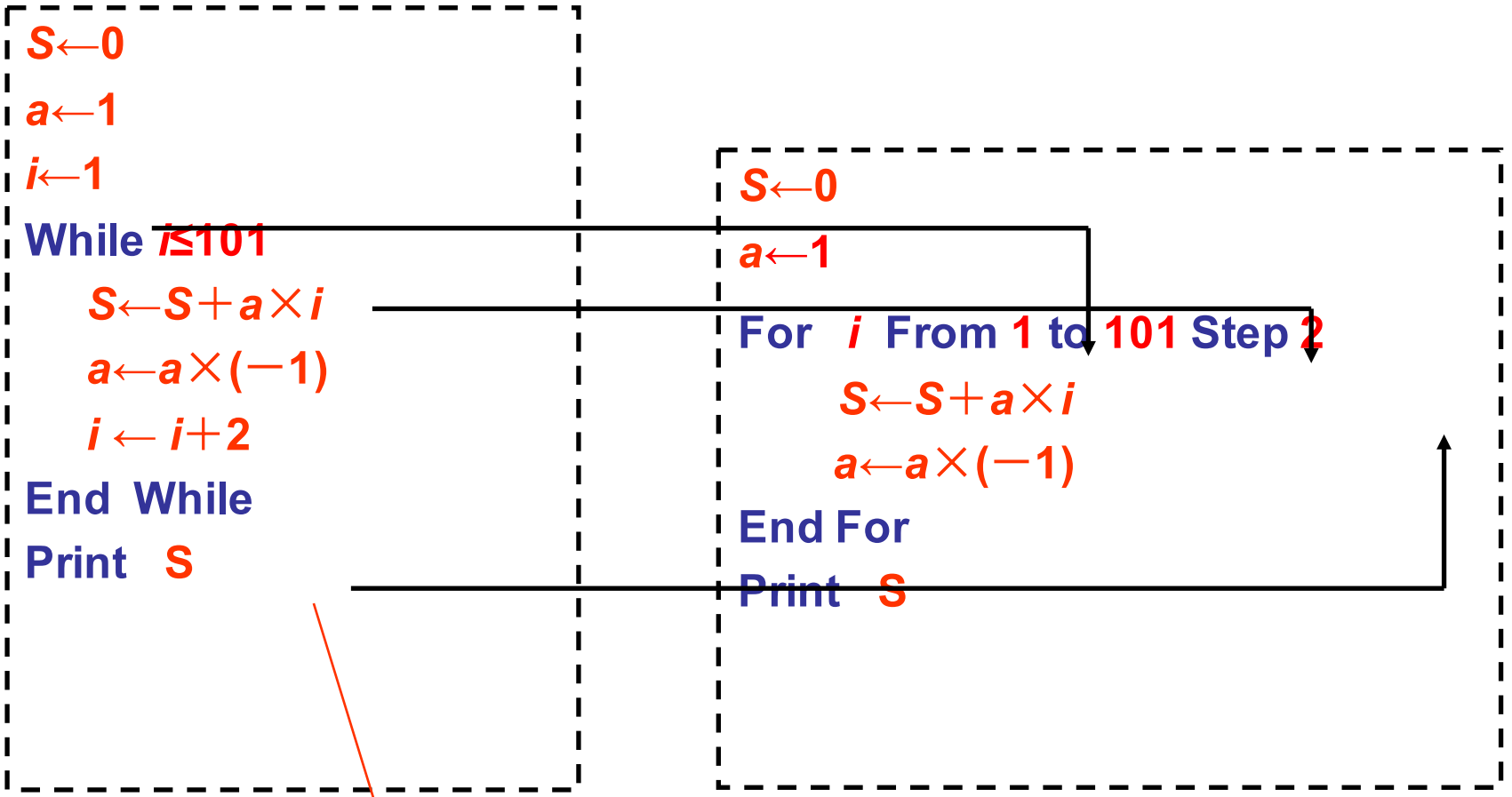


例下列伪代码实现的什么算法?如何用For语句改写该算法？



如何将直到型循环和当型循环的循环语句改为For语句？

例下列伪代码实现的什么算法?如何用For语句改写该算法？



1－3＋5－7＋9－…－101

i从1到101,每次增加2

实例1：求最大公约数

- 问题描述：求两个正整数 m 和 n 的最大公约数
- 自然语言描述：

while m is greater than zero:
    if n is greater than m, swap m and n.
    subtract n from m.
n is the GCD

C语言描述

```
int gcd(int m, int n)
{
    while( m > 0 ) {
        if( n > m ) {
            int t = m;
            m = n;
            n = t;
        }
        m -= n;
    }
    return n;
}
```

伪代码描述

GCD (m, n)

▷ 求取 m,n 的最大公约数

while $m>0$ **do**

if $n>m$ **then** $m \leftrightarrow n$ **end**

$m \leftarrow m - n$

end

n 为最大公约数

伪代码描述

- 原理：gcd(m,n) = gcd(n, m mod n)
- 结果：当n为0时,两数的最大公约数即为m

EUCLID-GCD (m, n)

▷ 求取 m,n 的最大公约数

while $n>0$ **do**

if $m<n$ **then** $m \leftrightarrow n$ **end**

$n \leftarrow m$ **mod** n

end

m 为最大公约数

实例2：牛顿逼近法求平方根

- 问题描述：求$\sqrt{x}$ 那样的 y，y>=0且$y^2 = x$
  - 数学描述 – 求解过程
  - 描述性 – 命令式
- 自然语言描述：
  - 如果已有一个猜测y，那么可以得到一个更好的猜测，即 x/y 与 y 的平均值，直到这个猜测y足够好。

X = 2	G = 1
X/G = 2	G = ½ (1+ 2) = 1.5
X/G = 4/3	G = ½ (3/2 + 4/3) = 17/12 = 1.416666
X/G = 24/17	G = ½ (17/12 + 24/17) = 577/408 = 1.4142156

伪代码描述

SQRT (x, g)

▷ 求取 x的平方根, g为第一个猜测值

do

$G \leftarrow g$

$g \leftarrow (x/G + G)/2$

until GOOD-ENOUGH (G, g)

G为x的平方根

伪代码描述

function POLICY-ITERATION(M, R) **returns** a policy
  inputs: M, a transition model
    R, a reward function on states
  local variables: U, a utility function, initially identical to R
    P, a policy, initially optimal with respect to U

  repeat
    $U \leftarrow$ VALUE-DETERMINATION(P, U, M, R)
    unchanged? $\leftarrow$ true
    for each state i **do**
      if $\max_a \sum_j M_{ij}^a U[j] > \sum_j M_{ij}^{P[i]} U[j]$ **then**
        $P[i] \leftarrow \arg\max_a \sum_j M_{ij}^a U[j]$
        unchanged? $\leftarrow$ false
    end
  until *unchanged?*
  return P

Reinforcement Learning

伪代码描述

输入：训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
　　基学习算法 $\mathfrak{L}$;
　　训练轮数 T.

过程：

1: $\mathcal{D}_1(\boldsymbol{x}) = 1/m$.
2: **for** $t = 1, 2, \ldots, T$ **do**
3: 　$h_t = \mathfrak{L}(D, \mathcal{D}_t)$;
4: 　$\epsilon_t = P_{\boldsymbol{x} \sim \mathcal{D}_t}(h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}))$;
5: 　**if** $\epsilon_t > 0.5$ **then break**
6: 　$\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$;
7: 　$\mathcal{D}_{t+1}(\boldsymbol{x}) = \frac{\mathcal{D}_t(\boldsymbol{x})}{Z_t} \times \begin{cases} \exp(-\alpha_t), & \text{if } h_t(\boldsymbol{x}) = f(\boldsymbol{x}) \\ \exp(\alpha_t), & \text{if } h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}) \end{cases}$
　　$= \frac{\mathcal{D}_t(\boldsymbol{x}) \exp(-\alpha_t f(\boldsymbol{x}) h_t(\boldsymbol{x}))}{Z_t}$
8: **end for**

输出：$H(\boldsymbol{x}) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x}) \right)$

图 8.3　AdaBoost算法

算法的效率分析

Linear Search Algorithm

- Algorithm
  1. Guess number = 1
  2. If incorrect, increment guess by 1
  3. Repeat until correct

- Example
  - Given number between 1…100
  - Pick 20
  - Guess sequence = 1, 2, 3, 4 … 20
  - Required 20 guesses

Binary Search Algorithm

- Algorithm
  1. Set Δ to n/4
  2. Guess number = n/2
  3. If too large, guess number – Δ
  4. If too small, guess number + Δ
  5. Reduce Δ by ½
  6. Repeat until correct

Binary Search Algorithm

- Example
  - Given number between 1…100
  - Pick 20
  - Guesses =
    - 50, Δ = 25, Answer = too large, subtract Δ
    - 25, Δ = 12 , Answer = too large, subtract Δ
    - 13, Δ = 6, Answer = too small, add Δ
    - 19, Δ = 3, Answer = too small, add Δ
    - 22, Δ = 1, Answer = too large, subtract Δ
    - 21, Δ = 1, Answer = too large, subtract Δ
    - 20, Yeah, you got it!
  - Required 7 guesses

Search Comparison

- Linear search algorithm
  - For number between 1…100: 50 steps
  - For number between 1…100,000: 50,000 steps
- Binary search algorithm
  - For number between 1…100: $\log_2(n) = 7$ steps
  - For number between 1…100,000: $\log_2(n) = 77$ steps
- Binary search is much more efficient!

算法的运行时间分析

- 算法的时间是每语句执行时间的总和
  - 每语句的执行时间=该语句执行次数（频度）
  - X该语句执行1次的时间
  - 假定：每语句执行1次的时间为1个时间单位
  - 则：算法的执行时间=∑各语句频度
- 问题的规模（Size）：n
  - 输入量的大小，如…
- 时间复杂度：算法的运行时间，是问题规模的函数

Analyzing Algorithms

- Goal
  - Find asymptotic complexity of algorithm
- Approach
  - Ignore less frequently executed parts of algorithm
  - Find critical section of algorithm
  - Determine how many times critical section is executed as function of problem size

Summarization Example

- Sum from 1 to n
  - Sum (for input size n)
  1. *A:* S = 0
  2. for (int i = 1; i <= n; i++)
  3. *B:* S = S + i
  4. *C:* Return S
- Code execution
  - A ⇒
  - B ⇒
  - C ⇒
- Time ⇒

Summarization Example

- Sum from 1 to n
  - Sum (for input size n)

1. A: S = 0
2. for (int i = 1; i <= n; i++)
3. *B:* S = S + n
4. C: Return S

critical section

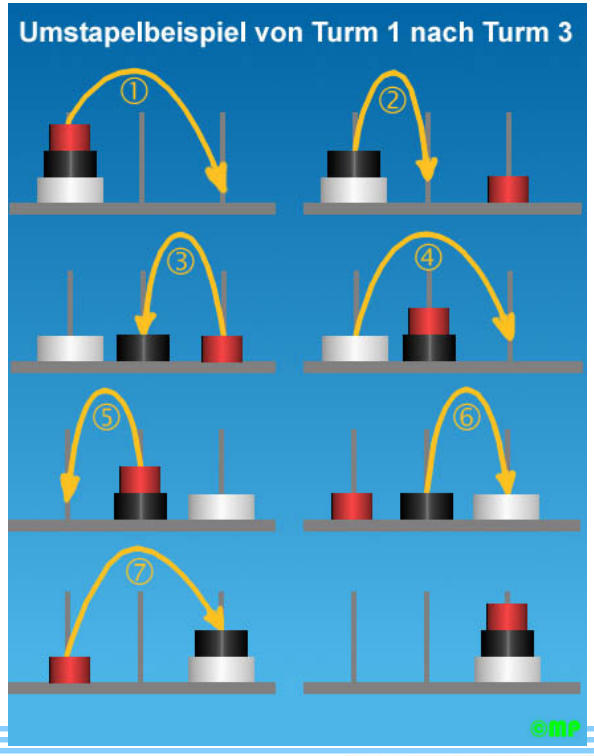
- Code execution
  - A ⇒ once
  - B ⇒ *n* times
  - C ⇒ once

S = n * (n + 1) / 2 ⇒ O(1)

- Time ⇒ 1 + n + 1 ⇒ O(n)

Hanoi Example

- Solve recursively
  - Move upper *n*-1 disks to Turm 2
  - Move the *n*th disk to Turm 3
  - Move the *n*-1 disks to Turm 3
- Steps
  - h(*n*) = 2h(*n*-1) + 1
  - h(1) = 1
- h(*n*) = 2^n - 1
  - $O(2^n)$



时间复杂度计算原则

- As n increases
  - Highest complexity term dominates
  - Lower complexity terms are ignored
- Examples
  - $2\,n + 100$ $\Rightarrow O(n)$
  - $n \log(n) + 10\,n$ $\Rightarrow O(n\log(n))$
  - $\frac{1}{2}\,n^2 + 100\,n$ $\Rightarrow O(n^2)$
  - $n^3 + 100\,n^2$ $\Rightarrow O(n^3)$
  - $1/100\,2^n + 100\,n^4$ $\Rightarrow O(2^n)$

渐近时间复杂度

- 从宏观上评价时间性能，只关心当n趋向无穷大时，T(n)的数量级(阶)

- 大O的数学定义

若T(n)和f(n)是定义在正整数集合上的两个函数，则T(n)=O(f(n)) 表示存在两个正的常数c和n_0，

例 :使得当$n \geq n_0$时都满足$0 \leq T(n) \leq c \cdot f(n)$.

$f(n)=3n+2=O(n)$　　因为 $3n+2 \leq 4n$　for $n \geq 2$

$f(n)= 6*2^n+n^2=O(2^n)$　因为$6*2^n+n^2 \leq 7*2^n$　for $n \geq 4$

渐近复杂度分析

- 时间复杂度是以算法中基本操作重复执行的次数的数量阶（级）作为算法的时间度量

常数阶	对数阶	线性阶	线性对数阶	平方阶	立方阶	…	K次方阶	指数阶
$O(1)$	$O(\log_2 n)$	$O(n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n^3)$		$O(n^k)$	$O(2^n)$

注：

1）O（）为渐近符号。

2）空间复杂度S(n)按数量级递增顺序也与上表类似

3）实际设计中，常常需要在时间复杂度与空间复杂度中寻求一种折衷（时空折衷）

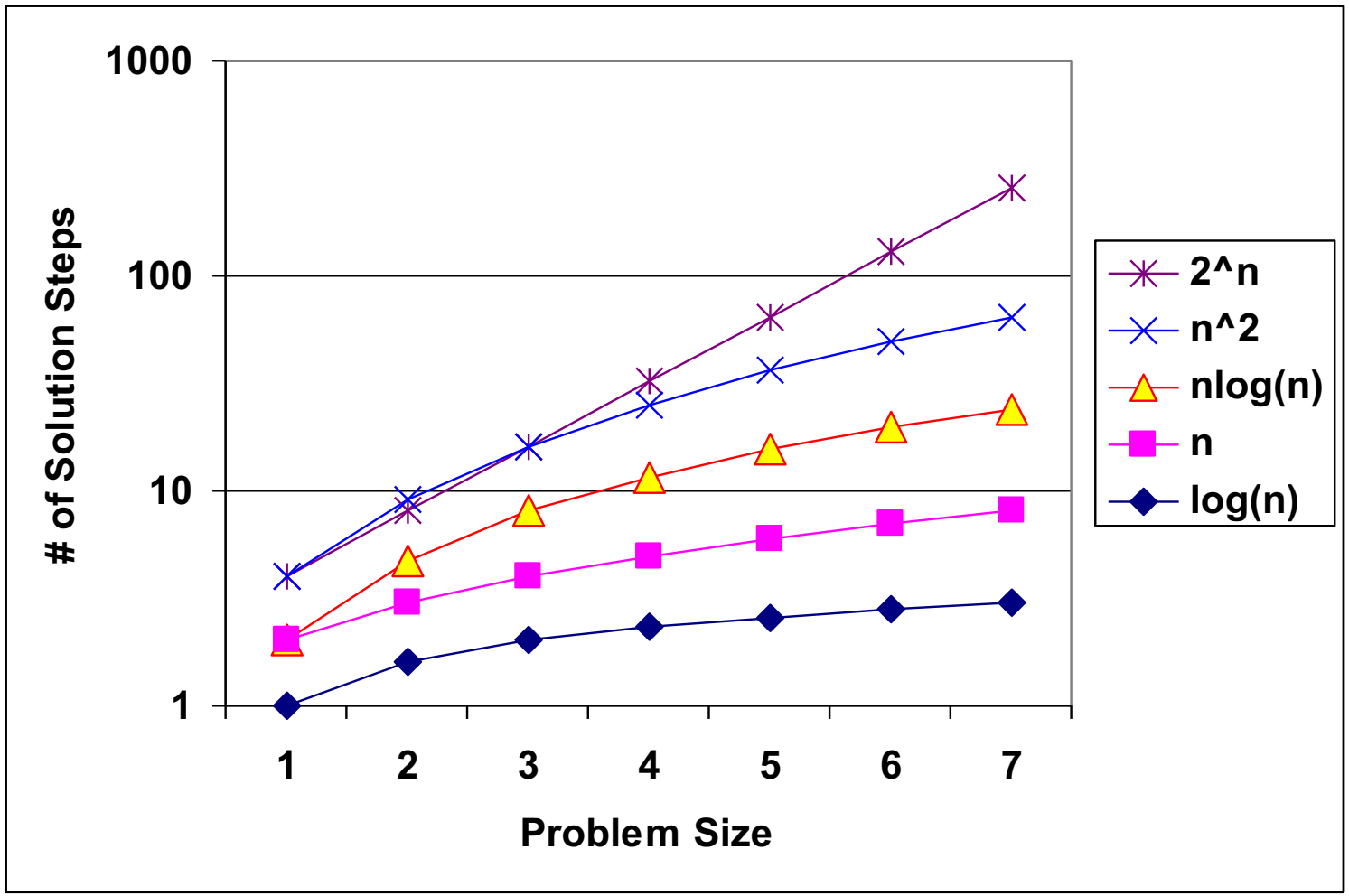
Asymptotic Complexity

Complexity	Name	Example
– O(1)	Constant	Array access
– O(log(n))	Logarithmic	Binary search
– O(n)	Linear	Largest element
– O(n log(n))	N log N	Optimal sort
– $O(n^2)$	Quadratic	2D Matrix addition
– $O(n^3)$	Cubic	2D Matrix multiply
– $O(n^k)$	Polynomial	Linear programming
– $O(k^n)$	Exponential	Integer programming

From smallest to largest

For size n, constant k > 1

Complexity Category Example



小结

- 算法
  - 对特定问题求解步骤的一种描述，它是指令的有限序列，是一系列输入转换为输出的计算步骤
- 算法是程序的灵魂
- 描述方式
  - 自然语言、流程图、N-S图、伪代码、程序语言
- 算法的复杂度（时空分析）
  - 时间复杂度
  - 空间复杂度

Extend Reading

Introduction to Algorithms

- Stu…aduate, graduate
- Co…re, ... in c…
- Bibliography
  - 《Introduction to Algorithms》(Second Edition),
    T. H. Cormen, C. E. Leiserson, R. L. Rivest (2002, Turing Award),
    The MIT Press
  - 《The Art of Computer Programming》, Donald E, Knuth
    1974, Turing Award

网友：“没有读过《Intro…》，不能算是一个真正的程序员”

“计算机算法的圣经”

“计算机程序设计理论的荷马史诗”

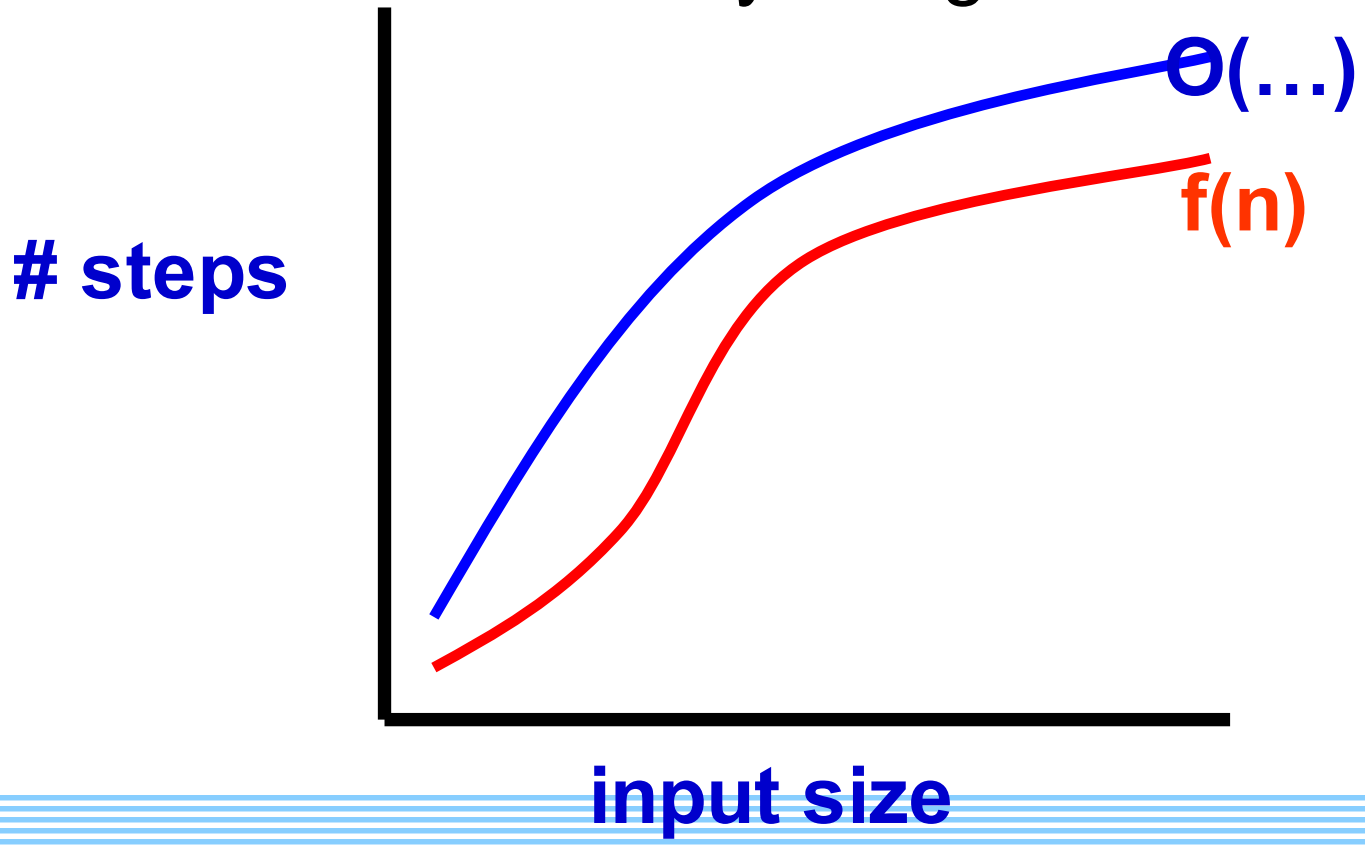
Bill Gates: “如果你认为你是一名真正优秀的程序员，请读Knuth的《计算机程序设计艺术》，如果你能读懂整套书的话，请给我发一份你的简历。”

思考与实践

- 二分查找的流程图或N-S图表示
- 用伪代码描述判断闰年的算法

Big-O Notation

- Represents
  - Upper bound on number of steps in algorithm
  - Intrinsic efficiency of algorithm for large inputs



Formal Definition of Big-O

- Function f(n) is O(g(n)) if
  - For some positive constants M, N_0
  - $M \times g(n) \geq f(n)$, for all $n \geq N_0$
- Intuitively
  - For some coefficient M & all data sizes $\geq N_0$
    - $M \times g(n)$ is always greater than f(n)
- Big-O: asymptotic complexity(渐近复杂度)