Algorithms

Chapter 3

Algorithms

Section 3.1

Section Summary

- Introduction to Algorithms
- Algorithms for Searching and Sorting
- Greedy Algorithms
- Stable Matchings
- Halting Problem

Video 30: Algorithms

- Introduction to Algorithms

What is an algorithm?

Finite set of **well-defined**, **computer-implementable** instructions to perform a specified **task**

- to perform a computation
- to solve a certain problem
- to reach a certain destination

Example

Task: Find the maximum value in a finite sequence of integers.

Algorithm:

1. Set the temporary maximum equal to the first integer in the sequence.
2. Compare the next integer in the sequence to the temporary maximum. If it is larger than the temporary maximum, set the temporary maximum equal to this integer.
3. Repeat the previous step if there are more integers. If not, stop.
4. When the algorithm terminates, the temporary maximum is the largest integer in the sequence.

Example

Sequence	3	5	1	7	2	1
Temporary maximum	3	5	5	7	7	7

Specifying Algorithms

- Algorithms can be specified in different ways.
    - Natural language
    - Pseudo-code
    - Programming language
- Pseudocode is an intermediate step between an Natural Language description (more precise) and a coding of these steps using a programming language (more general).
    - Programmers can use the description of an algorithm in pseudocode to construct a program in a particular language.
    - Pseudocode helps us analyze the properties of an algorithm, independent of the actual programming language used to implement it.

Example

Task: Find the maximum value in a finite sequence of integers.

Algorithm in pseudocode:

> **procedure** *max*(a_1, a_2, …., a_n: integers)
>   tmp_*max* := a_1
>   **for** *i* := 2 to *n*
>   **if** tmp_*max* < a_i **then** tmp_*max* := a_i
>   **return** tmp_*max*

Example

Task: Find the maximum value in a finite sequence of integers.

Algorithm in Python:

```
def max(a):
    tmp_max = a[0]
    for i in range(2, len(a)):
        if tmp_max < a[i]:
            tmp_max = a[i]
    return tmp_max
```

```
max([2,5,3,7,4,1])
```

```
7
```

Typical Problems Solved by Algorithms

1. *Searching problems*: finding the position of a particular element in a list.
2. *Sorting problems*: putting the elements of a list into increasing order.
3. *Optimization Problems*: determining the optimal value (maximum or minimum) of a particular quantity over all possible inputs.

Summary

- Definition of Algorithm
- Pseudocode
- Types of Algorithms

Video 31: Searching Algorithms

- Linear Search Algorithm
- Binary Search Algorithm

Searching Problems

Task: Given a list $S = a_1, a_2, a_3, \ldots, a_n$ of distinct elements and some x, if $x \in S$ return i such that $a_i = x$, else return 0.

Examples

- Find a word in a dictionary
- Find a name in a customer table
- Find an amount in a bank transaction table

Linear Search Algorithm

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

Algorithm

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

Linear Search Algorithm

```
procedure linear search(x: integer, a_1, a_2, …,a_n: distinct integers)
i := 1
while (i ≤ n and x ≠ a_i)
    i := i + 1
    if i ≤ n then location := i else location := 0
return location
```

Example

Sequence		3	5	1	7	2	1	
Searching x = 2								
$x \neq a_i$		T	T	T	T	F		
location	*1*	*2*	*3*	*4*	*5*	*5 (returned)*		
Searching x = 4								
$x \neq a_i$		T	T	T	T	T	F	
location	*1*	*2*	*3*	*4*	*5*	*6*	*0*	*0 (returned)*

Binary Search

Assume the input is a list of items in increasing order.

- The algorithm begins by comparing the element to be found with the middle element.
  - If the middle element is lower, the search proceeds with the upper half of the list.
  - If it is not lower, the search proceeds with the lower half of the list (including the middle position).
- Repeat this process until we have a list of size 1.
  - If the element we are looking for is equal to the element in the list, the position is returned.
  - Otherwise, 0 is returned to indicate that the element was not found.

Example

Binary search for 19 in the list:

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

1. The list has 16 elements, so the midpoint is 8. The value in the 8th 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. The midpoint of the current list (positions 9 through 16) is now the 12th position with a value of 16. Since 19 > 16, further search is restricted to the 13th position and above.

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

3. The midpoint of the current list is now the 14th position with a value of 19. Since 19 $\not>$ 19, further search is restricted to the portion from the 13th through the 14th positions .

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

4. The midpoint of the current list is now the 13th position with a value of 18. Since 19 > 18, search is restricted to the portion from the 14th position through the 14th.

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

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

Binary Search

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
   $i := 1$ *{i is the left endpoint of interval}*
   $j := n$ *{j is right endpoint of interval}*
   while $i < j$ *{at least two elements in the list}*
      $m := \lfloor (i + j)/2 \rfloor$ *{take the midpoint}*
      if $x > a_m$ then $i := m + 1$ **else** $j := m$
   if $x = a_i$ **then** *location* $:= i$ **else** *location* $:= 0$
   return *location*

Summary

- Search is a fundamental operation for data
- Linear and Binary Search
- Binary Search is more efficient, but requires sorting

Video 32: Sorting Algorithms

- Bubble Sort
- Insertion Sort

Sorting Problems

Task: Given a list S = a_1, a_2, a_3, …, a_n, return a list where the elements are put in increasing order.

Sorting is an important problem because:

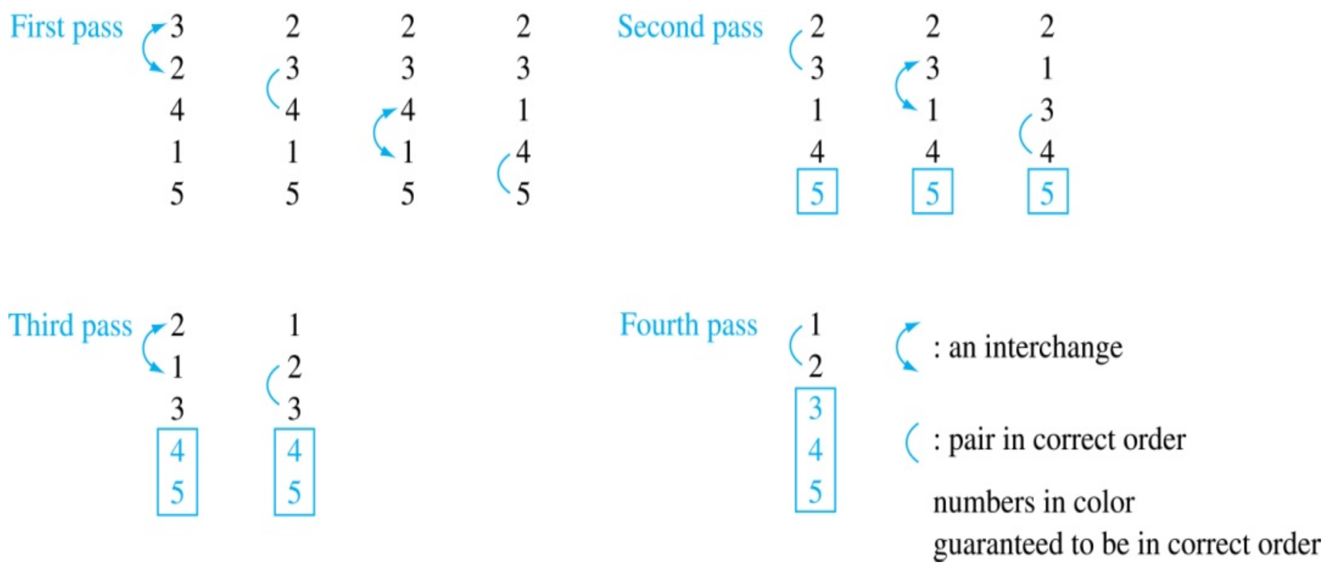
- A nontrivial percentage of all computing resources are devoted to sorting (e.g. in large databases)
- An amazing number of fundamentally different algorithms have been invented for sorting
- Sorting algorithms are useful to illustrate the basic notions of computer science.

Bubble Sort

Bubble sort makes multiple passes through a list.

- In one pass, every pair of elements that are found to be out of order are interchanged.
- Since the last element is guaranteed to be the largest after the first pass, in the second pass it needs no more to be inspected.
- In every pass one more element at the end needs to be no more inspected.

Example



- At the first pass the largest element has been put into the correct position
- At the end of the second pass, the 2nd largest element has been put into the correct position.
- In each subsequent pass, an additional element is put in the correct position.

Bubble Sort

Bubble sort makes multiple passes through a list.

Every pair of elements that are found to be out of order are interchanged.

procedure *bubblesort*($a_1,\ldots,a_n$: 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}

Insertion Sort

Insertion sort begins with the 2nd element.

- It compares the 2nd element with the 1st and puts it before the first if it is not larger.
- Next the 3rd element is put into the correct position among the first 3 elements.
- In each subsequent pass, the j+1st element is put into its correct position among the first j+1 elements.
- Linear search is used to find the correct position.

Example

Insertion sort with the input: 3 2 4 1 5

i. 2 3 4 1 5 (*first two positions are interchanged*)

ii. 2 3 4 1 5 (*third element remains in its position*)

iii. 1 2 3 4 5 (*fourth is placed at beginning*)

iv. 1 2 3 4 5 (*fifth element remains in its position*)

Insertion Sort Pseudocode

procedure *insertion sort*($a_1,\ldots,a_n$: real numbers with $n \geq 2$)
  for j := 2 **to** n
    i := 1
    while $a_j > a_i$ **and** $i < j$ *{move element a_j to right position}*
      $i := i + 1$
    $m := a_j$
    for k := 0 **to** $j - i - 1$ *{shift elements to make place for a_i}*
      $a_{j-k} := a_{j-k-1}$
    $a_i := m$

2 3 4 1 5   j = 4, i = 1 (after **while**)

1 2 3 4 5   m = a_4, j – i – 1 = 2, therefore $a_4 := a_3$, $a_3 := a_2$, $a_2 := a_1$ (**for** loop)

finally $a_1 := m = a_4$

Summary

- Sorting is a fundamental operation for data
- Bubble and Insertion Sort are basic algorithms
- More efficient ones will be shown later

Video 33: Optimization Algorithms

- Greedy Algorithms
- Cashier’s Algorithm

Optimization Problems

Optimization problems minimize or maximize some parameter over all possible inputs

Examples

- Finding a route between two cities with the smallest total mileage.
- Determining how to encode messages using the fewest possible bits.
- Finding the fiber links between network nodes using the least amount of fiber.

Greedy Algorithms

Optimization problems can often be solved using a *greedy algorithm*, which makes the “best” choice at each step.

- Making the “best choice” at each step does not necessarily produce an optimal solution to the overall problem
  - but in many instances, it does.
- After specifying the greedy algorithm,
  - Either we prove that this approach always produces an optimal solution
  - or we find a counterexample to show that it does not.

Cashier's Algorithm

Task: Find for an amount of any n cents the least total number of coins using the following coins: quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent).

(Greedy) Idea: At each step choose the coin with the largest possible value that does not exceed the amount left.

Example

Change for $n = 67$ cents.

1. First choose a quarter leaving $67 - 25 = 42$ cents.
2. Then choose another quarter leaving $42 - 25 = 17$ cents
3. Then choose 1 dime, leaving $17 - 10 = 7$ cents.
4. Choose 1 nickel, leaving $7 – 5 = 2$ cents.
5. Choose a penny, leaving one cent.
6. Choose another penny leaving 0 cents.

A total of 6 coins have been used.

Cashier's Algorithm

The algorithm works with any coin denominations $c_1, c_2, \ldots, c_r$

procedure *change*($c_1, c_2, \ldots, c_r$: values of coins, where $c_1 > c_2 > \ldots > c_r$; n: a positive integer)
for $i := 1$ **to** r *{start with the largest coins}*
  $d_i := 0$ *{d_i counts the coins of denomination c_i}*
  while $n \geq c_i$
    $d_i := d_i + 1$ *{add a coin of denomination c_i}*
    $n = n - c_i$ *{remove the value of the coin}*
return $d_1, d_2, \ldots, d_r$

For the 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

Show that the change making algorithm for *U.S.* coins is optimal.

Lemma 1: If *n* is a positive integer, then *n* cents in change using quarters, dimes, nickels, and pennies, using the fewest coins possible

1. has at most 2 dimes, 1 nickel, 4 pennies
2. cannot have 2 dimes and 1 nickel
3. the total amount of change in dimes, nickels, and pennies cannot exceed 24 cents.

Proof of Lemma 1

Proof:

Property 1: By contradiction (not the fewest coins)

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

Property 2: By contradiction (not the fewest coins)

- If we had 2 dimes and 1 nickel, we could replace them with a quarter.

Property 3: is a consequence

- The largest allowable combination, 2 dimes and 4 pennies, has a maximum value of 24 cents.

Proving Optimality

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

Proof: By contradiction.

1. Assume there is a positive integer *n* such that change can be made with a fewer total number of coins than given by the algorithm.
2. Let q' be the number of quarters used in this optimal solution
   1. *$q' \leq q$, since the greedy algorithm uses the maximum number of quarters possible*
   2. *$q' < q$ is not possible* since then we would have more than 25 cents in dimes, nickels, and pennies, which contradicts Lemma 1.
3. Therefore $q' = q$
4. Similarly the greedy algorithm chose the maximum possible number of dimes
   1. we cannot replace a dime by at most 1 nickel and 4 pennies of value 9.
5. Similarly the greedy algorithm chose the maximum number of nickels, and therefore also the number of nickels and pennies is the same.

Cashier's Algorithm Discussion

Optimality depends on the denominations available.

If we allow only quarters (25 cents), dimes (10 cents), and pennies (1 cent), the algorithm no longer produces the minimum number of coins.

Counterexample: Consider the example of 31 cents.

- The optimal number of coins is 4, i.e., 3 dimes and 1 penny.
- The algorithm outputs 1 quarter and 6 dimes.

Summary

- Greedy Algorithms
- Cashier's Algorithm
- Optimality Proof

Video 34: Stable Matchings

- Matching
- Maximum Matching
- Stable Matching
- Greedy Algorithm for finding the Stable Matching

Stable Matchings

Task: Pair elements from equally sized two groups considering their preferences for members of the other group so that there are no ways to improve the preferences.

This requires first some definitions …

Matchings

Definition: Given a finite set A, a **matching** of A is a set of (unordered) pairs of distinct elements of A where any element occurs in at most one pair (such pairs are called independent).

Definition: A **maximum matching** is a matching that contains the largest possible number of pairs.

Examples

Let A = {1, 2, 3, 4}

- {(1, 2)} and {(1, 3), (2, 4)} are matchings
- {(2, 2)} and {(1, 2), (2, 4)} are not matchings
- {(1,3),(2,4)} is a maximum matching

Let A = {1, 2, 3, 4, 5}

- {(1, 3), (2, 4)} and {(5, 3), (2, 4)} are maximum matchings

Preferences

A **preference list** L_x defines for every element $x \in A$ the order in which the element prefers to be paired with. $x \in A$ prefers y to z if y precedes z on L_x.

Example: A = {Lou, Glenn, Bobbie, Tyler}

- L_{Lou} =(Glenn, Bobbie, Tyler)
- L_{Glenn} =(Bobbie, Lou, Tyler)
- L_{Bobbie} =(Lou, Glenn, Tyler)
- L_{Tyler} =(Lou, Glenn, Bobbie)

Lou prefers Glenn over Bobbie, and Bobbie over Tyler

Stability of Matching

Definition: A matching is **unstable** if there are two pairs (x, y), (v, w) in the matching such that x prefers v to y and v prefers x to w.

Definition: A **stable** matching is a matching that is not unstable.

Example: {(Glenn, Lou), (Bobbie, Tyler)} is unstable

- L_{Glenn} =(Bobbie, Lou, Tyler): Glenn prefers Bobbie over Lou
- L_{Bobbie} =(Lou, Glenn, Tyler): Bobbie prefers Glenn over Tyler

Therefore Glenn and Bobbie will leave their current partner and pair up.

Stability of Matching

New matching: {(Glenn, Bobbie), (Lou, Tyler)}

- L_{Lou} =(Glenn, Bobbie, Tyler): Lou prefers Bobbie
- L_{Bobbie} =(Lou, Glenn, Tyler): Bobbie prefers Lou

Therefore Lou and Bobbie will leave their current partner and pair up.

New matching: {(Lou, Bobbie), (Glenn, Tyler)}

- L_{Lou} =(Glenn, Bobbie, Tyler): Lou prefers Glenn
- L_{Glenn} =(Bobbie, Lou, Tyler): Glenn prefers Lou

New matching {(Glenn, Lou), (Bobbie, Tyler)} is initial matching

There does not exist a stable maximal matching!

Marriage Problem

To guarantee, irrespective of the preference lists, the existence of a stable maximum matching it suffices to use a more stringent pairing rule.

Definition: Given a set with even cardinality, partition A into two disjoint subsets A_1 and A_2 with $A_1 \cup A_2 = A$ and $|A_1| = |A_2|$. A **matching** is
a bijection from the elements of one set to the elements of the other set.

That means, that pairs can only consist of one element of A_1 and A_2 each.

Example

Assume A_1 = {Lou, Glenn} and A_2 = {Bobbie, Tyler}

We have to adapt the preference lists

- L_{Lou} = (Bobbie, Tyler)
- L_{Glenn} = (Bobbie, Tyler)
- L_{Bobbie} = (Lou, Glenn)
- L_{Tyler} = (Lou, Glenn)

Now {(Lou, Bobbie), (Glenn, Tyler)} is stable

{(Bobbie, Glenn), (Lou, Tyler)} is unstable (Lou prefers Bobbie, Bobbie prefers Lou)

Existence of Stable Maximum Matching

A greedy algorithm to construct a stable maximum matching for the marriage problem

- It proofs existence of stable matching
- It is efficient

(and it produced a Nobel price)

Gale-Shapley Algorithm

Let M be the set of pairs under construction;

Initially M = $\emptyset$

While $|M| < |A_1|$:

Select an unpaired $x \in A_1$

Let x propose to the first element $y \in A_2$ on L_x:

if y is unpaired **then** add the pair (x, y) to M

else *(i.e., if y is paired already)*

Let $x' \in A_1$ be the element that y is paired to, *(i.e., $(x', y) \in M$)*

if x' precedes x on L_y **then** remove y from L_x

else *(i.e., if x precedes x' on L_y)*

Replace $(x', y) \in M$ by (x, y) and remove y from $L_{x'}$

Example

Examples Assume that $k = 4$, that $A_1 = \{x_1, x_2, x_3, x_4\}$, $A_2 = \{y_1, y_2, y_3, y_4\}$, that all $x \in A_1$ have the same preference list $L_x = (y_1, y_2, y_3, y_4)$, and that $L_{y_1} = (x_2, x_1, x_3, x_4)$, $L_{y_2} = (x_4, x_3, x_2, x_1)$, $L_{y_3} = (x_2, x_3, x_4, x_1)$ and $L_{y_4} = (x_4, x_1, x_2, x_3)$. Assume furthermore that the unmatched $x \in A_1$ with the lowest index is always selected (why does this not make a difference?). During the construction the following steps are performed :

```
x1 proposes y1: y1 accepts x1's proposal
                          M: (x1,y1)
x2 proposes y1: y1 dumps current x1 and accepts x2's proposal
                          M: (x2,y1)
x1 proposes y2: y2 accepts x1's proposal
                          M: (x1,y2) (x2,y1)
x3 proposes y1 but y1 prefers current x2 to x3
                          M: (x1,y2) (x2,y1)
x3 proposes y2: y2 dumps current x1 and accepts x3's proposal
                          M: (x2,y1) (x3,y2)
x1 proposes y3: y3 accepts x1's proposal
                          M: (x1,y3) (x2,y1) (x3,y2)
x4 proposes y1 but y1 prefers current x2 to x4
                          M: (x1,y3) (x2,y1) (x3,y2)
x4 proposes y2: y2 dumps current x3 and accepts x4's proposal
                          M: (x1,y3) (x2,y1) (x4,y2)
x3 proposes y3: y3 dumps current x1 and accepts x3's proposal
                          M: (x2,y1) (x3,y3) (x4,y2)
x1 proposes y4: y4 accepts x1's proposal
                          M: (x1,y4) (x2,y1) (x3,y3) (x4,y2)
resulting stable maximum matching: (x1,y4) (x2,y1) (x3,y3) (x4,y2)
```

Summary

- Definition of Maximum Stable Matching
- Gale-Shapley Algorithm
    - Shows existence of maximum stable matching

Video 35: Halting Problem

- Definition of Halting Problem
- A Famous Theorem

Unsolvable Problems

Can every problem by solved by an algorithm?

Answer (Turing): No!

He defined an unsolvable problem, the **halting problem**: Can we 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?

Halting Problem

Theorem: The halting problem that cannot be solved using any procedure.

The proof requires an accurate description of what is a procedure, the input and output and of how a procedure can be encoded as string (Turing machine).

Proof Sketch

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

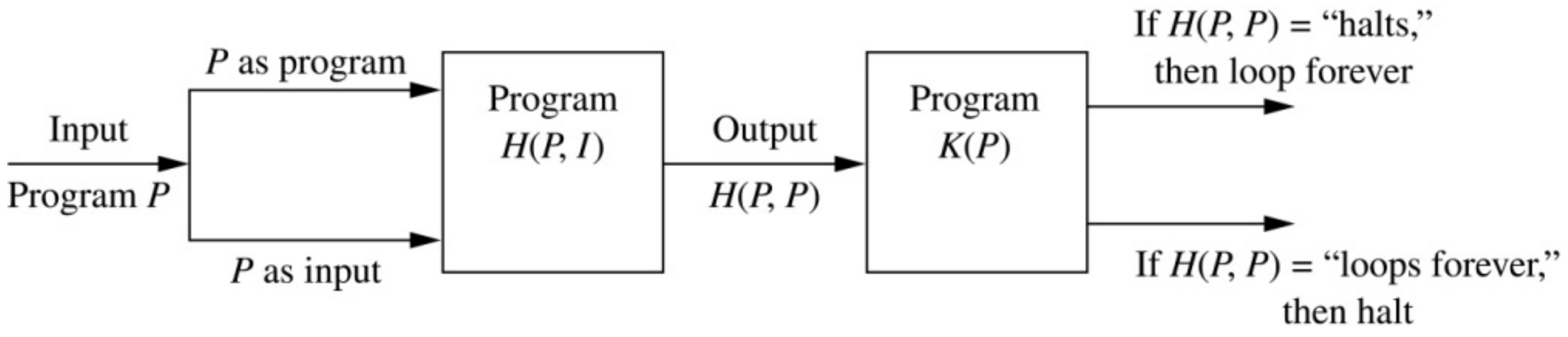
The procedure $H(P, I)$ takes as input a program P and the input I to P.

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

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

Proof Sketch



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.

Therefore, there can not be a procedure that can decide whether or not an arbitrary program halts.

Summary

- Concept of Algorithm
- Searching and Sorting Algorithms
- Greedy algorithms take locally the best decisions
- Algorithms can show the existence of a solution to a problem
- Not every problem can be solved by an algorithm