Discrete Mathematics[1]
http://iscasmc.ios.ac.cn/DM2016

Lijun Zhang

April 5, 2016

[1]Materials about the axiomatization are provided by Dr. Wanwei Liu.

Contents

Chapter 4

Algorithms

4.1 Syntax of an algorithm in pseudo-code

In this part, we consider an extension of the programming language **WHILE** presented in the Exercise sheet 2. Basically, the grammar is extended with *arrays* and *procedures*.

Let A be a finite alphabet, such as the English alphabet, and consider the following simple programming language **PROC**, an extension of **WHILE**, whose well-formed programs are those obtained by applying the rules of the following grammar:

Numbers:

$$Num ::= d \mid dNum \qquad \text{where } d \in \{0, 1, \ldots, 9\}$$

Identifiers:

$$\begin{aligned} Id &::= aId' && \text{where } a \in A \\ Id' &::= \lambda \mid aId' \mid NumId' \end{aligned}$$

Numeric expressions:

$$\begin{aligned} \mathit{Exp} ::= \mathit{Num} &\mid \mathit{Id} \mid \mathit{Id}[\mathit{Exp}] \mid \textbf{length}(\mathit{Id}) \mid (\mathit{Exp}) \\ &\mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} - \mathit{Exp} \mid \mathit{Exp} * \mathit{Exp} \mid \mathit{Exp}/\mathit{Exp} \\ &\mid \mathit{Id}(\mathit{Exp}, \ldots, \mathit{Exp}) \end{aligned}$$

Boolean expressions:

$$\begin{aligned} \mathit{BExp} ::= \textbf{true} &\mid \textbf{false} \mid \mathit{Exp} = \mathit{Exp} \mid (\mathit{BExp}) \\ &\mid \mathit{Exp} \leq \mathit{Exp} \mid \mathit{Exp} < \mathit{Exp} \\ &\mid \mathit{Exp} \geq \mathit{Exp} \mid \mathit{Exp} > \mathit{Exp} \\ &\mid \neg \mathit{BExp} \mid \mathit{BExp} \wedge \mathit{BExp} \mid \mathit{BExp} \vee \mathit{BExp} \end{aligned}$$

Procedure declaration

$$\begin{aligned} \mathit{Pr} ::= &\quad \textbf{procedure}\ \mathit{Id}(\mathit{Id}, \ldots, \mathit{Id})\ P \\ &\mid \textbf{procedure}\ \mathit{Id}(\mathit{Id}, \ldots, \mathit{Id})\ P;\ \textbf{return}\ \mathit{Exp} \end{aligned}$$

Programs:

$$\begin{aligned} P ::= \textbf{skip} &\mid \mathit{Id} := \mathit{Exp} \mid \mathit{Id}[\mathit{Exp}] := \mathit{Exp} \mid P; P \\ &\mid \textbf{if}\ \mathit{BExp}\ \textbf{then}\ P\ \textbf{else}\ P\ \textbf{fi} \\ &\mid \textbf{while}\ \mathit{BExp}\ \textbf{do}\ P\ \textbf{done} \\ &\mid \textbf{for}\ \mathit{Id} := \mathit{Exp}\ \textbf{to}\ \mathit{Exp}\ \textbf{do}\ P\ \textbf{done} \\ &\mid \mathit{Id}(\mathit{Exp}, \ldots, \mathit{Exp}) \end{aligned}$$

4.2 Examples of algorithms

Algorithm 4.2.1 (Maximum in a generic array)**.** The following pseudo-code in **PROC** returns the maximum value occurring in the provided `array`.

```
procedure max(array)
  maxvalue := array[0];
  for i := 1 to length(array) − 1 do
    if array[i] > maxvalue then
      maxvalue := array[i]
    else
      skip
    fi
  done;
  return maxvalue
```

$\Theta(n)$

Algorithm 4.2.2 (Index of a value in an array)**.** The following pseudo-code in **PROC** returns the index of the specified `value` if it occurs in the provided `array`, otherwise **length**(`array`) is returned.

```
procedure indexOf(value, array)
  index := length(array);
  for i := 0 to length(array) − 1 do
    if array[i] = value then
      index := i
    else
      skip
    fi
  done;
  return index
```

$O(n)$

Algorithm 4.2.3 (Index of a value in a sorted array)**.** The following pseudo-code in **PROC** returns the index of the specified `value` if it occurs in the provided sorted `array`, otherwise **length**(`array`) is returned.

```
procedure indexOfSorted(value, array)
  index := length(array);
  low := 0;
  high := length(array) − 1;
  while low < high do
    middle := (low + high)/2;
    if array[middle] < value then
      low := middle + 1
    else
      high := middle
    fi
  done;
  if array[low] = value then
    index := low
  else
    skip
  fi;
  return index
```

2^k

$\log(n)$

Algorithm 4.2.4 (Swap elements in an array)**.** The following pseudo-code in **PROC** swaps the elements at index `i` and `j` in the provided `array`.

```
procedure swap(array, i, j)
  temp := array[i];
  array[i] := array[j];
  array[j] := temp
```

$O(1)$

Algorithm 4.2.5 (Bubble sort)**.** The following pseudo-code in **PROC** sorts the provided `array`.

```
procedure bubbleSort(array)
  for i := 0 to length(array) − 1 do
    for j := 0 to (length(array) − 1) − i do
      if array[j] > array[j + 1] then
        swap(array, j, j + 1)
      else
        skip
      fi
    done
  done
```

$i=0 \quad n$

$i=1 \quad n-1$

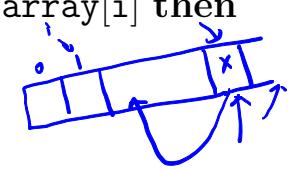
$\vdots$

$+ \quad i=n-1 \quad 1$

$O(n^2)$

Algorithm 4.2.6 (Gnome sort)**.** The following pseudo-code in **PROC** sorts the provided `array`.

```
procedure gnomeSort(array)
  i := 0;
  while i < length(array) do
    if i = 0 ∨ array[i − 1] ≤ array[i] then
      i := i + 1
    else
      swap(array, i, i − 1);
      i := i − 1
    fi
  done
```



$O(n^2)$

$O(n \lg n)$

Algorithm 4.2.7 (Insertion sort)**.** The following pseudo-code in **PROC** sorts the provided `array`.

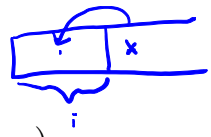
$i=1 \quad 1$

$i=2 \quad 2$

$\vdots$

$+ \quad i=n-1 \quad n-1$

$O(n^2)$



procedure `insertionSort`(`array`)
  for `i` := 1 **to length**(`array`) − 1 **do**
    `j` := `i`;
    while `j` > 0 ∧ `array`[`j` − 1] > `array`[`j`] **do**
      `swap`(`array`, `j`, `j` − 1);
      `j` := `j` − 1
    done
  done

Greedy Algorithms

Algorithm 4.2.8 (Change Making)**.** The algorithm makes changes $c_1 > c_2 \ldots > c_r$ for n cents.

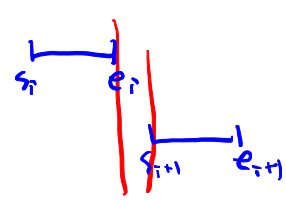
25, 10, 5, 1 ≤ 24

procedure `procedureChange`($c_1, c_2, \ldots, c_r$)
  for $i := 1$ **to** r **do**
    $d_i := 0$;
    while $n \geq c_i$ **do**
      $d_i := d_i + 1$;
      $n := n - c_i$
    done
  done

25	10	1

n = 30
1×25
+ 5×1
6

Algorithm 4.2.9 (Earliest Ending Time Job Scheduling)**.** Two jobs are *compatible* if they do not overlap. Assumption: sorts talks with ending times $e_1 < e_2 < \ldots e_n$.



procedure `schedule`($e_1, e_2, \ldots, e_n$)
  $S := \emptyset$;
  for $i := 1$ **to** n **do**
    if talk i is compatible with S then $S := S \cup \{talk\ i\}$
  done

$\phi := a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$

$\frac{|\phi_v|}{|\phi|}$

$O(2^{|\phi|})$

Remark: set not defined in **PROC**

Definition 4.2.10. *The* Tableau *approach for the satisfiability problem of propositional logic: a propositional formula is satisfiable iff it has a consistent tableau.*

- formula set Γ is sat iff $\bigwedge_{\phi \in \Gamma} \phi$ is sat
- $\{\phi\}$ root

$\Gamma \cup \{\phi\}$ $\quad$ Γ, ϕ $\quad$ $\phi = \begin{cases} \neg \\ \vee \\ \wedge \end{cases}$

$$\frac{\Gamma, \phi_1 \wedge \phi_2}{\Gamma, \phi_1, \phi_2} \wedge \qquad \frac{\Gamma, \phi_1 \vee \phi_2}{\Gamma, \phi_1} \vee_l \qquad \frac{\Gamma, \neg\neg\psi}{\Gamma, \psi} \neg\neg$$

$$\frac{\Gamma, \phi_1 \vee \phi_2}{\Gamma, \phi_2} \vee_r$$

- Γ is a tableau iff $\Gamma \subseteq AP \cup \{\neg p \mid p \in AP\}$
- Γ is consis. iff $\nexists\, a \in AP : a, \neg a \in \Gamma$

Definition 4.2.11 (Decidability).

- *A decision problem consists of a set of instances and a subset of yes-instances.*
- *The Primality problem: is the instance x a prime number?*
- *The answer (solution) to any decision problem is just one bit (true or false).*
- *A problem Q is* decidable *iff there is an algorithm A, such that for each instance q of Q, the computation $A(q)$ stops with an answer.*
- *A problem Q is semi-decidable iff there is an algorithm A, such that for each instance q of Q, if q holds, then $A(q)$ stops with the positive answer; otherwise, $A(q)$ either stops with the negative answer,* or does not stop.

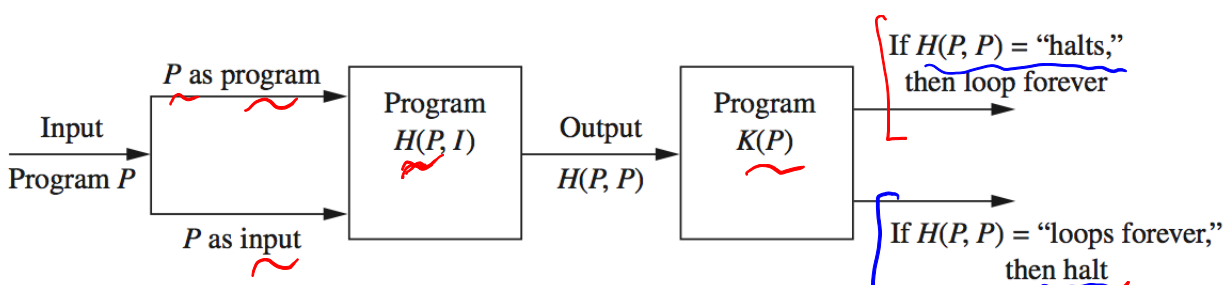
$H(A, I)$

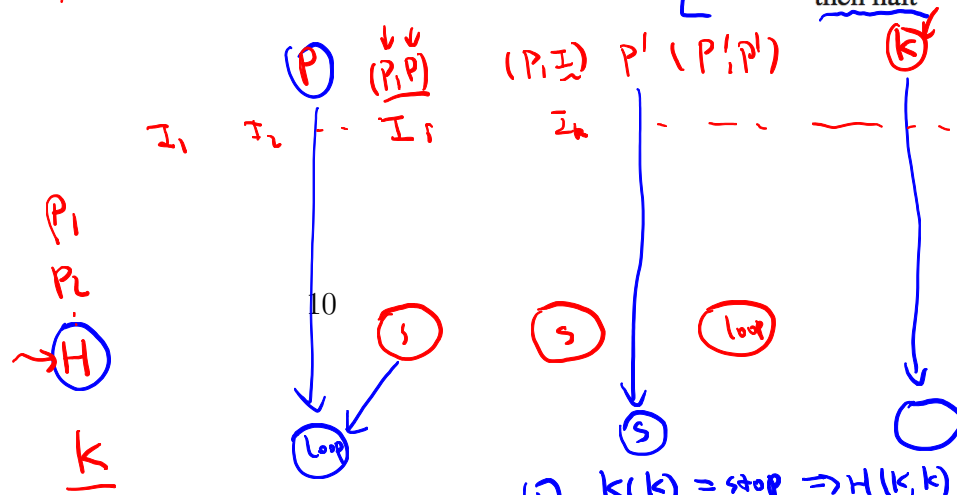
Definition 4.2.12 (The Halting Problem). *It takes as input a computer program and input to the program and determines whether the program will eventually stop when run with this input.*

- *If the program halts, we have our answer.*
- *If it is still running after any fixed length of time has elapsed, we do not know whether it will never halt or we just did not wait long enough for it to terminate.*

(2) $k(k) = \text{loop}$
$\Rightarrow H(k,k) = \text{halts}$
$\Rightarrow k(k) = \text{halts}$ ↯

Definition 4.2.13 (Undecidability of the Halting Problem).

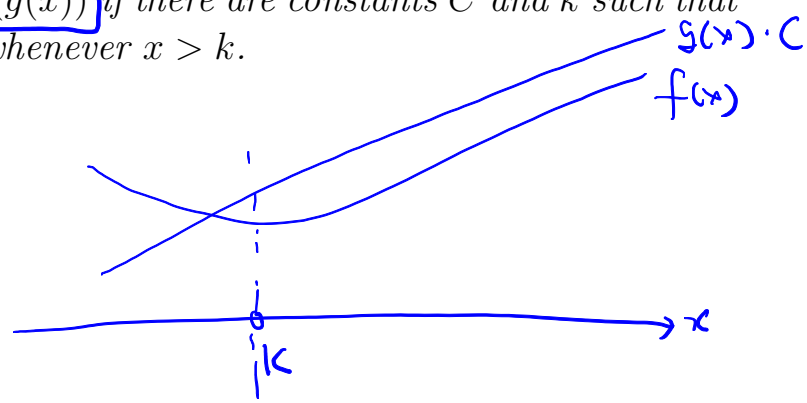




(1) $K(K) = \text{stop} \Rightarrow H(K,K) = \text{loop}$
$\Rightarrow K(K)$ loop ↯

4.3 The Growth of Functions

Definition 4.3.1. *Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $\boxed{f(x) \text{ is } O(g(x))}$ if there are constants C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$.*



Definition 4.3.2. *Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that $|f(x)| \geq C|g(x)|$ whenever $x > k$.*

Moreover, We say that $f(x)$ is $\underline{\Theta(g(x))}$ if $f(x)$ is $O(g(x))$ and $\Omega(g(x))$.

2. Assume n^2 is $O(n)$.

$\Rightarrow \exists\, C, k: \quad \underbrace{n^2 \le C \cdot n} \quad \forall\, n \ge k$

$\Rightarrow \underbrace{n \le C} \quad \lightning$

Exercise 4.3.3. *Show that:*

$\log n! \le \log n^n = n \log n$

1. *Show that log $n!$ is $O(nlogn)$.* $C = 1,\ k = 2$
2. *Show that n^2 is not $O(n)$.*
3. *Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$. Then $(f_1 + f_2)(x)$ is $O(max(|g_1(x)|, |g_2(x)|))$.* $C = 2,\ k = 1$
4. *Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.* $C =$ $\quad f_1 \le c_1 g_1$, $f_2 \le c_2 g_2$
5. *Show that $3x^2 + 8xlogx$ is $\Theta(x^2)$.*
6. *Assume $a_n \neq 0$. Show that $\sum_{i=0}^{n} a_i x^i$ is $\Theta(x^n)$.*

5: $O(x^2)$: $\begin{cases} C = 12,\ k = 1 \\ C = 4,\ k \end{cases}$

$\Omega(x^2)$: $C = 1$

$x \ge 1$

$\sum_{i=0}^{n} a_i x^i \le c \cdot x^n$

$\le x^n \boxed{\sum_{i=1}^{n} a_i}$

6. $O(x^n)$: $C =$

$k = 1$

$\Omega(x^n)$: $C = a_n$

4.4 Complexity of Algorithms

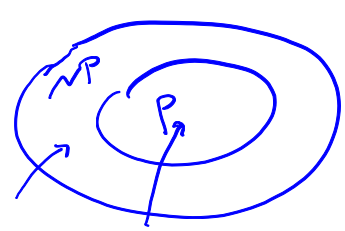
We are interested in the *time complexity* of the algorithm.

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Definition 4.4.1.

- *A problem that is solvable using an algorithm with polynomial worst-case complexity is called tractable. Tractable problems are said to belong to class P.*
- *Problems for which a solution can be checked in polynomial time are said to belong to the class NP.* non-deterministic
- *The $P = NP$ problem asks whether NP, the class of problems for which it is possible to check solutions in polynomial time, equals P, the class of tractable problems.*
- *NP-complete problems: It is an NP problem and if a polynomial time algorithm for solving it were known, then $P = NP$.*



- *The satisfiability problem is NP-complete.* *Cook-Levin Theorem*

A prize of $1,000,000$ *dollars is offered by the Clay Mathematics Institute for its solution.*