

# Algorithms

## Chapter 3

With Question/Answer Animations

# Chapter Summary

- Algorithms (3 class days)
  - Example Algorithms
  - Algorithmic Paradigms
- Growth of Functions (2 class days)
  - Big- $O$  and other Notation
- Complexity of Algorithms (1 class day)

# Algorithms

Section 3.1

# Algorithm Section Summary

- Day 1
  - Properties (Slides 5-10)
  - Searching (Slides 11-15)
- Day 2/3
  - Sorting (Slides 16-25)
- Day 3
  - Greedy (Slides 26-34)
  - Halting Problem (Slide 35-37)
    - Be able to state what it is (proof optional)

# Problems and Algorithms

In many domains  $\exists$  problems that ask for output with specific properties when given valid input.

- In computer programming, an *algorithm* is a procedure or set of steps which
  - takes valid input
  - produces desired output.
- More generally, an *algorithm* is a finite set of precise instructions for calculations or for solving a problem.



Abu Ja'far Mohammed Ibin Musa Al-Khowarizmi (780-850), spent most of life in present-day Iraq

# Algorithms

**Example:** Describe algorithm for finding **maximum** value or **max** in finite sequence of integers  $\{a_1, a_2, \dots, a_n\}$

**Solution:**

1. Set maximum:  $\max = a_1$
2. Compare next integer in sequence to max
  - If it is larger than max, set max equal to this integer.
3. Repeat previous step if  $\exists$  more integers. If not, stop.
4. Upon termination, max is largest integer in sequence.

An actual implementation would depend on the computer programming language, although clearly there is need for a loop, which could either be “while” or “for”.

# Pseudocode

- Algorithms can be described in English or in a particular programming language such as Java or Python.
- *Pseudocode* is an intermediate between an English language description & the coding using a programming language.
- The form of pseudocode we use is specified in Appendix 3. It uses some of the structures found in C++ and Java.
- Programmers can easily take the description of an algorithm in pseudocode & construct a program in a particular language.
- Pseudocode allows us to analyze the number and type of steps required, independent of any particular implementation.
- Compilers often have optimizing algorithms which may greatly improve the user-inputted code, but that is another story.

# Properties of Algorithms

- *Input*: An algorithm has *input* values from a specified set.
- *Output*: The algorithm produces the *output* values from a specified set. The output values are the *solution*.
- *Correctness*: An algorithm should produce the correct output values for each set of input values.
- *Finiteness, Effectiveness, Generality*: An algorithm should produce the output after a finite # of steps for any input.



# Finding Maximum Element

The algorithm in pseudocode:

```
procedure max( $a_1, a_2, \dots, a_n$ : integers)
  max :=  $a_1$ 
  for  $i := 2$  to  $n$ 
    if  $max < a_i$  then  $max := a_i$ 
  return max{max is the largest element}
```

- Does this algorithm have all the properties listed on the previous slide?

(Input, Output, Correctness, Finiteness, Effectiveness, Generality)

# Our Algorithm Problems

Three classes of problems will be studied in this section:

1. *Searching*: finding position of particular element in list.
2. *Sorting*: putting elements of list into increasing order.
3. *Optimization*: determining optimal value (max or min) of a particular quantity  $\forall$  inputs.

# Searching

The *searching problem* is to locate an element  $x$  in the list of  $a_1, a_2, \dots, a_n$ , or determine that it is not in the list.

- The solution is (first) location of term in list that equals  $x$  ( $i$  is the solution if  $x = a_i$ ) or 0 if  $x$  is not in the list.
- For example, before boarding a plane, airlines must check to see if a customer is on the no-fly list.
- We study two different algorithms:
  - linear
  - binary

# Linear Search Algorithm

## (for non-sorted sets)

The linear search algorithm locates an item in a list by examining elements in the sequence one at a time, starting at the beginning:

- First compare  $x$  with  $a_1$ . If they are equal, **return 1**.
- If not, try  $a_2$ . If  $x = a_2$ , **return 2**.
- Keep going, & if no match found when entire list is scanned, **return 0**.

```
procedure linear search( $x$ : integer;  $a_1, a_2, \dots, a_n$ : integers)
 $i := 1$ 
while ( $i \leq n$  and  $x \neq a_i$ )
     $i := i + 1$ 
if  $i \leq n$  then  $location := i$ 
else  $location := 0$ 
return  $location$  { $location$  is the subscript of 1st term that
    equals  $x$ , or is 0 if  $x$  is not found}
```

# Binary Search (for sorted sets)

- Assume input is list of items in increasing order.
- Algorithm begins by comparing element to be found with middle element.
  - If middle element is lower, search proceeds with upper half of list.
  - If not, search proceeds with lower half of list.
- Repeat this process until we have a list of size 1.
  - If element looked for is equal to this element, position is returned.
  - Otherwise, 0 is returned to indicate that element was not found.
- In Section 3.3, we show that binary search algorithm is efficient.

# Binary Search pseudocode

```
procedure binary search( $x$ : integer,  $a_1, a_2, \dots, a_n$ : increasing integers)
   $i := 1$  { $i$  is the left endpoint of interval}
   $j := n$  { $j$  is right endpoint of interval}
  while  $i < j$ 
     $m := \lfloor (i + j) / 2 \rfloor$ 
    if  $x > a_m$  then  $i := m + 1$ 
    else  $j := m$ 
  if  $x = a_i$  then  $location := i$ 
  else  $location := 0$ 
  return  $location$  { $location$  is the subscript  $i$  of the term  $a_i$  equal to  $x$ ,
    or 0 if  $x$  is not found}
```

# Binary Search example input

**Example:** The steps taken by a binary search for 19 in the list:

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

1. List has 16 elements, so midpoint is 8. The value in the 8<sup>th</sup> position is 10. Since  $19 > 10$ , further search is restricted to positions 9 through 16.

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

2. Midpoint of list (positions 9 through 16) is now 12<sup>th</sup> position with value of 16. Since  $19 > 16$ , further search is restricted to the 13<sup>th</sup> position and above.

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

3. Midpoint of current list is now 14<sup>th</sup> position with a value of 19. Since  $19 \neq 19$ , further search is restricted to the portion from 13<sup>th</sup> through the 14<sup>th</sup> positions.

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

4. Midpoint of the current list is now 13<sup>th</sup> position with value of 18. Since  $19 > 18$ , search is restricted to the portion from the 14<sup>th</sup> position through the 14<sup>th</sup>.

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

5. Now list has single element and loop ends. Since  $19=19$ , location 14 is returned.

# Sorting

To *sort* elements of a list is to put them in increasing order.

Example is files in directory, sorted using any one of several categories (name, size, type, date created, date modified)

Substantial computing resources are devoted to sorting

Many algorithms have been invented for sorting:

binary, insertion, bubble, selection, merge, quick, tournament

In Section 3.3, we discuss the efficiency of these algorithms.



# Bubble Sort on $a_1, \dots, a_n$

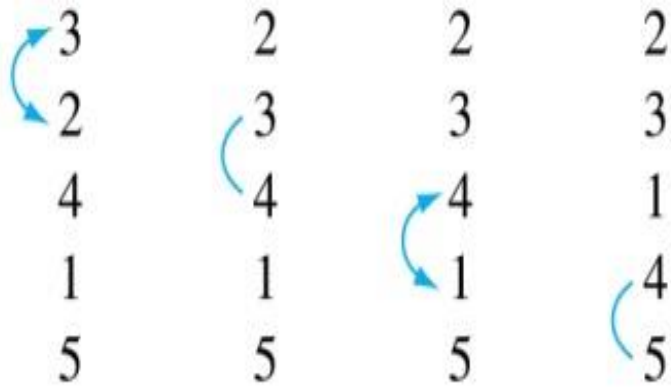
*Bubble sort* in  $n - 1$  passes through a list, interchanges every pair of elements found to be out of order.


```
procedure bubblesort( $a_1, \dots, a_n$ : real numbers with  $n \geq 2$ )  
  for  $i := 1$  to  $n - 1$   
    for  $j := 1$  to  $n - i$   
      if  $a_j > a_{j+1}$  then interchange  $a_j$  and  $a_{j+1}$   
  { $a_1, \dots, a_n$  is now in increasing order}
```


After the  $i^{\text{th}}$  pass, the last  $i$  elements are in order, so the passes decrement in length for each new pass.

# Bubble Sort on 3 2 4 1 5

First pass

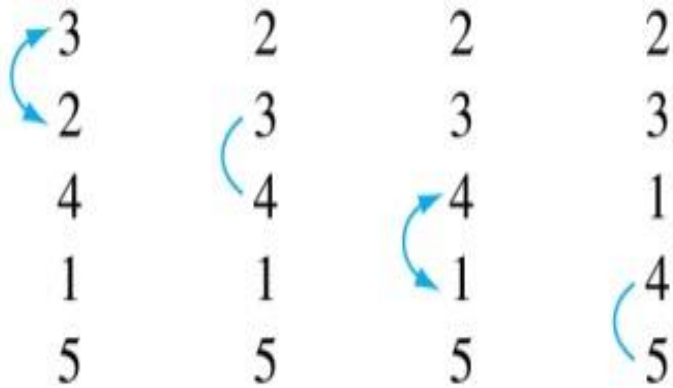


 : an interchange

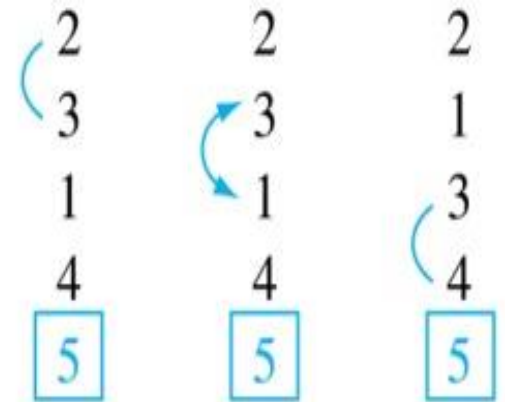
 : pair in correct order


# Bubble Sort on 3 2 4 1 5


First pass



Second pass



 : an interchange

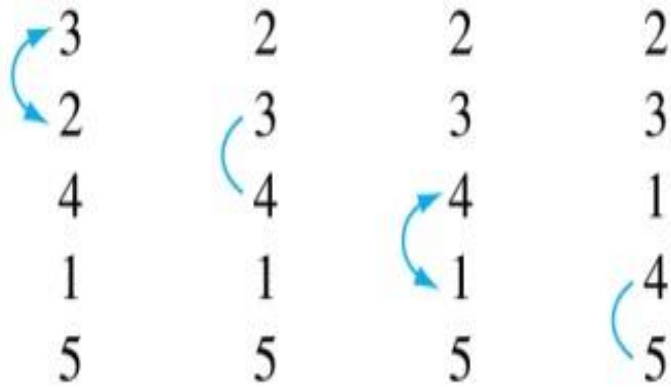
 : pair in correct order

numbers in color

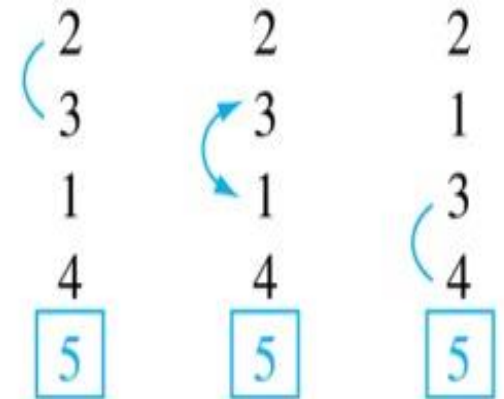
guaranteed to be in correct order

# Bubble Sort on 3 2 4 1 5

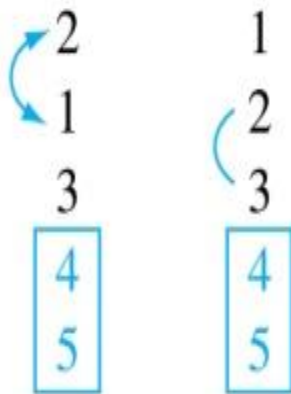
First pass





Second pass



Third pass



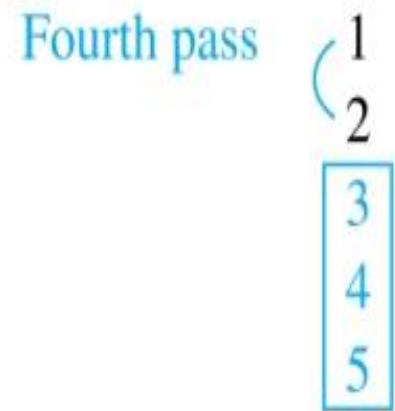
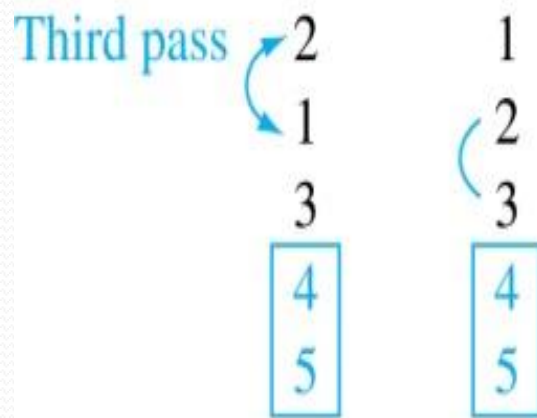
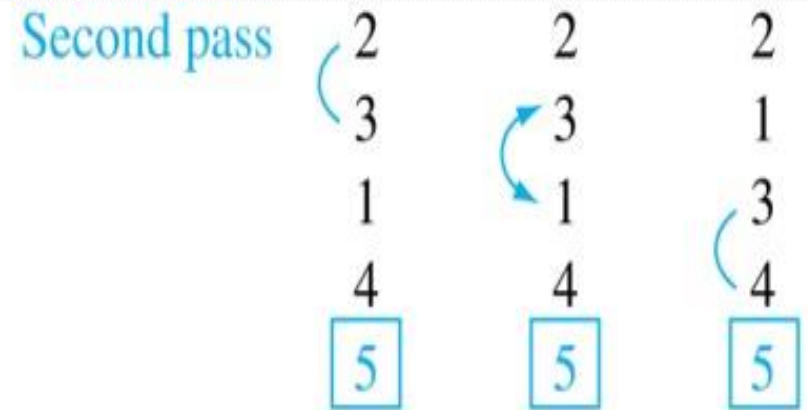
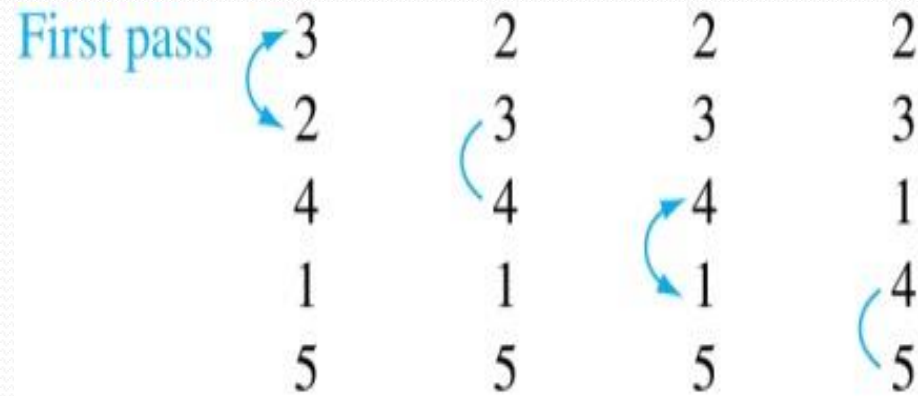
 : an interchange

 : pair in correct order

numbers in color

guaranteed to be in correct order

# Bubble Sort on 3 2 4 1 5



# Insertion Sort

- *Insertion sort* begins with 2<sup>nd</sup> element. It compares 2<sup>nd</sup> element with 1<sup>st</sup> and puts it before the 1<sup>st</sup> if it is not larger.
- Next the 3<sup>rd</sup> element is put into the correct position among the first 3 elements.
- In each subsequent pass, the  $n+1$ <sup>st</sup> element is put into its correct position among the first  $n+1$  elements.
- Linear search is used to find the correct position.

# Insertion Sort

**procedure** *insertion sort*( $a_1, \dots, a_n$ : real numbers with  $n \geq 2$ )

**for**  $j := 2$  to  $n$

$i := 1$

**while**  $a_j > a_i$

$i := i + 1$

$m := a_j$

**for**  $k := 0$  to  $j - i - 1$

$a_{j-k} := a_{j-k-1}$

$a_i := m$

{Now  $a_1, \dots, a_n$  is in increasing order}

# Insertion Sort on 3 2 4 1 5

3 2 4 1 5    1<sup>st</sup> two positions are interchanged

2 3 4 1 5    3<sup>rd</sup> element remains in its position

2 3 4 1 5    4<sup>th</sup> element is placed at beginning

1 2 3 4 5    5<sup>th</sup> element remains in its position

1 2 3 4 5



# Optimization

*Optimization problems* minimize or maximize some parameter over all possible inputs.

- Examples we will study are:
  - Finding a route between two cities with smallest total mileage.
  - Determining how to encode messages using fewest possible bits.
  - Finding links between network nodes using least amount of fiber.
- Optimization problems often solved using a *greedy algorithm*, which makes the “best” choice at each step.

# Greedy Algorithms

Making “best choice” at each step does not necessarily give optimal solution, but in many instances it does.

- After specifying “best choice” at each step
  - we show an optimal solution always produced;
  - or find counterexample to show that it does not.
- Greedy approach is an example of *algorithmic paradigm*, a general approach for designing an algorithm.
- We return to algorithmic paradigms in Section 3.3.

# Example: Making Change



Make change of  $n = 67$  cents with quarters, dimes, nickels, and pennies, using least total number of coins.

**Idea:** At each step choose coin with the largest possible value that does not exceed the amount of change left.

1. First choose quarter leaving  $67 - 25 = 42$  & another quarter leaving  $42 - 25 = 17$ .
2. Then choose dime, leaving  $17 - 10 = 7$ .
3. Choose nickel, leaving  $7 - 5 = 2$ .
4. Choose penny, leaving 1 & another penny leaving 0.

# Greedy Change-Making Algorithm:

coin denominations  $c_1, c_2, \dots, c_r$

```
procedure change( $c_1, c_2, \dots, c_r$ : values of coins,  
                where  $c_1 > c_2 > \dots > c_r$ ;  $n$ : a positive integer)  
for  $i := 1$  to  $r$   
     $d_i := 0$  { $d_i$  counts coins of denomination  $c_i$ }  
    while  $n \geq c_i$   
         $d_i := d_i + 1$  {add a coin of denomination  $c_i$ }  
         $n := n - c_i$   
{the sequence  $d_1, d_2, \dots, d_r$  provides # of each coin needed}
```

For example of U.S. currency, we have quarters, dimes, nickels and pennies, with  $c_1 = 25$ ,  $c_2 = 10$ ,  $c_3 = 5$ , and  $c_4 = 1$ .

# Proving Optimality

If  $n \in \mathbb{Z}^+$ , then  $n$  cents in change using quarters, dimes, nickels, and pennies, using fewest coins possible has:

**Lemma 1:** at most 2 dimes, 1 nickel, 4 pennies, and cannot have 2 dimes and a nickel.

**Proof:** By contradiction

- If we had 3 dimes, we replace them with a quarter and a nickel.
- If we had 2 nickels, we replace them with 1 dime.
- If we had 5 pennies, we replace them with a nickel.
- If we had 2 dimes & 1 nickel, we replace them with a quarter.

**Lemma 2:** total amount of change not in quarters  $\leq 24$  cents.

**Proof:** The allowable combinations, have a maximum value of 24 cents: 2 dimes and 4 pennies.

# Proving Optimality for U.S. Coins

**Theorem:** The greedy change-making algorithm for U.S. coins produces change using the fewest coins possible.

**Proof:** By contradiction.

1. Assume  $\exists n \in \mathbb{Z}^+$  such that change can be made for  $n$  cents using quarters, dimes, nickels, and pennies, with a fewer total number of coins than given by the algorithm.
2. Suppose  $q' < q$  where  $q' = \#$ quarters used in this optimal way and  $q = \#$ quarters in the greedy algorithm's solution. But this is not possible by Lemma 2, since value of the coins other than quarters can not be greater than 24 cents.
3. Similarly, by Lemma 1, the two algorithms must have the same number of dimes, nickels, and pennies (exercise).

# Greedy Change-Making Algorithm

Optimality depends on the denominations available.

- For U.S. coins, optimality still holds if we add half dollar coins (50 cents) and dollar coins (100 cents).
- But if we allow only quarters (25 cents), dimes (10 cents), and pennies (1 cent), the algorithm no longer produces the minimum number of coins.
  - Consider the example of 31 cents. The optimal number of coins is 4, i.e., 3 dimes and 1 penny. What does the algorithm output?

# Greedy Scheduling

**Example:** We have a group of proposed talks with start and end times. Construct a greedy algorithm to schedule as many as possible in a lecture hall, under the following assumptions:

- When a talk starts, it continues till the end.
- No two talks can occur at the same time.
- A talk can begin at the same time that another ends.
- Once we have selected some of the talks, we cannot add a talk which is incompatible with those already selected because it overlaps at least one of these previously selected talks.
- How should we make the “best choice” at each step of the algorithm? That is, which talk do we pick ?

The talk that starts earliest among those compatible with already chosen talks?

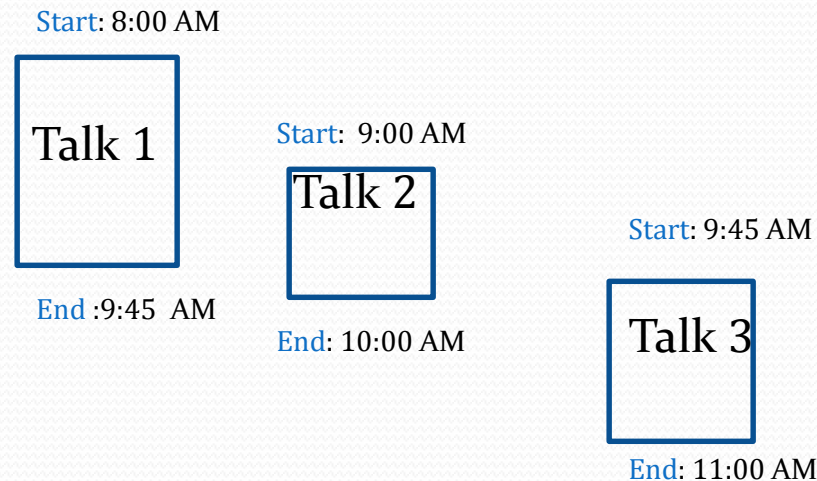
The talk that is shortest among those already compatible?

The talk that ends earliest among those compatible with already chosen talks?



# Greedy Scheduling

- Picking the shortest talk doesn't work.



- Can you find a counterexample here?
- But picking the one that ends soonest does work. The algorithm is specified on the next page.

# Greedy Scheduling algorithm

**Solution:** At each step, choose talks with the earliest ending time among talks compatible with those selected.

```
procedure schedule( $s_1, s_2, \dots, s_n$  : start times;  $e_1, e_2, \dots, e_n$  : end times)
  sort talks by finish time {reorder so that  $e_1 \leq e_2 \leq \dots \leq e_n$ }
   $S := \emptyset$ 
  for  $j := 1$  to  $n$ 
    if talk  $j$  is compatible with  $S$  then
       $S := S \cup \{\text{talk } j\}$ 
  return  $S$  {  $S$  is the set of talks scheduled }
```

- Will be proven correct by induction in Chapter 5.

# Halting Problem

Develop a procedure that takes as input a computer program along with its input and determines whether the program will eventually halt with that input.

**Solution:** Proof by contradiction.

Assume that there is such a procedure and call it  $H(P,I)$ .

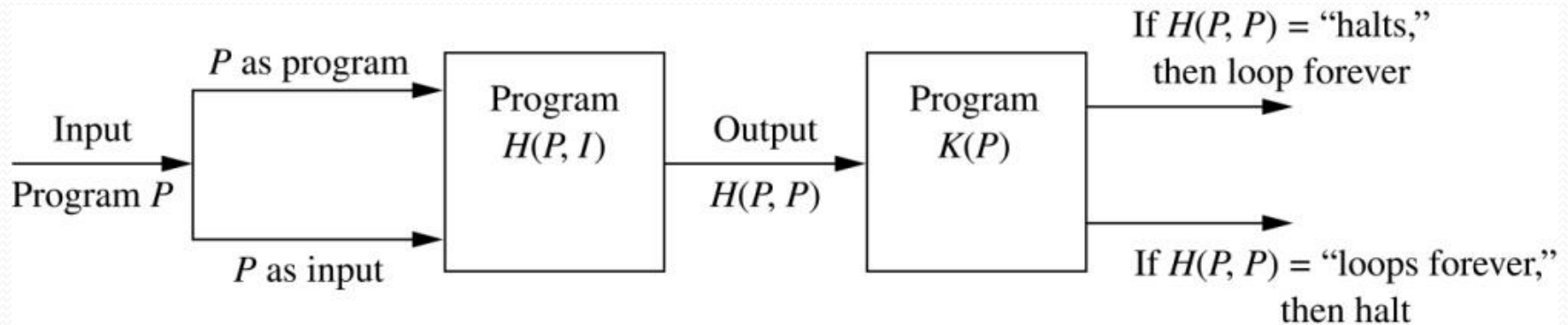
Procedure  $H(P,I)$  takes as input program  $P$  & input  $I$ .

- $H$  outputs “halt” if it is the case that  $P$  will stop when run with input  $I$ .
- Otherwise,  $H$  outputs “loops forever.”

# Halting Problem (cont.)

Since a program is a string of characters, we call  $H(P,P)$ . Construct a procedure  $K(P)$ , which works as follows.

- If  $H(P,P)$  outputs “loops forever” then  $K(P)$  halts.
- If  $H(P,P)$  outputs “halt” then  $K(P)$  goes into an infinite loop printing “ha” on each iteration.



# Halting Problem (cont.)

Now we call  $K$  with  $K$  as input, i.e.  $K(K)$ .

- If the output of  $H(K,K)$  is “loops forever” then  $K(K)$  halts. **A Contradiction.**
- If the output of  $H(K,K)$  is “halts” then  $K(K)$  loops forever. **A Contradiction.**

Hence,  $\nexists$  procedure that can decide whether or not an arbitrary program halts.

$\therefore$  The halting problem is unsolvable.