<u>Chapter 3</u>
Algorithms

Section 3.1 : Algorithms

Algorithms

- An *algorithm* is a finite sequence of precise instructions for performing a computation or for solving a problem.
- *Pseudocodes* are used to generalize any algorithm across different programming languages.
  - Provides an intermediate step between an English language description of an algorithm and an implementation of this algorithm in a programming language.

Properties of Algorithms

- ***Input*** **:** An algorithm has input values from a specified set.
- ***Output*** **:** From each set of input values an algorithm produces output values from a specified set. The output values are the solutions to the problem.
- ***Definiteness*** **:** The steps of an algorithm must be defined precisely.
- ***Finiteness*** **:** An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.
- ***Effectiveness*** **:** It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- ***Generality*** **:** The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

Searching Algorithms

- The task of locating an element in an ordered list in different contexts are called **searching problems**.
  - For instance, a program that checks the spelling of words searches for them in a dictionary, which is just an ordered list of words.
- Searching Algorithms:
  - Linear Search
  - Binary Search

Searching Algorithms (Contd.)

- Linear Search:
  - INPUT: An ordered list of elements.
  - OUTPUT: Index of the desired element in the list (if found).

ALGORITHM 2 The Linear Search Algorithm.

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct 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 the term that equals x, or is 0 if x is not found}

Searching Algorithms (Contd.)

- Binary Search:
  - INPUT: A list of elements sorted in Ascending order.
  - OUTPUT: Index of the desired element in the list (if found).

ALGORITHM 3 The Binary Search Algorithm.

procedure *binary search* (x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {i is left endpoint of search interval}
$j := n$ {j is right endpoint of search 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}

Sorting Algorithms (Contd.)

- The task of ordering elements from an unordered list, in ascending or descending order.
  - For instance, sorting in ascending order, the list $7, 2, 1, 4, 5, 9$ produces the list $1, 2, 4, 5, 7, 9$ and the list d, h, c, a, f (using alphabetical order) produces the list a, c, d, f, h.
- Sorting Algorithms:
  - Bubble Sort
  - Insertion Sort

Sorting Algorithms (Contd.)

- Bubble Sort:
  - INPUT: An unsorted list of elements.
  - OUTPUT: A sorted list of elements in ascending(or descending) order.
  - Smaller Elements "*bubble*" to the top while the Larger Elements "*sink*" to the bottom.
  - Simplest but not the most efficient one.

ALGORITHM 4 The Bubble Sort.

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}
$\{a_1, \ldots, a_n$ is in increasing order$\}$

Sorting Algorithms (Contd.)

- Insertion Sort:
  - INPUT: An unsorted list of elements.
  - OUTPUT: A sorted list of elements in ascending(or descending) order.
  - Simple but usually not the most efficient one.

ALGORITHM 5 The Insertion Sort.

procedure *insertion sort*($a_1, a_2, \ldots, a_n$: 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** $j - i - 1$
        $a_{j-k} := a_{j-k-1}$
    $a_i := m$
$\{a_1, \ldots, a_n$ is in increasing order$\}$

Greedy Algorithms

- Selects the best option instead of considering all possible options that may lead to an optimal solution.
- Once known, that a $Greedy\ Algorithm$ finds a feasible solution, it is necessary to determine optimality of the solution.

Greedy Algorithms(Contd.)

- **Example 5:**

  Consider the problem of making n cents change with $quarters(25), dimes(10), nickels(5)$, and $pennies(1)$, and using the $least\ total\ number$ of coins.

- **Solution:**

  We can devise a greedy algorithm for making change for n cents by making a locally optimal choice at each step; that is, at each step we choose the coin of the largest denomination possible to add to the pile of change without exceeding n cents.

  - For example, to make change for $67\ cents$, we first select a $quarter$ (leaving $42\ cents$). We next select a second quarter (leaving $17\ cents$), followed by a $dime$ (leaving $7\ cents$), followed by a $nickel$ (leaving $2\ cents$), followed by a $penny$ (leaving $1\ cent$), followed by $1\ penny$.

Greedy Algorithms(Contd.)

ALGORITHM 6 Greedy Change-Making Algorithm.

procedure *change*($c_1, c_2, \ldots, c_r$: values of denominations of coins, where
$c_1 > c_2 > \cdots > c_r$; n: a positive integer)
for $i := 1$ **to** r
    $d_i := 0$ {d_i counts the coins of denomination c_i used}
    while $n \geq c_i$
        $d_i := d_i + 1$ {add a coin of denomination c_i}
        $n := n - c_i$
{d_i is the number of coins of denomination c_i in the change for $i = 1, 2, \ldots, r$}

The Halting Problem

- Problem Statement:
  - *Is there any procedure that takes as input*
    - *A computer program and*
    - *Input to the program*

  and determines whether the program will eventually stop when run with this input?

The Halting Problem(Contd.)

- Discussion:
  - Consider a procedure $H(P, I)$ with the following:
    - INPUT: P as a $procedure$ and I as the $input$ to the procedure P.
    - OUTPUT:

$$H(P, I) = \begin{cases} \text{"}Halts\text{"} & , if\ P\ stops\ with\ input\ I \\ \text{"}Loops\ Forever\text{"} & , otherwise \end{cases}$$

  - POINT to be noted!!:
    - When a procedure is coded, it is expressed as a string of characters, which can be interpreted as a sequence of bits. Meaning, the procedure itself can be interpreted as an INPUT. Thus, it is safe to assume that, H can take the procedure P as both of its parameters, i.e. $H(P, P)$ is possible.

The Halting Problem(Contd.)

- Consider another procedure $K(P)$ with the following:
  - INPUT: The output of $H(P,P)$ which is either "$Halts$" or "$Loops\ Forever$".
  - OUTPUT:

$$K(P) = \begin{cases} "Halts" & , if\ H(P,P)\ outputs\ "Loops\ Forever" \\ "Loops\ Forever" & , if\ H(P,P)\ outputs\ "Halts" \end{cases}$$

i.e. $K(P)$ specifies the opposite of whatever $H(P,P)$ gives as output.

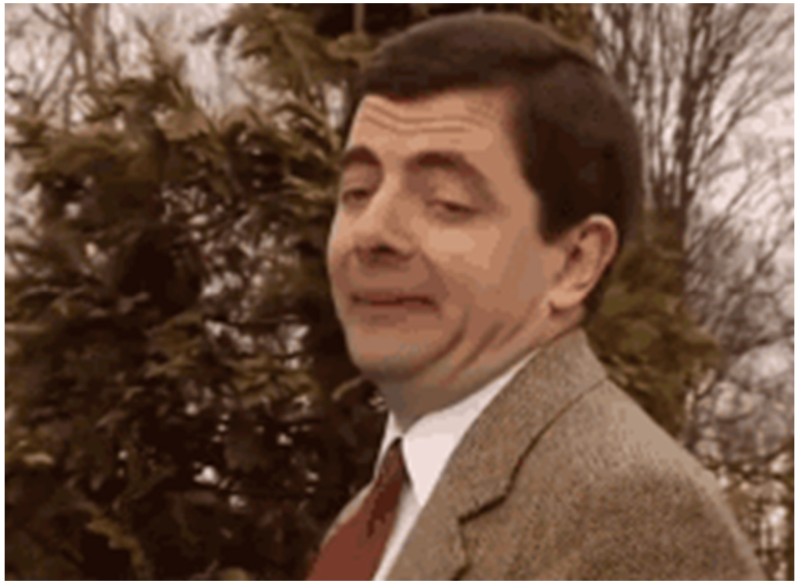
- With all these definitions in mind, let us now consider $H(K,K)$, i.e. K itself becomes the INPUT to H and H will determine whether K "$Halts$" or "$Loops\ Forever$".

The Halting Problem(Contd.)

- So, for K to give an output, it first needs to know the output of H.
  - If $OUTPUT\left(H(K,K)\right) = K$ "Halts", then $OUTPUT\left(K(K)\right) = K$ "Loops Forever".
  - If $OUTPUT\left(H(K,K)\right) = K$ "Loops Forever", then $OUTPUT\left(K(K)\right) = K$ "Halts".

The Halting Problem(Contd.)

- We knew that was easy!!!!!



The Halting Problem(Contd.)

- Wait…….WHAT!!!!!?????

The Halting Problem(Contd.)

- THIS IS NOT POSSIBLE!!!
- K cannot "$Halt$" and "$Loop\ Forever$" AT THE SAME TIME!!!
- Clearly this is a contradiction!!!
- Thus, we can conclude, "$The\ Halting\ Problem\ is\ UNSOLVABLE$" – courtesy of Alan Turing.

The Halting Problem(Contd.)

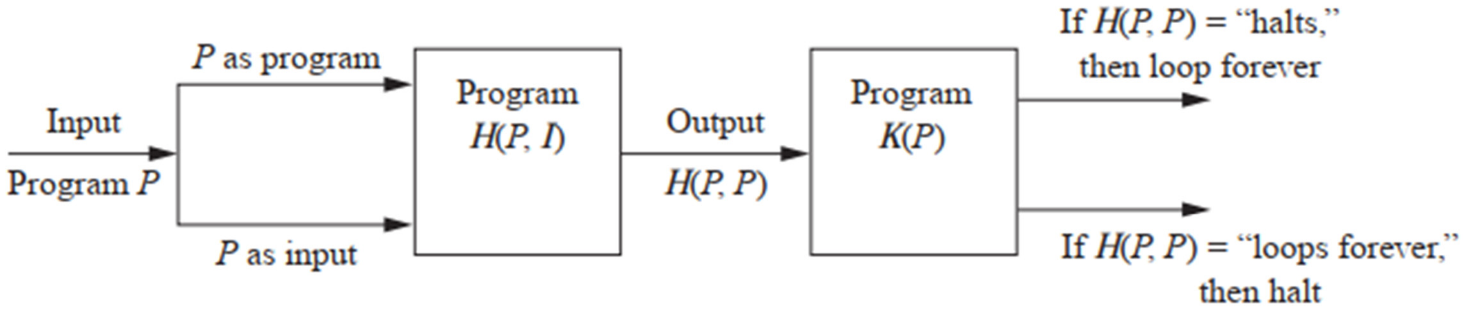


FIGURE 2 Showing that the Halting Problem is Unsolvable.

THE END