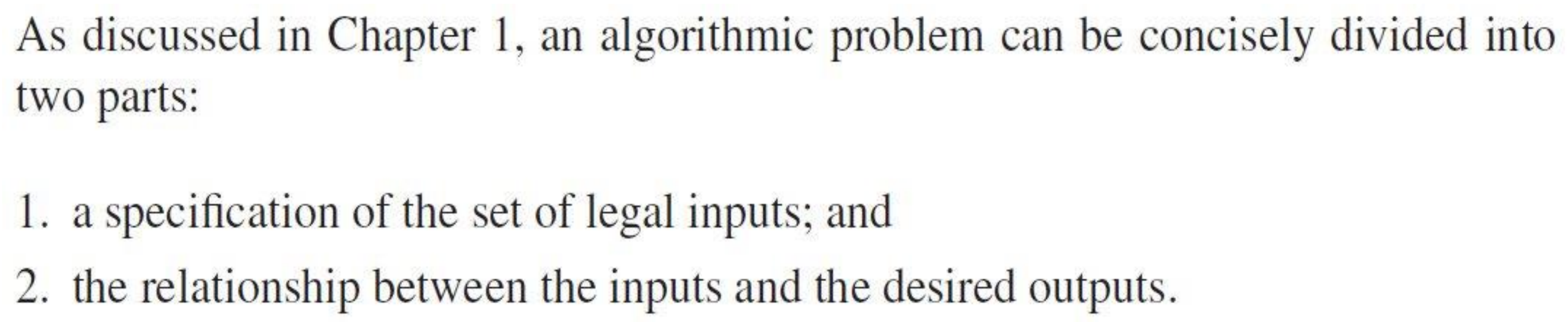
计算机问题求解 – 论题2-2 - 算法的正确性

2015年3月03日

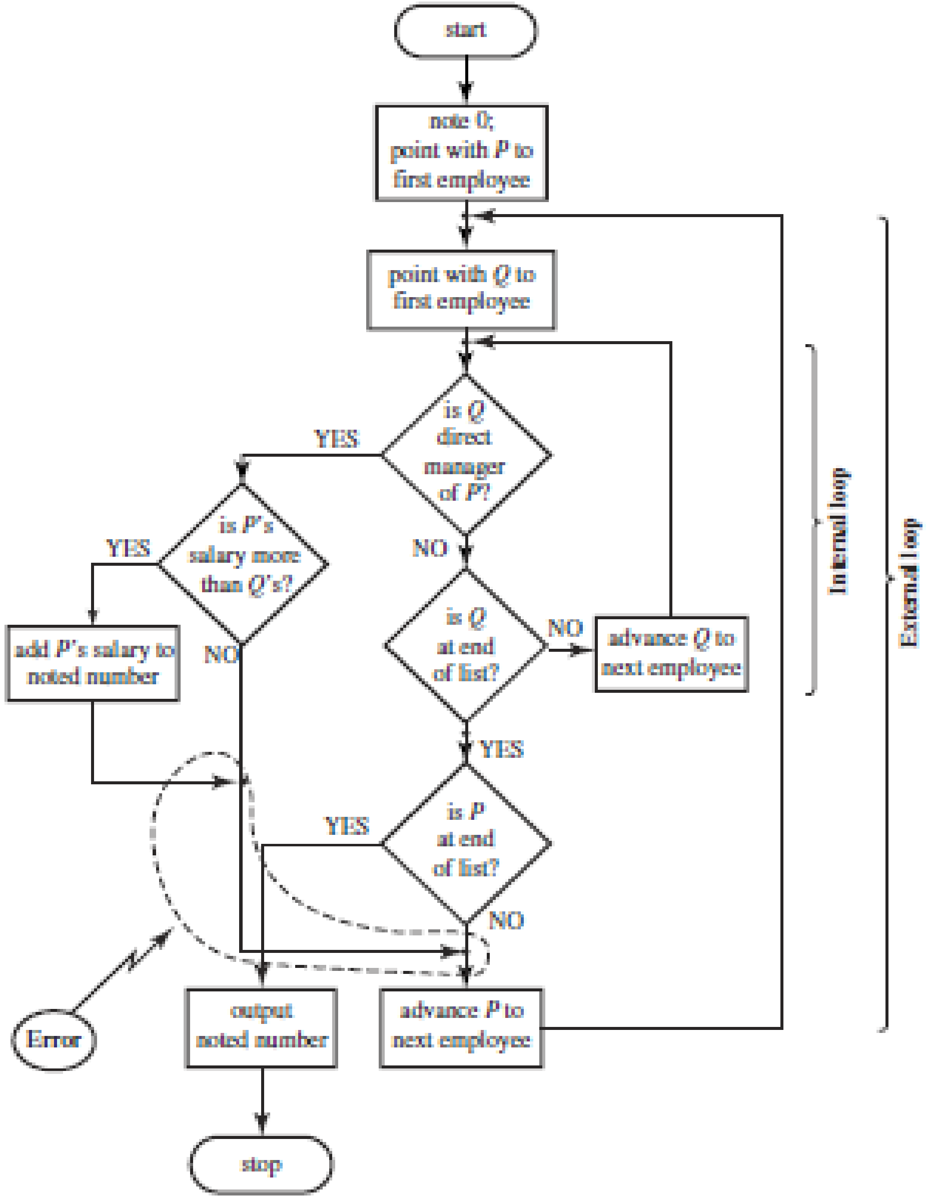

As discussed in Chapter 1, an algorithmic problem can be concisely divided into two parts:

1. a specification of the set of legal inputs; and
2. the relationship between the inputs and the desired outputs.

这段话和我们今天的主题什么关系？

问题1:

人们似乎对于计算机“犯错”比对于人犯错更加苛刻,这是为什么呢?



问题2:

这个错误可能的后果是什么?为什么这样的错比“语言错”更严重?

问题3：

书上的对文本中出现“money”一词的句子计数的例子出现了什么样的错误，你能说出它的性质与前一个例子有什么不同吗？

几种不同的错误

- “语言错”
  - 很容易被语言处理软件发现，甚至自动纠正。
- “语义错”
  - 合格的程序员很少会犯这类错误。
- 解题逻辑错误
  - “粗心”造成的错误。
    - 常见(相对来说)
  - “真正的”逻辑错误。

    不常见，但这个是真的“伤脑筋”！

问题4：

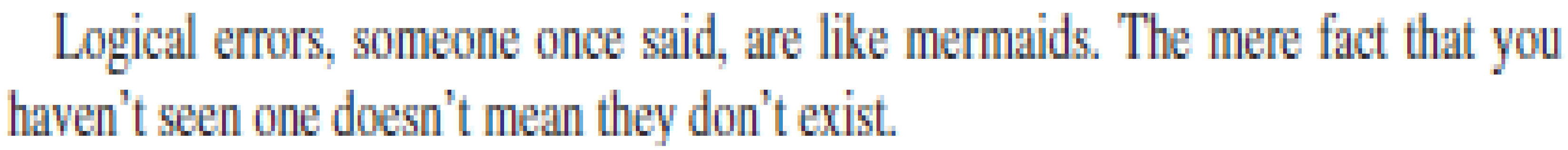
“Computers Do Not Err!”

你如何理解这句话？

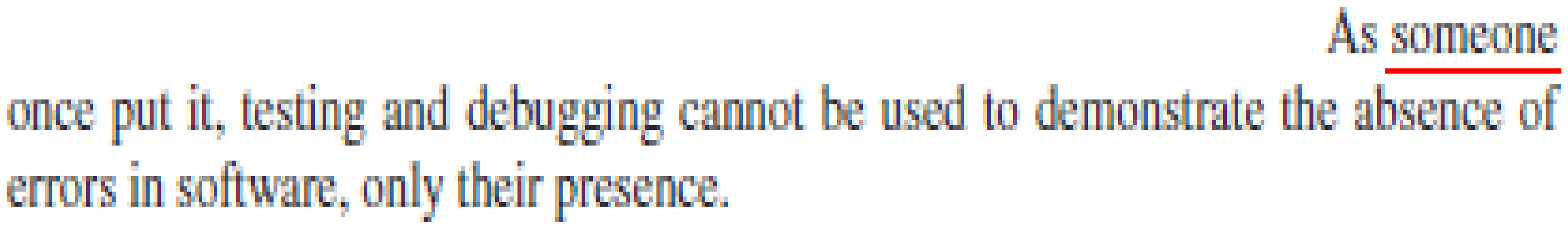
Test and debugging

- A designer might try out an algorithm on several typical and atypical inputs and not find the error. In fact, a programmer will normally test a program on numerous inputs, sometimes called **test sets**, and will gradually rid it of its language errors and most of its logical errors.
- Most algorithmic problems have infinite sets of legal inputs, and hence infinitely many candidate test sets, each of which has the potential of exposing a new error.

Debugging 的局限性



Logical errors, someone once said, are like mermaids. The mere fact that you haven't seen one doesn't mean they don't exist.



As someone once put it, testing and debugging cannot be used to demonstrate the absence of errors in software, only their presence.

"someone" believed to be Dijkstra:

Program testing can be used to show the presence of bugs, but never to show their absence!

Also, Dijkstra believed to say:

Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had the better remain pure mathematicians.

关于debugging的思考

- 为什么我们可以：
- The process of repeatedly executing an algorithm, or running a program, with the intention of finding and eliminating errors is called **debugging**？

Infinite computation

为什么infinite computation 有时也被称为 infinite loop?

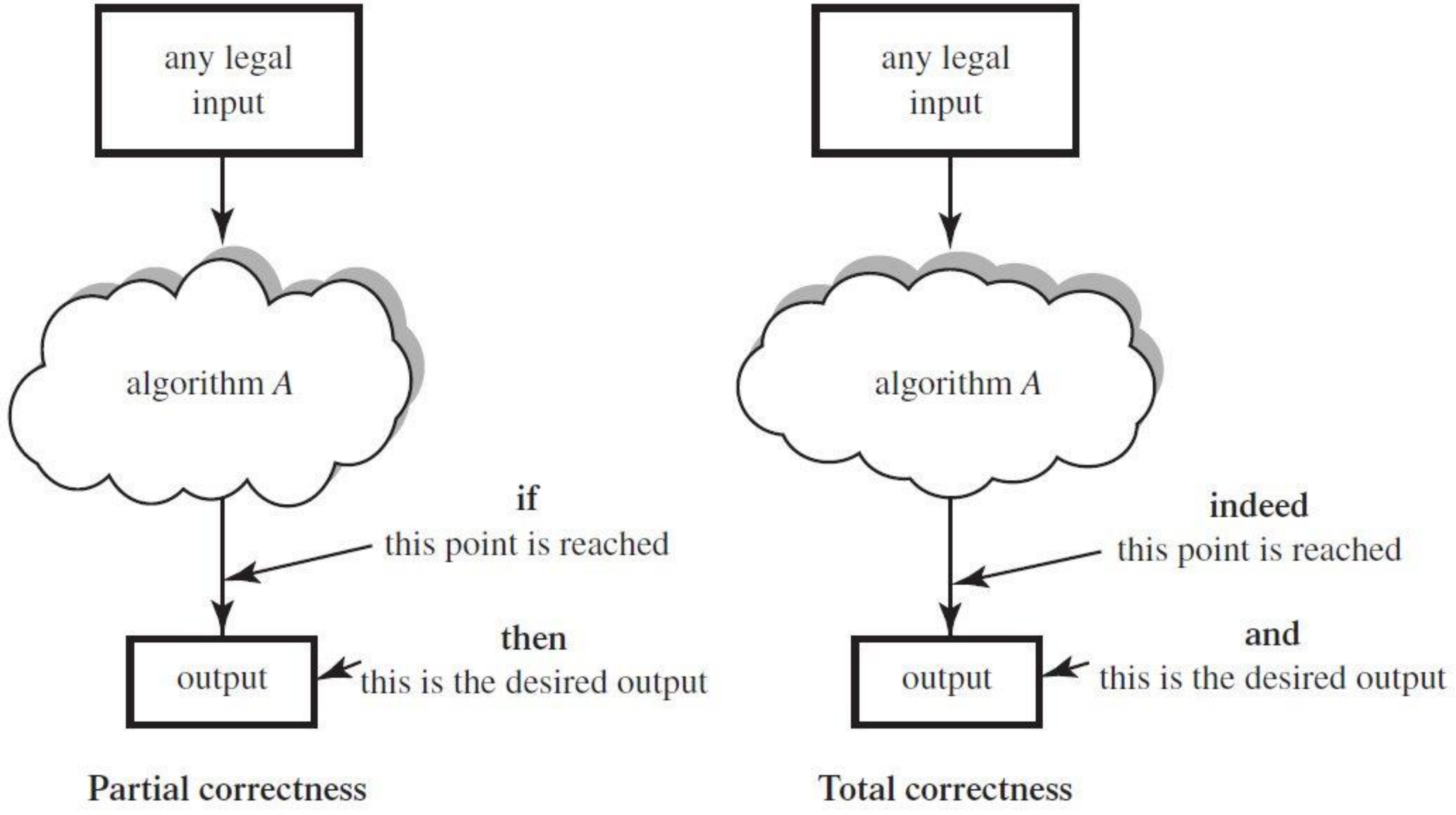
Infinite loop有什么作用？

如何避免不正确的Infinite loop出现？

As discussed in Chapter 1, an algorithmic problem can be concisely divided into two parts:

1. a specification of the set of legal inputs; and
2. the relationship between the inputs and the desired outputs.

部分和完全正确性

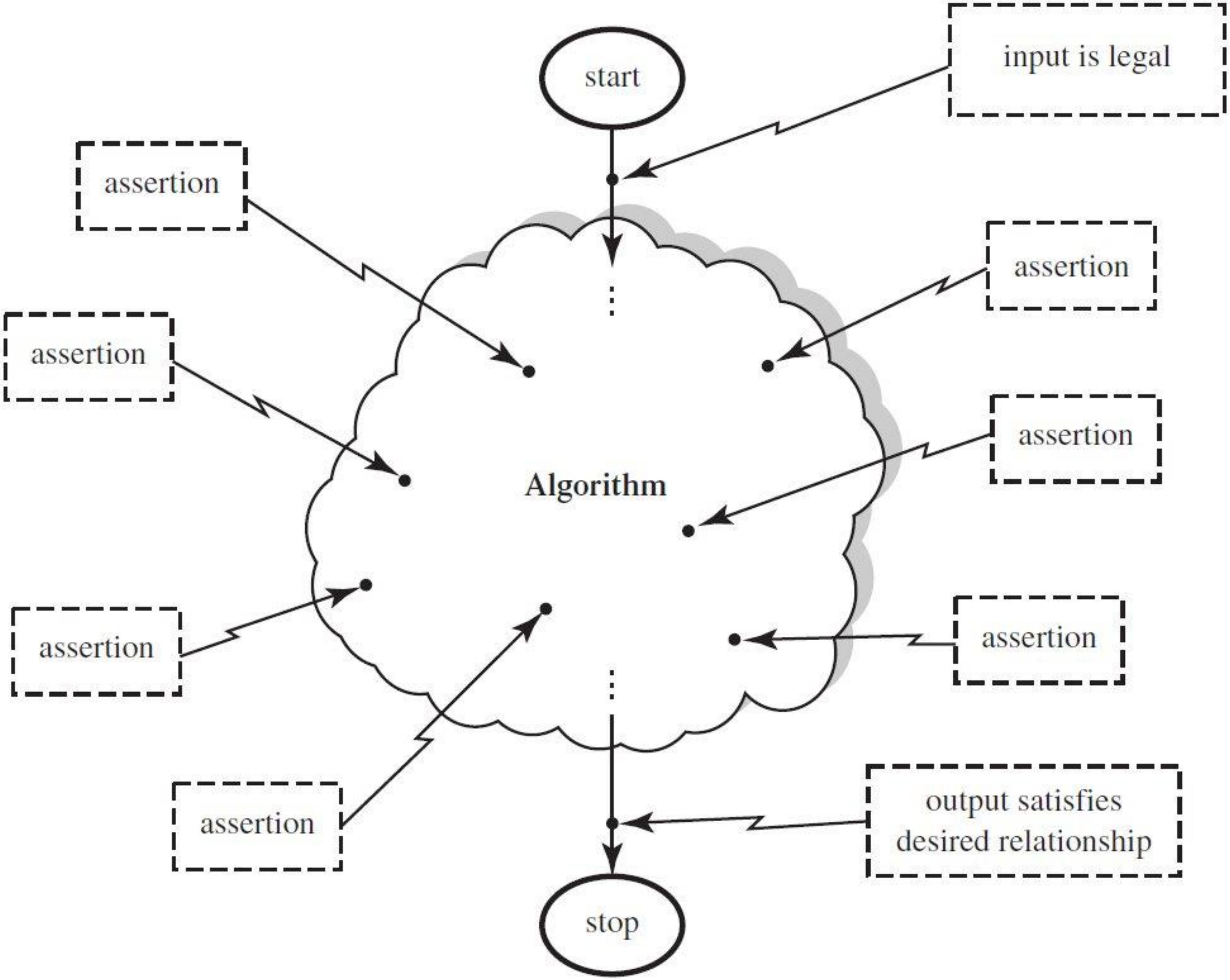

any legal input
algorithm A
if
this point is reached
then
this is the desired output
output
Partial correctness
any legal input
algorithm A
indeed
this point is reached
and
this is the desired output
output
Total correctness

算法正确性证明：

- 自动验证：
  - some sort of super-algorithm that would accept as inputs a description of an algorithmic problem *P* and an algorithm *A* that is proposed as a solution, and would determine if indeed *A* solves *P*.
- 人工证明：
  - Can we ourselves prove our algorithms to be correct? Is there any way in which we can use formal, mathematical techniques to realize this objective?

如何利用什么数学技术来证明算法的正确性？

- “部分正确性”：
  - We do not care whether execution ever reaches the endpoint, but that if it does we will not be in a situation where the outputs differ from the expected ones.
  - we wish to capture the behavior of the algorithm by making careful statements about what it is doing at certain points.
  - we thus attach **intermediate assertions** to various **checkpoints** in the algorithm’s text.


start
input is legal
assertion
assertion
assertion
assertion
Algorithm
assertion
assertion
assertion
output satisfies desired relationship
stop

如何理解以下文字?

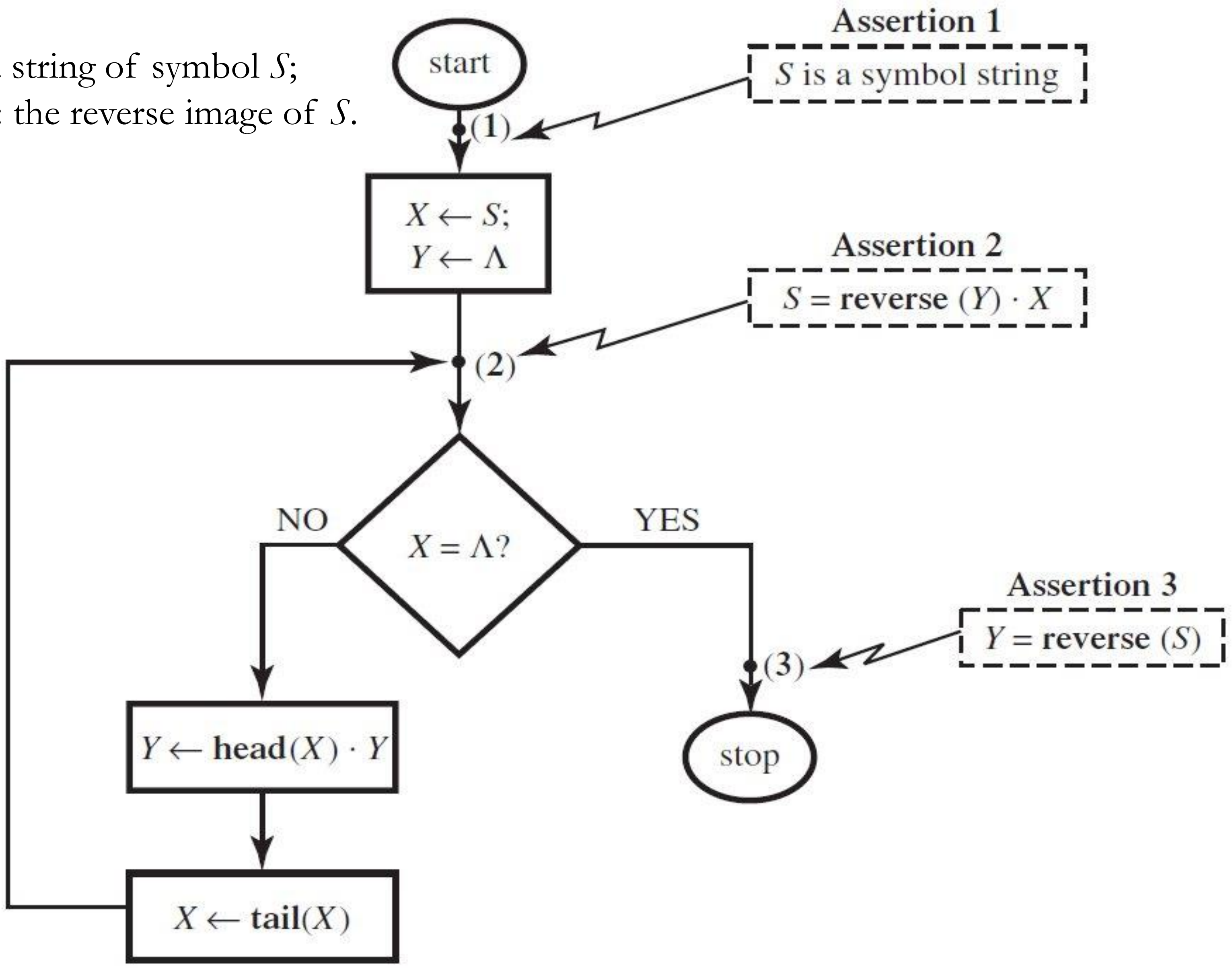
- Attaching an assertion to a checkpoint means that we believe that whenever execution reaches the point in question, in any execution of the algorithm on any legal input, the assertion will be true.

Intermediate Assertions at Checkpoints

问题：

Input: a string of symbol S;

Output: the reverse image of S.



问题6:

这些intermediate assertions为什么被称为“invariants”？

什么又叫循环不变式？

```
subroutine square of A:
    (1) set $C$ to 0;
    (2) set $D$ to 0;
    (3) while ($D \neq A$) do
            set $C$ to $C+A$;
            set $D$ to $D+1$;
    (4) return $C$.
```

这个过程计算*A*的平方。

问题7：

你能指出循环不变式是什么吗？

$$C_k = A \times D_k$$

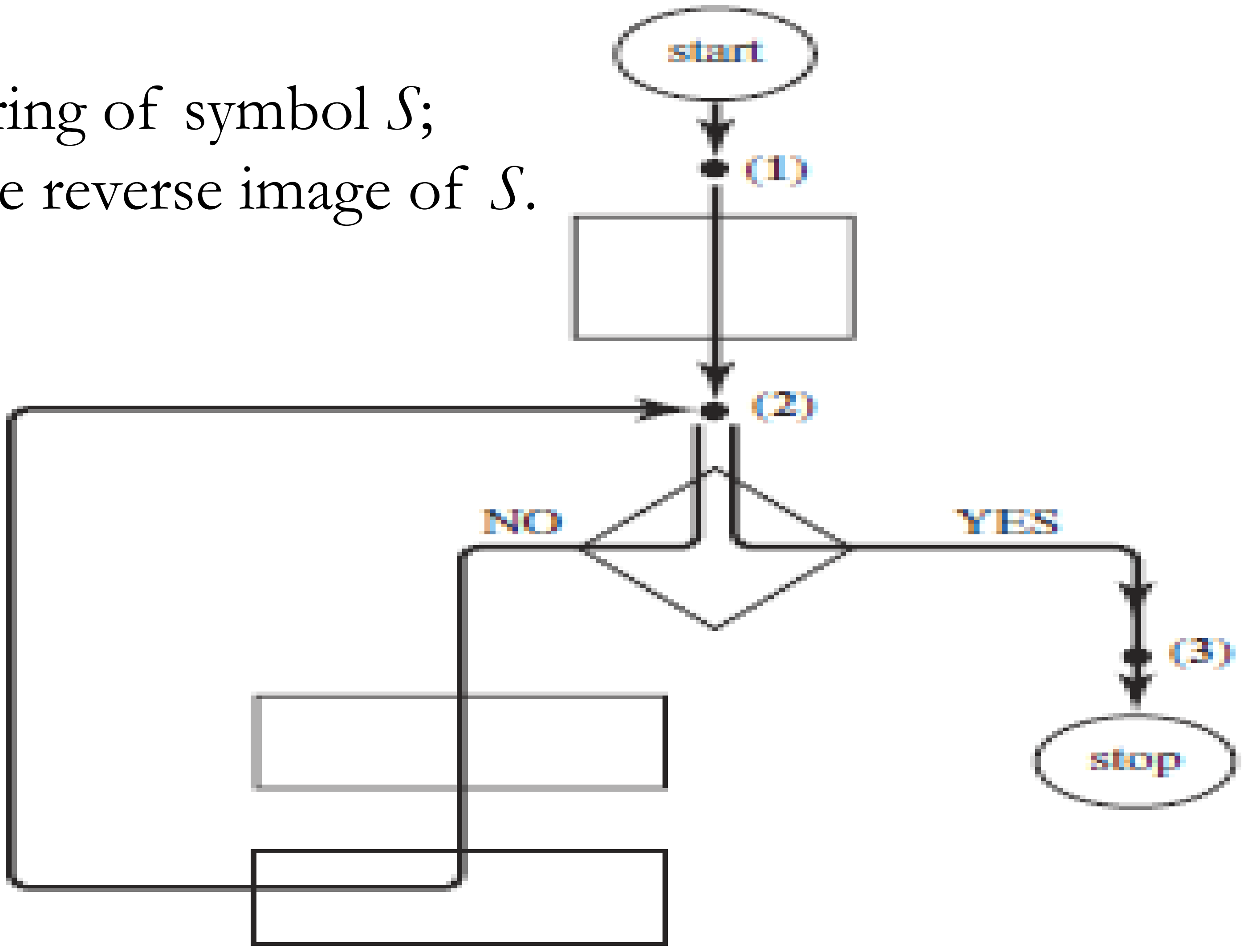
Then:

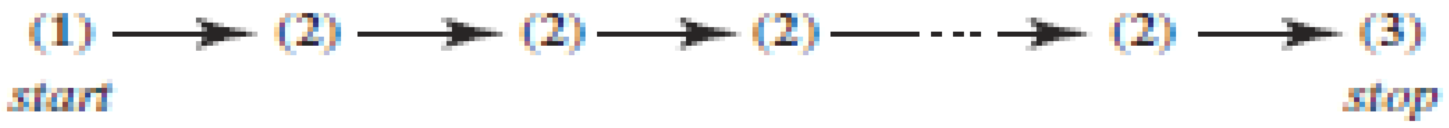
- 要证明一个算法的部分正确性：
  - 在哪里设置checkpoint?
  - 什么样的“局部特性”用断言形式描述?
  - proceeding locally from checkpoint to checkpoint does not bring about any violations of the invariance properties

问题：

Input: a string of symbol S;

Output: the reverse image of S.





我们可以/必须证明：

$(1 \to 2)$: for any string S, after carrying out the two instructions $X \leftarrow S; Y \leftarrow \Lambda$, the equality $S = \textbf{reverse}(Y) \cdot X$ will hold.

$(2 \to 3)$: if $S = \textbf{reverse}(Y) \cdot X$, and $X = \Lambda$, then $Y = \textbf{reverse}(S)$.

$(2 \to 2)$: if $S = \textbf{reverse}(Y) \cdot X$, and $X \neq \Lambda$, then after carrying out the instructions $Y \leftarrow \textbf{head}(X) \cdot Y; X \leftarrow \textbf{tail}(X)$, the same equality, namely $S = \textbf{reverse}(Y) \cdot X$, will hold for the new values of X and Y.

$(2 \to 2)$: if $S = \textbf{reverse}(Y) \cdot X$, and $X \neq \Lambda$, then $S = \textbf{reverse}(\textbf{head}(X) \cdot Y) \cdot \textbf{tail}(X)$.

收敛性：total正确性的证明方法

  - showing that something good eventually happens (not that bad things do not); namely, that the algorithm indeed reaches its endpoint and terminates successfully.
- find some quantity and show that it **converges:**
  - quantity keeps decreasing as execution proceeds from one checkpoint to another, but that it cannot decrease forever: there is some bound below which it can never go

“部分”与“完全”正确性

- **Euclid algorithm**
  - input: nonnegative integer *m*,*n*
  - output: gcd(*m*,*n*)
  - procedure

```
Euclid(int m,n)
    if n=0
        then return m
        else return Euclid(n, m mod n)
```

1 **if *d* is the GCD of *m* and *n*, it must be theGCD of *n* and (*m* mod *n*)**

2 **(*m* mod *n*) is always less than *n*, so, the algorithm must terminate**

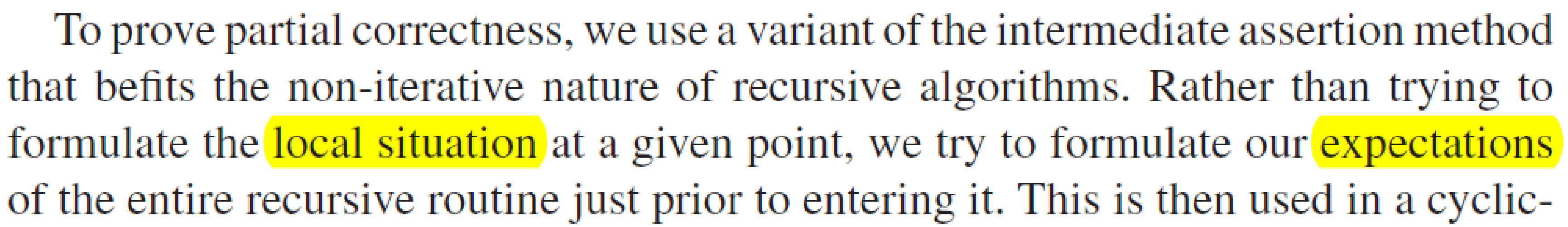
问题：

你能否用这个例子解释 Partial 和 Total正确性？

但是，针对递归算法：

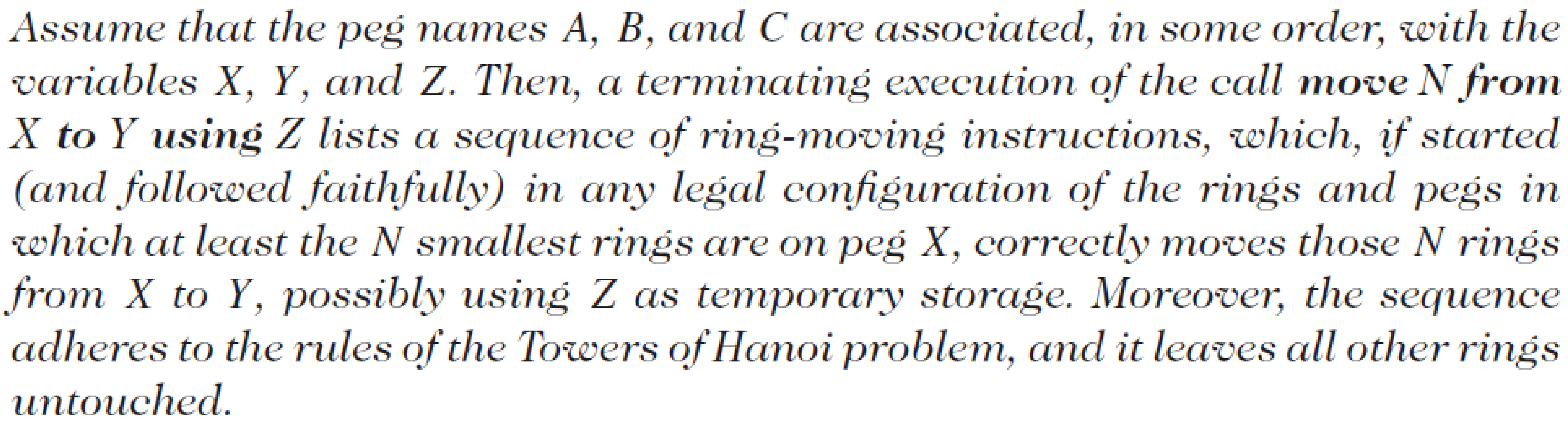
```
subroutine **move** $N$ **from** $X$ **to** $Y$ **using** $Z$:
(1) if $N$ is 1 then output "move $X$ to $Y$";
(2) otherwise (that is, if $N$ is greater than 1) do the following:
    (2.1) call **move** $N - 1$ **from** $X$ **to** $Z$ **using** $Y$;
    (2.2) output "move $X$ to $Y$";
    (2.3) call **move** $N - 1$ **from** $Z$ **to** $Y$ **using** $X$;
(3) return.
```

我们有类似的思考



To prove partial correctness, we use a variant of the intermediate assertion method that befits the non-iterative nature of recursive algorithms. Rather than trying to formulate the local situation at a given point, we try to formulate our expectations of the entire recursive routine just prior to entering it. This is then used in a cyclic-

Our expectation of Move:



*Assume that the peg names A, B, and C are associated, in some order, with the variables X, Y, and Z. Then, a terminating execution of the call **move** N **from** X **to** Y **using** Z lists a sequence of ring-moving instructions, which, if started (and followed faithfully) in any legal configuration of the rings and pegs in which at least the N smallest rings are on peg X, correctly moves those N rings from X to Y, possibly using Z as temporary storage. Moreover, the sequence adheres to the rules of the Towers of Hanoi problem, and it leaves all other rings untouched.*

Then:

- 我们用数学归纳法证明 the expectation holds for every N!

问题：这种方法和我们用循环不变量的保持证明循环的正确性有何差异？

Open topic:

- 请你以merge sort的递归算法为例，说明：
- 1，递归算法的“不变量”和非递归（循环）算法的“不变量”有何差异？
- 2，证明这个算法的正确性。

（注意：合并过程的正确性也需要给出证明）