Theory of Computation 10
Complexity Considerations

Frank Stephan

Department of Computer Science

Department of Mathematics

National University of Singapore

fstephan@comp.nus.edu.sg

Repetition 1

Models of Computation

- Turing Machine: States like Finite Automaton plus Turing tape carrying input/output and working space; head of machine working and moving on tape; updates of symbols, states and movement given by Turing table.
- Register Machine: Adding and Subtracting and Comparing natural numbers in registers; conditional and unconditional jumps between numbered statements.
- Primitive recursive and μ-recursive functions: Functions defined from some base functions together with concatenation, primitive recursion and, in the case of μ-recursive functions, search for places where some condition holds.

Repetition 2

Example: Multiplication can be done naively by repeated addition.

Line 1: Function Mult$(\mathbf{R_1}, \mathbf{R_2})$;
Line 2: $\mathbf{R_3} = \mathbf{0}$;
Line 3: $\mathbf{R_4} = \mathbf{0}$;
Line 4: If $\mathbf{R_3} = \mathbf{R_1}$ Then Goto Line 8;
Line 5: $\mathbf{R_4} = \mathbf{R_4} + \mathbf{R_2}$;
Line 6: $\mathbf{R_3} = \mathbf{R_3} + \mathbf{1}$;
Line 7: Goto Line 4;
Line 8: Return$(\mathbf{R_4})$.

Repetition 3

Primitive Recurisive: Addition, Multiplication, Subtraction, Exponentiation, Factorial, Choose-Function, Outcomes of Comparisons, Linear Functrions and Polynomials.

Not Primitive Recursive: Ackermann Function:

- $\mathbf{f(0,y) = y+1}$;
- $\mathbf{f(x+1,0) = f(x,1)}$;
- $\mathbf{f(x+1,y+1) = f(x,f(x+1,y))}$.

Partial Recursive: Primitive recursive plus search for an input which makes function to $\mathbf{0}$.

Repetition 4

Theorem 9.23

For a partial function f, the following are equivalent:

- f as a function from strings to strings can be computed by a Turing machine;
- f as a function from natural numbers to natural numbers can be computed by a register machine;
- f as a function from natural numbers to natural numbers is partial recursive.

Church's Thesis

All reasonable models of computation over Σ^* and $\mathbb{N}$ are equivalent and give the same notion as the partial recursive functions.

Regular Sets and Turing Machines I

Theorem 10.1. A Turing machine with only one tape can recognise in linear time of the input size exactly the regular sets.

Proof Idea Trakhtenbrot as well as Hartmanis proved that a one-tape Turing machine running in linear time visits each cell at most k times for some constant k.

Instead of now modifying overwriting the tape symbol each time, one enlarges the work tape alphabet and creates symbols which are tuples holding for each of the visit all information in a coordinate (number of visit,state, new symbol, direction). Thus the symbols are now $4k + 1$-tuples of “basic entries” which give a full picture of the Turing machine behaviour when looking at it.

Regular Sets and Turing Machines II

The full result of the computation can be verified by a finite automaton reading it from the left to the right.

An nfa can guess all these entries while scanning the input word from the left to the right and memorise them for the current and previous symbol in each step in the state.

While guessing and processing, it can verify that the guessed information for the next symbol fits with that for the current symbol and gives a full accepting run of the Turing machine. If the verification and guessing goes through and the computation ends in an accepting state then the nfa accepts the word else it rejects the word.

Recall acceptance means that there is at least one accepting computation and at least one accepting run of the nfa.

Turing machine power

Exercise 10.2. Extend the proof sketch from previous slide to a full proof and fill also in the details of the result of Trakhtenbrot and Hartmanis. That is, show that every language recognised by a linear time one-tape Turing machine is regular.

Exercise 10.3. Show that a Turing machine with two tapes can recognise the set of palindromes which is not regular.

Note the same applies to Turing machines which move with two independent heads over the input, even if these do not write anything but only read the symbols. Here the idea is that one head goes to the end and the other one to the start of the word. Then they move in opposite direction over the word, one symbol per cycle and compare the symbols. If both heads reach the opposite end of the word and the corresponding symbols were always the same then the word is a palindrome else it is not a planindrome.

Register versus Turing machines

Complexities	Register Machine Floyd, Knuth [1990]	Multitape TM Turing [1936]
Addition Subtraction Comparison	$\mathbf{1}$ $\mathbf{1}$ $\mathbf{1}$	$\boldsymbol{\Theta}(\mathbf{n})$ $\boldsymbol{\Theta}(\mathbf{n})$ $\boldsymbol{\Theta}(\mathbf{n})$
Multiplication	$\boldsymbol{\Theta}(\mathbf{n})$	$\mathbf{O}(\mathbf{n}\log\mathbf{n})$
Bitwise And, Or, ...	$\boldsymbol{\Theta}(\mathbf{n})$	$\boldsymbol{\Theta}(\mathbf{n})$
Doubling Halving	$\mathbf{1}$ $\boldsymbol{\Theta}(\mathbf{n})$	$\mathbf{O}(\mathbf{n})$ $\mathbf{O}(\mathbf{n})$
Regular Set	$\boldsymbol{\Theta}(\mathbf{n})$	$\boldsymbol{\Theta}(\mathbf{n})$

Complexity Class P

Size parameter $\mathbf{n}$ (for example logarithm of input number, number of digits); Algorithm is in polynomial time iff it needs time polynomial in $\mathbf{n}$ to solve the problem, that is, time $(\mathbf{n}+\mathbf{k})^{\mathbf{k}}$ for some $\mathbf{k}>\mathbf{0}$. For certain problem classes, other parameters $\mathbf{n}$ are used like number of states of automaton. Sometimes several parameters are there (number of nodes and number of edges in graph).

Polynomial Time Computations can be used to describe both the solving of decision problems in polynomial time (Class $\mathbf{P}$) and computing functions in polynomial time.

Equivalence of Computation Models

Consider two compoutation models $\mathbf{C}$ and $\mathbf{D}$. $\mathbf{C}$ cannot compute more than $\mathbf{D}$ in polynomial time iff there is a polynomial $\mathbf{p}$ and a constant $\mathbf{c}$ such that the following holds:
1. Every primitive operation in $\mathbf{C}$ on inputs of length $\mathbf{n}$ can be done in time $\mathbf{p(n)}$ steps in $\mathbf{D}$;
2. Every primitive operation of $\mathbf{C}$ with inputs up to length $\mathbf{n}$ produces only outputs of length up to $\mathbf{n + c}$.

Let $\mathbf{p, q}$ be polynomials. If algo using $\mathbf{C}$ needs $\mathbf{q(n)}$ steps on inputs of length $\mathbf{n}$ then all intermediate values have at most length $\mathbf{n + c \cdot q(n)}$ and need $\mathbf{q(p(n + c \cdot q(n)))}$ steps according to model $\mathbf{D}$.

Example 10.5

Register Machine model of Floyd and Knuth satisfies both conditions compared to multitape Turing machines.

First, multitape Turing machines can add, subtract and compare $\mathbf{n}$-bit numbers in $\mathbf{O(n)}$ steps.
Second, comparing is $\{\mathbf{0}, \mathbf{1}\}$-valued and adding or subtracting increases the bits needed to store the result at most by $\mathbf{1}$.

Harmanis and Simon investigated a model allowing adding, subtracting, comparing and bitwise and / or / eor to combine binary numbers, one still is in polynomial time; however, when the machine can also multiply in unit time, it can in polynomial time solve all $\mathbf{NP}$ and $\mathbf{PSPACE}$ problems.

Non-Determinism

Non-deterministic register computations can computations can guess numbers where the values generated by non-determinism should be between 0 and the value of a given register; non-deterministic Turing machines can choose a new state non-deterministically in dependence of choices in the Turing table.

The class $\mathbf{NP}$ is that of all sets $\mathbf{A}$ where a non-deterministic computation produces an accepting computation (among many choices) on members of $\mathbf{A}$ and does not produce any accepting computation on non-members of $\mathbf{A}$. Furthermore, the run-time of the computation has always to be bounded by a polynomial in the input-size.

Idea: $\mathbf{NP}$ is the class of problems where one can prove the solvability of an instance by guessing a polynomial-sized solution and then verifying the solution in polynomial time.

Example Provable Speed-Up

Stockmeyer proved that deterministic register machines need $\Theta(n)$ steps to determine whether a number is odd.
A non-deterministic computation is faster:

Line 1: Function Remainderbytwo(R_1);
Line 2: Guess R_2 from range $\{0, 1, \ldots, R_1\}$;
Line 3: $R_3 = R_2 + R_2$;
Line 4: If $R_3 = R_1$ Then Return 0;
Line 5: $R_3 = R_3 + 1$;
Line 6: If $R_3 = R_1$ Then Return 1;
Line 7: Reject Computation.

The program guesses $R_2 = \text{floor}(R_1/2)$ and checks whether R_1 is $2R_2$ or $2R_2 + 1$.

Computations must verify whether the result based on guesses is correct and reject computations based on wrong guesses.

NP-Complete Problems

Definition 10.8. A problem $\mathbf{A}$ is $\mathbf{NP}$-complete iff for every further problem $\mathbf{B}$ in $\mathbf{NP}$ there is a polynomial time computable function $\mathbf{f}$ such that for all $\mathbf{B}$-instances $\mathbf{x}$, $\mathbf{x} \in \mathbf{B}$ iff $\mathbf{f}(\mathbf{x}) \in \mathbf{A}$.

This implies that if one can solve $\mathbf{A}$ in polynomial time then one can solve all $\mathbf{NP}$-problems in polynomial time.

Example 10.9. Manders and Adleman [1976] showed that set of all $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathbb{N}^3$ for which there are $\mathbf{x}, \mathbf{y} \in \mathbb{N}$ with $\mathbf{a} \cdot \mathbf{x}^2 + \mathbf{b} \cdot \mathbf{y} = \mathbf{c}$ is $\mathbf{NP}$-complete.

A nondeterministic register machine reads $\mathbf{a}, \mathbf{b}, \mathbf{c}$, guesses $\mathbf{x}, \mathbf{y}$ with range $\{0, 1, \ldots, \mathbf{c}\}$ and then checks whether the equation $\mathbf{a} \cdot \mathbf{x}^2 + \mathbf{b} \cdot \mathbf{y} = \mathbf{c}$ is satisfied. If so then it ouputs $\mathbf{YES}$ else it rejects the computation. The solvable instances are those which have an accepting computation outputting $\mathbf{YES}$.

10.10 Satisfiability Problems

A literal is a term of the form $\mathbf{x}$ or $\neg\mathbf{x}$ or $\mathbf{0}$ or $\mathbf{1}$ for a Boolean variable $\mathbf{x}$. A clause is a disjunction of literals like $\mathbf{x} \vee \mathbf{y} \vee \neg\mathbf{z}$ or $\mathbf{u} \vee \mathbf{0}$. An instance is a conjunction of clauses.

$\mathbf{SAT}$ consists of all satisfiability instances for which there is an assignment of Boolean values to the variables such that all clauses are true, that is, such that in every clause some literal is evaluated to $\mathbf{1}$. A $\mathbf{kSAT}$ instance is an instance of satisfiability in which every clause has at most $\mathbf{k}$ literals.

While $\mathbf{1SAT}$ and $\mathbf{2SAT}$ are in $\mathbf{P}$, the problems $\mathbf{3SAT}$, $\mathbf{4SAT}$, ... and $\mathbf{SAT}$ itself are all $\mathbf{NP}$-complete.

The complexity parameters of an instance are the number $\mathbf{n}$ of variables used and the number $\mathbf{m}$ of clauses.

Graph Problems

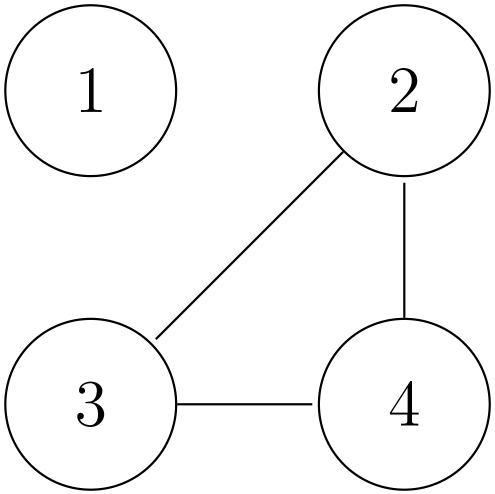
Parameters numbers $\mathbf{n}$ of nodes and $\mathbf{m}$ of edges.

Example 10.11: CLIQUE. Let $(\mathbf{V}, \mathbf{E})$ be a graph and $\mathbf{k} \leq \mathbf{n}$. Now $\mathbf{CLIQUE}$ is the set of all such $(\mathbf{V}, \mathbf{E}, \mathbf{n}, \mathbf{k})$ for which $\mathbf{n} = |\mathbf{V}|$ and for which there is a subset $\mathbf{W} \subseteq \mathbf{V}$ of $\mathbf{k}$ nodes such that each two distinct nodes in $\mathbf{W}$ are connected by an edge in $\mathbf{E}$. $\mathbf{CLIQUE}$ is $\mathbf{NP}$-complete.

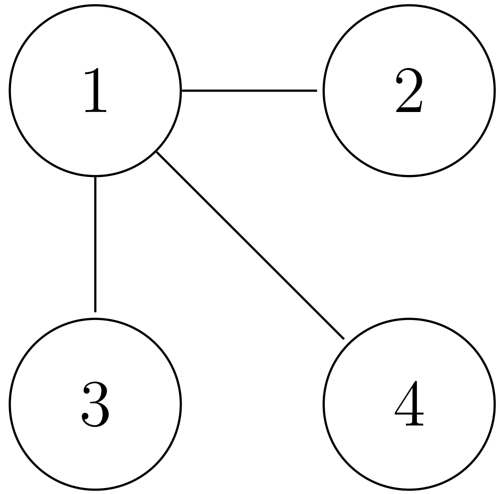
Example 10.12: INDEPENDENTSET. Let $(\mathbf{V}, \mathbf{E})$ be a graph and $\mathbf{k} \leq \mathbf{n}$. Now $\mathbf{INDEPENDENT}$ is the set of all such $(\mathbf{V}, \mathbf{E}, \mathbf{n}, \mathbf{k})$ for which $\mathbf{n} = |\mathbf{V}|$ and for which there is a subset $\mathbf{W} \subseteq \mathbf{V}$ of $\mathbf{k}$ nodes such that no two distinct nodes in $\mathbf{W}$ are connected by an edge in $\mathbf{E}$. $\mathbf{INDPENDENTSET}$ is $\mathbf{NP}$-complete.

Example of Reduction

Recall that for $\mathbf{NP}$-complete problems $\mathbf{B}$ and further $\mathbf{NP}$-problems $\mathbf{A}$ there is a polynomial time computable function $\mathbf{f}$ with $\mathbf{A(x) = B(f(x))}$ for all instances $\mathbf{x}$.



is mapped to



The graphic illustrates a polynomial-time many-one reduction from $\mathbf{CLIQUE}$ to $\mathbf{INDEPENDENTSET}$ where the mapping exchanges edge and non-edge between each possible pair of nodes. So a $\mathbf{CLIQUE}$ of size $\mathbf{3}$ is mapped to an independent set of size $\mathbf{3}$ and $\mathbf{(V, E, 4, k)}$ is in $\mathbf{CLIQUE}$ iff $\mathbf{(V, f(V, E), 4, k)}$ is in $\mathbf{INDEPENDENTSET}$.

10.16. Polynomial Space

A register machine uses polynomial space iff there is a polynomial $\mathbf{p}$ such that for inputs all between -2^n and 2^n, all registers are all the time between $-2^{p(n)}$ and $2^{p(n)}$. $\mathbf{PSPACE}$ contains all problems solvable by a register program using polynomial space; exhaustive search shows that $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

Example 10.17. A $\mathbf{PSPACE}$-complete problem is quantified satisfiability $\mathbf{QSAT}$ where there are variables $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ and where $\mathbf{QSAT}$ is the set of all instances (F, n, m) of m clauses which satisfy that for all x_1 there is y_1 for all x_2 there is y_2 ... for all x_n there is y_n such that all clauses in F are satisfied.

Exercises 10.18 and 10.19

Exercise 10.18

Make a proof that every deterministic register program whose space bound (register size) is bounded by $\mathbf{p(n)}$ bits throughout the overall time and which always halts runs at most in time $\mathbf{2^{O(p(n))}}$ for the same polynomial $\mathbf{p}$.

Exercise 10.19

Show that the following problem is in $\mathbf{NP}$:

$\mathbf{CONNECTEDHALVES}$ is the set of all graphs $\mathbf{(V, E)}$ such that one can split $\mathbf{V}$ into two subsets $\mathbf{U, W}$ such that $\mathbf{|U| \leq |W| \leq |U| + 1}$ and every node in $\mathbf{U}$ is connected to every node in $\mathbf{W}$ by an edge.

Primitive Recursive

Theorem
A function is primitive recursive iff it can be computed by a register program where the only type of goto-commands which can go backwards are For-Loops, where one cannot go into or out of a For-Loop and once the For-Loop is started, its boundaries cannot be modified and the loop-variable can only be updated by the commands of the loop itself.

Remark
One can replace the Goto-commands completely by allowing only For-Loops, If-Then-Else statements and Switch-statements which are properly nested.

For full generality of Partial-Recursive functions, one would then also need While-Loops in addition to the For-Loops.

Example

Line 1: Function Factor$(\mathbf{R_1}, \mathbf{R_2})$;
Line 2: $\mathbf{R_3} = \mathbf{R_1}$;
Line 3: $\mathbf{R_4} = \mathbf{0}$;
Line 4: If $\mathbf{R_2} < \mathbf{2}$ Then Goto Line 10;
Line 5: For $\mathbf{R_5} = \mathbf{0}$ to $\mathbf{R_1}$
Line 6: If Remainder$(\mathbf{R_3}, \mathbf{R_2}) > \mathbf{0}$ Then Goto Line 9;
Line 7: $\mathbf{R_3} =$ Divide$(\mathbf{R_3}, \mathbf{R_2})$;
Line 8: $\mathbf{R_4} = \mathbf{R_4} + \mathbf{1}$;
Line 9: Next $\mathbf{R_5}$;
Line 10: Return$(\mathbf{R_4})$.

This function computes how often $\mathbf{R_2}$ is a factor of $\mathbf{R_1}$ and is primitive recursive.

Collatz Function

Not known whether primitive recursive or whether total at all.

Line 1: Function Collatz$(\mathbf{R_1})$;
Line 2: If Remainder$(\mathbf{R_1}, \mathbf{2}) = \mathbf{0}$ Then Goto Line 6;
Line 3: If $\mathbf{R_1} = \mathbf{1}$ Then Goto Line 8;
Line 4: $\mathbf{R_1} =$ Mult$(\mathbf{R_1}, \mathbf{3}) + \mathbf{1}$;
Line 5: Goto Line 2;
Line 6: $\mathbf{R_1} =$ Divide$(\mathbf{R_1}, \mathbf{2})$;
Line 7: Goto Line 2;
Line 8: Return$(\mathbf{R_1})$.

Lothar Collatz conjectured in 1937 that this function is total.

Simulating Collatz Function

Line 1: Function Collatz$(\mathbf{R_1}, \mathbf{R_2})$;
Line 2: $\mathbf{LN} = \mathbf{2}$;
Line 3: For $\mathbf{T} = \mathbf{0}$ to $\mathbf{R_2}$
Line 4: If $\mathbf{LN} = \mathbf{2}$ Then Begin If Remainder$(\mathbf{R_1}, \mathbf{2}) = \mathbf{0}$
Then $\mathbf{LN} = \mathbf{6}$ Else $\mathbf{LN} = \mathbf{3}$; Goto Line 10 End;
Line 5: If $\mathbf{LN} = \mathbf{3}$ Then Begin If $\mathbf{R_1} = \mathbf{1}$ Then $\mathbf{LN} = \mathbf{8}$
Else $\mathbf{LN} = \mathbf{4}$; Goto Line 10 End;
Line 6: If $\mathbf{LN} = \mathbf{4}$ Then Begin $\mathbf{R_1} =$ Mult$(\mathbf{R_1}, \mathbf{3}) + \mathbf{1}$;
$\mathbf{LN} = \mathbf{5}$; Goto Line 10 End;
Line 7: If $\mathbf{LN} = \mathbf{5}$ Then Begin $\mathbf{LN} = \mathbf{2}$; Goto Line 10 End;
Line 8: If $\mathbf{LN} = \mathbf{6}$ Then Begin $\mathbf{R_1} =$ Divide$(\mathbf{R_1}, \mathbf{2})$;
$\mathbf{LN} = \mathbf{7}$; Goto Line 10 End;
Line 9: If $\mathbf{LN} = \mathbf{7}$ Then Begin $\mathbf{LN} = \mathbf{2}$; Goto Line 10 End;
Line 10: Next $\mathbf{T}$;
Line 11: If $\mathbf{LN} = \mathbf{8}$ Then Return$(\mathbf{R_1} + \mathbf{1})$ Else Return$(\mathbf{0})$.

Exercise 10.20

Write a program for a primitive recursive function which simulate the following function with input $\mathbf{R_1}$ for $\mathbf{R_2}$ steps.

Line 1: Function Expo$(\mathbf{R_1})$;
Line 2: $\mathbf{R_3 = 1}$;
Line 3: If $\mathbf{R_1 = 0}$ Then Goto Line 7;
Line 4: $\mathbf{R_3 = R_3 + R_3}$;
Line 5: $\mathbf{R_1 = R_1 - 1}$;
Line 6: Goto Line 3;
Line 7: Return$(\mathbf{R_3})$.

Exercise 10.21

Write a program for a primitive recursive function which simulate the following function with input $\mathbf{R_1}$ for $\mathbf{R_2}$ steps.

Line 1: Function Repeatadd$(\mathbf{R_1})$;
Line 2: $\mathbf{R_3 = 3}$;
Line 3: If $\mathbf{R_1 = 0}$ Then Goto Line 7;
Line 4: $\mathbf{R_3 = R_3 + R_3 + R_3 + 3}$;
Line 5: $\mathbf{R_1 = R_1 - 1}$;
Line 6: Goto Line 3;
Line 7: Return$(\mathbf{R_3})$.

Bounded Simulation

Theorem 10.22

For every partial-recursive function $\mathbf{f}$ there is a primitive recursive function $\mathbf{g}$ and a register machine $\mathbf{M}$ such that for all $\mathbf{t}$,

> If $\mathbf{f(x_1, \ldots, x_n)}$ is computed by $\mathbf{M}$ within $\mathbf{t}$ steps
> Then $\mathbf{g(x_1, \ldots, x_n, t) = f(x_1, \ldots, x_n) + 1}$
> Else $\mathbf{g(x_1, \ldots, x_n, t) = 0}$.

In short words, $\mathbf{g}$ simulates the program $\mathbf{M}$ of $\mathbf{f}$ for $\mathbf{t}$ steps and if an output $\mathbf{y}$ comes then $\mathbf{g}$ outputs $\mathbf{y + 1}$ else $\mathbf{g}$ outputs $\mathbf{0}$.

Recursively Enumerable

Theorem 10.24

The following notions are equivalent for a set $A \subseteq \mathbb{N}$:

(a) A is the range of a partial recursive function;

(b) A is empty or A is the range of a total recursive function;

(c) A is empty or A is the range of a primitive recursive function;

(d) A is the set of inputs on which some register machine terminates;

(e) A is the domain of a partial recursive function;

(f) There is a two-place recursive function g such that $A = \{x : \exists y\, [g(x, y) > 0]\}$.

Definition 10.25

The set A is recursively enumerable iff it satisfies any of the above equivalent properties.

(a) to (c) and (c) to (b)

If $\mathbf{A}$ is empty then $(\mathbf{c})$ holds; if $\mathbf{A}$ is not empty then there is an element $\mathbf{a} \in \mathbf{A}$ which is now taken as a constant. For the partial function $\mathbf{f}$ whose range $\mathbf{A}$ is, there is, by Theorem 10.22, a primitive function $\mathbf{g}$ such that either $\mathbf{g}(\mathbf{x}, \mathbf{t}) = \mathbf{0}$ or $\mathbf{g}(\mathbf{x}, \mathbf{t}) = \mathbf{f}(\mathbf{x}) + \mathbf{1}$ and whenever $\mathbf{f}(\mathbf{x})$ takes a value there is also a $\mathbf{t}$ with $\mathbf{g}(\mathbf{x}, \mathbf{t}) = \mathbf{f}(\mathbf{x}) + \mathbf{1}$. Now one defines a new function $\mathbf{h}$ which is also primitive recursive such that if $\mathbf{g}(\mathbf{x}, \mathbf{t}) = \mathbf{0}$ then $\mathbf{h}(\mathbf{x}, \mathbf{t}) = \mathbf{a}$ else $\mathbf{h}(\mathbf{x}, \mathbf{t}) = \mathbf{g}(\mathbf{x}, \mathbf{t}) - \mathbf{1}$. The range of $\mathbf{h}$ is $\mathbf{A}$.

$(\mathbf{c}) \Rightarrow (\mathbf{b})$: This follows by definition as every primitive recursive function is also recursive.

(b) to (d) and (d) to (e)

$(\mathbf{b}) \Rightarrow (\mathbf{d})$: Given a function $\mathbf{h}$ whose range is $\mathbf{A}$, one can make a register machine which simulates $\mathbf{h}$ and searches over all possible inputs and checks whether $\mathbf{h}$ on these inputs is $\mathbf{x}$. If such inputs are found then the search terminates else the register machine runs forever. Thus $\mathbf{x} \in \mathbf{A}$ iff the register machine program following this behaviour terminates after some time.

$(\mathbf{d}) \Rightarrow (\mathbf{e})$: The domain of a register machine is the set of inputs on which it halts and outputs a return value. Thus this implication is satisfied trivially by taking the function for $(\mathbf{e})$ to be exactly the function computed from the register program for $(\mathbf{d})$.

(e) to (f) and (f) to (a)

$(\mathbf{e}) \Rightarrow (\mathbf{f})$: Given a register program $\mathbf{f}$ whose domain $\mathbf{A}$ is according to $(\mathbf{e})$, one takes the function $\mathbf{g}$ as defined by Theorem 10.22 and this function indeed satisfies that $\mathbf{f}(\mathbf{x})$ is defined iff there is a $\mathbf{t}$ such that $\mathbf{g}(\mathbf{x}, \mathbf{t}) > \mathbf{0}$.

$(\mathbf{f}) \Rightarrow (\mathbf{a})$: Given the function $\mathbf{g}$ as defined in $(\mathbf{f})$, one defines that if there is a $\mathbf{t}$ with $\mathbf{g}(\mathbf{x}, \mathbf{t}) > \mathbf{0}$ then $\mathbf{f}(\mathbf{x}) = \mathbf{x}$ else $\mathbf{f}(\mathbf{x})$ is undefined. The latter comes by infinite search for a $\mathbf{t}$ which is not found. Thus the partial recursive function $\mathbf{f}$ has range $\mathbf{A}$.

Decidable and Undecidable Problems

A set $\mathbf{L}$ is called decidable or recursive iff there is a recursive function $\mathbf{f}$ such that, for all $\mathbf{x}$, if $\mathbf{x} \in \mathbf{L}$ then $\mathbf{f(x) = 1}$ else $\mathbf{f(x) = 0}$. One says that the function $\mathbf{f}$ decides the membership in $\mathbf{L}$.

A set $\mathbf{L}$ is called undecidable or nonrecursive iff there is no such recursive function $\mathbf{f}$ deciding the membership in $\mathbf{L}$.

Observation
Every recursive set is recursively enumerable.

The Halting Problem

Definition 10.27 [Turing 1936]
Let $\mathbf{e}, \mathbf{x} \mapsto \varphi_{\mathbf{e}}(\mathbf{x})$ be a universal partial recursive function covering all one-variable partial recursive functions. Then the set $\{(\mathbf{e}, \mathbf{x}) : \varphi_{\mathbf{e}}(\mathbf{x})$ is defined$\}$ is called the general halting problem and $\mathbf{K} = \{\mathbf{e} : \varphi_{\mathbf{e}}(\mathbf{e})$ is defined$\}$ is called the diagonal halting problem.

The name stems from the fact that $\varphi_{\mathbf{e}}(\mathbf{x})$ is defined iff the $\mathbf{e}$-th register machine with input $\mathbf{x}$ halts and produces some output.

Theorem 10.28 [Turing 1936]
Both the diagonal halting problem and the general halting problem are recursively enumerable and undecidable.

Proof

Let $\mathbf{F}$ be a function with inputs $\mathbf{e}, \mathbf{x}$ which simulates $\varphi_{\mathbf{e}}(\mathbf{x})$ and $\mathbf{F}(\mathbf{e}, \mathbf{x}) = \varphi_{\mathbf{e}}(\mathbf{x})$ if and only if this simulation terminates. Note that $\mathbf{F}$ exists and is partial-recursive.

Assume that there is a function $\mathbf{Halt}$ which can check whether $\varphi_{\mathbf{e}}(\mathbf{e})$ halts. If so, then $\mathbf{Halt}(\mathbf{e}) = \mathbf{1}$ else $\mathbf{Halt}(\mathbf{e}) = \mathbf{0}$.

Now condider this program.

Line 1: Function Diagonalise$(\mathbf{R_1})$;
Line 2: $\mathbf{R_2} = \mathbf{0}$;
Line 3: If $\mathbf{Halt}(\mathbf{R_1}) = \mathbf{0}$ Then Goto Line 5;
Line 4: $\mathbf{R_2} = \mathbf{F}(\mathbf{R_1}, \mathbf{R_1}) + \mathbf{1}$;
Line 5: Return$(\mathbf{R_2})$.

Function Diagonalise

The function Diagonalise has only one input.

If $\varphi_e(e)$ is undefined then $\mathbf{Halt(e) = 0}$ and $\mathbf{Diagonalise(e) = 0}$.

If $\varphi_e(e)$ is defined then $\mathbf{Halt(e) = 1}$ and $\mathbf{F(e,e)} = \varphi_e(e)$ will be computed in Line 4 and the output will be $\varphi_e(e) + 1$.

Thus $\mathbf{Diagonalise(e)}$ differs from $\varphi_e(e)$ for all $\mathbf{e}$ and is not among $\varphi_0, \varphi_1, \ldots$; as all partial-recursive functions with one input are in this list, neither $\mathbf{Diagonalise}$ nor $\mathbf{Halt}$ can be recursive.

The halting problem equals $\{(\mathbf{e}, \mathbf{x}) : \mathbf{F}(\mathbf{e}, \mathbf{x})$ halts$\}$. Thus it is the domain of a partial recursive function and recursively enumerable. Similarly, the diagonal halting problem $\mathbf{K} = \{\mathbf{e} : \mathbf{F}(\mathbf{e}, \mathbf{e})$ halts$\}$ is the domain of a partial-recursive function and recursively enumerable.

R.E. and Recursive

Theorem 10.29

A set $\mathbf{L}$ is recursive iff both $\mathbf{L}$ and $\mathbb{N} - \mathbf{L}$ are recursively enumerable.

Exercise 10.30

Prove this connection.

Exercises 10.31 – 10.33

Prove that the following variants of the halting problem are undecidable:

10.31: $\{\mathbf{e} : \varphi_{\mathbf{e}}(\mathbf{2e+5})$ is defined$\}$;

10.32: $\{\mathbf{e} : \varphi_{\mathbf{e}}(\mathbf{e^2+1})$ is defined$\}$;

10.33: $\{\mathbf{e} : \varphi_{\mathbf{e}}(\mathbf{e/2})$ is defined$\}$, where $\mathbf{1/2}$ is rounded to $\mathbf{0}$ and $\mathbf{3/2}$ to $\mathbf{1}$ and so on.

Further Homeworks 10.34-10.36

Show that the following sets are recursively enumerable by proving that a register machine halts exactly on the members of the set:
Exercise 10.34: $\{x \in \mathbb{N} : x$ is a square$\}$.
Exercise 10.35: $\{x \in \mathbb{N} : x$ is prime$\}$.

Exercise 10.36
Prove that the set $\{e : \varphi_e(e/2)$ is defined$\}$ is recursively enumerable by proving that it is the range of a primitive recursive function. Here $e/2$ is the downrounded value of e divided by 2, so $1/2$ is 0 and $3/2$ is 1.

Further Homeworks 10.37-10.39

Exercise 10.37: Prove or disprove: Every recursively enumerable set is either $\emptyset$ or the range of a function which can be computed in polynomial time.

Exercise 10.38: Prove or disprove: Every recursively enumerable set is either $\emptyset$ or the domain of a function $\mathbf{f}$ where the graph $\{(\mathbf{x}, \mathbf{f}(\mathbf{x})) : \mathbf{x} \in \mathbf{dom}(\mathbf{f})\}$ can be decided in polynomial time, that is, given inputs $\mathbf{x}, \mathbf{y}$, one can decide in polynomial time whether $(\mathbf{x}, \mathbf{y}) = (\mathbf{x}, \mathbf{f}(\mathbf{x}))$.

Exercise 10.39: Prove or disprove: Every recursively enumerable set is either $\emptyset$ or the domain of a $\{\mathbf{0}, \mathbf{1}\}$-valued function $\mathbf{f}$ where the graph $\{(\mathbf{x}, \mathbf{f}(\mathbf{x})) : \mathbf{x} \in \mathbf{dom}(\mathbf{f})\}$ can be decided in polynomial time.

Summary of Lecture:
$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subset \mathbf{PRIMREC} \subset \mathbf{REC} \subset \mathbf{RE}$. It is an open problem whether the first two inclusions are proper.

Weeks 12 and 13

Two identical lectures on Wednesday 3 November and Friday 5 November 2021, both from 10:00 hrs to 12:00 hrs. Go to the one you prefer. Both use the same login for the Friday 5 November Lecture.

Alternative offered as staff and students should have the possiblity for rest on the NUS Well-Being Day. So those who prefer the usual time can go for the class as usual and those who want to take rest can go for the Wednesday class. Those who cannot make it for both can self-study and then use the office hour in either week 12 or week 13 for questions.

Tutorial in Week 13 for both Lectures 11 and 12. Those who want to do tutorial questions for last lecture in Week 13 can do so, but need to learn ahead in notes beforehand, as the corresponding Lecture is on Friday in Week 13.