Indian Association for the Cultivation of Science

(Deemed to be University under *de novo* Category)

Master's/Integrated Master's-PhD Program/ Integrated Bachelor's-Master's Program/PhD Course

Theory of Computation II: COM 5108

Lecture I

Instructor: Goutam Biswas *Autumn Semester 2025*

1 Introduction

What are the basic questions about the scope of symbolic computation. Can we define and compute all functions from the set of non-negative integers ($\mathbb{N}_0$) to itself ($[f : \mathbb{N}_0 \to \mathbb{N}_0]$)? Does every subset of $\mathbb{N}_0$ has a finite description? Following the idea of diagonalization of *Georg Cantor*, it was proved by *Alan Turing* and others[1] that it is impossible achieve that. Turing designed a model (mathematical) of computer known as *Turing machine* and proved that all functions are not computable. Starting from Cantor's idea of the hierarchy of infinite sets, it is possible to argue that the collection of *finitely describable* functions is countably infinite in size. But the collection of all possible functions on $\mathbb{N}_0$ is uncountable. So it is impossible to have finite definitions of all these functions in a formal system. There are functions *too complex* to describe and so to compute.

Similarly the collection of subsets of $\mathbb{N}_0$ is again *uncountably* infinite. And we cannot have formal and finite description of all of them. The testing of membership of each such subset, known as a *decision problem*, is also not possible.

There are functions that are *not computable* and there are sets whose membership is *not decidable*. They are the most complex problems that are *unsolvable*. This was established not only by Turing using his machine model but there are several other formalisms that are equivalent in computing power to the Turing machine. A few examples are λ-*calculus* and λ-*definable* functions, μ-*recursive* functions. There are other machine models such as *automata* with *2 push-down stack*, *RAM-machines*, rewriting systems such as *Post-systems*, etc. All these *formal systems* are *Turing complete* i.e. in terms of computing power they are equivalent to Turing machine. This is the well known *Church-Turing thesis*: it claims that any function definable by any of these formalisms can be computed by the Turing machine and vice versa.

There are studies about the *hierarchy* of undecidable problems. But for time being we postpone its discussion and concentrate on collection of *computable*

[1] Alonzo Church, Kurt Gödel and others.

functions or *tasks*. Consider a well defined *computable* task for which a Turing machine exists which always halts with the result. A more practical version of such recipe of computation is called an *algorithm*. There may be more than one *algorithm* (Turing machine) for a task. The next important question is how *useful* is the computation of an algorithm. Does it take too long time, does it take too much extra space, does it consume lots of energy (depends on time, space and the nature of algorithm). The absolute value of time and space is of lesser importance for a theoretical discussion[2]. What is the worst case requirement of resource for an algorithm of a task? How does the requirement of resource increases (for an algorithm) with the increase in the size of the task? Is there any minimum requirement of resource for a given task such that no algorithm can do better than that. Here is an example.

Example 1. Let $L = \{x \in \{0,1\}^* : \ x = x^R\}$, x^R is *reverse* of x, be a language over $\Sigma = \{0,1\}$. A *single-tape* Turing machine (A_1) can decide the membership of x in L ($x \stackrel{?}{\in} L$) in time roughly proportional to n^2 ($O(n^2)$), where $n = |x|$. Whereas a 2-tape Turing machine (A_2) can decide the membership of x in L in time roughly proportional to n ($O(n)$). [Note: we shall define big-O afterwards.] We assume same proportionality constant in both the cases (need not be true, but does not matter). If for $n = 10$ it takes 100 μs by A_1 and 10 μs by A_2. It will take $10^{12} = 11$ days for A_1 and 1 s for A_2 if $|x| = 10^6$.
Another important point to note is that the performance of A_2 in terms of time is the best what we can get for this problem.

In terms of space requirement, A_1 uses some constant amount ($O(1)$) of extra space (other than the input). But A_2 uses $O(n)$ extra space.

The example shows that the resource requirement is dependent on computing model. But there is a thesis which claims that if the time requirement for a problem on a reasonable model is t, then on a single-tape Turing machine it is some polynomial of t.

In case of computability or decidability there is a clear partition. Some problems are intrinsically computable (decidable) and some are not. If a language is computable (decidable) on a Turing machine, it remains computable (decidable) on any other equivalent model (*Church-Turing* Thesis).

Undecidability has a hierarchy. At the lowest level we have the *Turing decidable languages*, then *Turing recognizable* languages, and languages that are not even so.

Example 2. $L_{DFA} = \{< M, x >:$ the DFA M accepts the input $x\}$ is a Turing *decidable* language. But
$L_{TM} = \{< M, x >:$ the Turing machine M accepts the input $x\}$ is not Turing *decidable* but a Turing *recognizable* language. L_{TM} is the lowest-level undecidable language. The complement of L_{TM}, $\{< M, x >:$ M does not halt on $x\}$. is not even *Turing recognizable*.

In complexity theory an algorithm (a Turing machine or an equivalent recipe of computation that always terminates) is considered to be *time efficient* if its running time t is a polynomial function of the input size n. For a real computational an $O(n)$ algorithm is better than an $O(n \log n)$ algorithm and an $O(n \log n)$ algorithm is better than $O(n^2)$ etc.[3]. I do not know whether there

[2]Computer hardware may become faster and more memory space may be available at a low cost.

[3]Mathematically an $O(n)$ algorithm is also an $O(n \log n)$, $O(n^2)$ algorithm! We abuse the

is any algorithm of running time $O(n^{100})$ (but not $O(n^{99})$)[4]. If there is one, it may not be good for any real use. But this *abstraction* is useful. There are several reasons for that. Composition of two polynomials is a polynomial.

Example 3. Let $p(x) = x^2 + 2x + 3 \equiv O(n^2)$ and $q(x) = x^3 + 2 \equiv O(n^3)$. The composition $p(q(x)) = (x^3 + 2)^2 + 2(x^3 + 2) + 3 = x^6 + 4x^3 + 11 \equiv O(n^6)$.

If we apply a polynomial time bounded algorithm to the output of a polynomial time algorithm, the composite computation is also polynomial time bounded.

The second reason is, if the running time of a computing task on a particular computing model is bounded by a polynomial, then on any other reasonable model its running time is also bounded by a polynomial. There is a polynomial speed-up or polynomial slowdown as in (*Example 1*).

A large number of decision problems can be solved in polynomial time. They belong to a class known as **P**. But there are also a large number of decision problems for which there is no known polynomial algorithm. At the same time there is no *proof* that they cannot be solved in polynomial time. But often for many problems of the second category, if a *proof* or *certificate* of the existence of solution is supplied with the problem instant, the validity of the proof can be verified in polynomial time[5]. It is as if finding a proof is difficult, but the verification of the correctness can be checked efficiently (in polynomial time). This class of problems is called **NP**[6]. Clearly a problem in **P** is also in **NP**, as the polynomial time algorithm can produce a polynomial length certificate which can be verified in polynomial time. So $\mathbf{P} \subseteq \mathbf{NP}$, what is not known is whether the subset is proper.

Example 4. The decision problem of a language like
Reachable $= \{< G, s, d >:$ in the graph G the node d is reachable from node $s\}$ can be solved in linear time (over the size of the graph) by BFS or DFS algorithms. i So it belongs to the class **P**.
On the other hand there is no known polynomial time bounded algorithm for the decision problem of the language
CLIQUE: $\{< G, k >:$ the undirected graph G has a clique of size $k\}$. But if a set of k vertices of G is supplied as a certificate of solution, its correctness can be checked in polynomial time. So *CLIQUE* belongs to the class **NP**. However there is no known proof that it does not belong to class **P**.

Reduction function can be restricted to *polynomial time* to reduce problems within the class **NP**.

The interesting property of problems like *CLIQUE* is that all problems of **NP** is *polynomial time reducible* to it. The collection of such problems are known as **NP-complete** problems.

Till date the complexity theory fails to determine the absolute time complexity status of **NP**-complete problems. But they form an *equivalence class*. Following are three members of the class.

(a) $SAT = \{\phi :$ the propositional formula ϕ is satisfiable$\}$.

(b) *CLIQUE*$= \{< G, k >:$ undirected graph G with k-clique$\}$.

notation.

[4]May be we Sapient cannot conceive of such algorithm.

[5]Obviously the length of the proof is bounded by a polynomial.

[6]Originally the class was defined in terms of polynomial time bounded nondeterministic Turing machine as decider of such problems.

(c) *3-COLOR* $= \{G :$ undirected graph is 3-colourable $\}$.

Apparently very different looking problems are equivalent in terms of computational difficulty.

P is the class of well defined problems which can be solved *efficiently* (in polynomial time) and **NP** is the class of well defined problems for which the solution can be checked efficiently (in polynomial time). A *task* may be proving a theorem or the verification of the proof.

Is solving some problem is more difficult than checking its solution? Is proving a theorem more difficult than checking the correctness of the proof? That is the **P** *versus* **NP** question.

Answer to the question is not known. But there is a strong belief that the *inequality* is the fact. So the *intractability* of a problem B is demonstrated by reducing (polynomial time) a known **NP**-*hard* problem A to B.

Let A be a decision problem in **NP** and a is an instance of the problem. There is a polynomial time function $f : \{0,1\}^* \to \{0,1\}^*$ such that $a \in A$ if and only if $f(a) \in$ *3-COLOR*. Note that instances of both A and *3-COLOR* are *encoded* over the alphabet $\{0,1\}$. There is an *efficient computation* that translates the encoding of a to an undirected graph $f(a) = G$ so that a has the property of A ($a \in A$) if and only if $f(a) = G$ is 3-colourable.

Asymptotic Notations

In *computational complexity* we deal with functions from $\mathbb{N}_0 \to \mathbb{N}_0$ ($\mathbb{N}_0 = \{0,1,2,3,\cdots\}$). But we are more interested about the *asymptotic behavior* of the function. If $f, g : \mathbb{N}_0 \to \mathbb{N}_0$, then $f = O(g)$ (respectively, $f = \Omega(g)$), if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ (respectively, $f(n) \geq c \cdot g(n)$) for all but finitely many $n \in \mathbb{N}_0$. If $f = O(g)$ as well as $f = \Omega(g)$, then $f = \Theta(g)$.

Similarly, $f = o(g)$ (respectively, $f = \omega(g)$), if $f(n) < c \cdot g(n)$ (respectively $f > c \cdot g(n)$) for all $c > 0$ and all but finitely many $n \in \mathbb{N}_0$.

$f(n) =$ *poly*(n), if g is a polynomial and $f(n) \leq g(n)$ for all but finitely many $n \in \mathbb{N}_0$.

2 Models of Computation and Computational Tasks

In mathematics we do not talk much about the representation of objects. But computation is performed on representations. It transforms a representation of a problem instance to the representation of its solution. Finite list of numerical data is sorted, a positive integer is tested for primality, maximal clique is extracted from an undirected graph. Computation deals with representation of list, positive integer, graph etc.

All our objects are finite and encoded as a finite binary sequence, elements of $\{0,1\}^* = \bigcup_{n \in \mathbb{N}_0} \{0,1\}^n$, where $\mathbb{N}_0 = \{0,1,2,\cdots\}$.

If $x \in \{0,1\}^*$ and its length $|x| = m$, then $x \in \{0,1\}^m$. If $x = x_1 x_2 \cdots x_n$, the i^{th} bit of x is x_i. If $x, y \in \{0,1\}^*$, then we can concatenate them to get $xy = x_1 x_2 \cdots x_m y_1 y_2 \cdots y_n \in \{0,1\}^{m+n}$, where $|x| = m$ and $|y| = n$.

It may be necessary to deal with ordered n-tuples $(w_1, w_2, \cdots, w_n)$, where $w_i \in \{0,1\}^*$, $i = 1, 2, \cdots, n$. This can be be encoded as a single string over $\{0,1\}*$. We denote the encoding as $< w_1, \cdots, w_n >$.

Example 5. Let $s = (0101, 100, 1)$. We encode '0' as '01', '1' as '10', and 'comma' as '11'. The encoded string of s is '01100110111001011110'.

Under the encoding scheme every element of $\{0,1\}^*$ may not represent a valid object. Proper encodings form a proper subset of $\{0,1\}^*$. In practical computing this can be easily detected by *parsing*. But we may ignore the issue for theoretical discussion and associate the invalid elements of $\{0,1\}^*$ to some fixed object.

2.1 Computational Problems

We are familiar with different types of computation problems e.g. *search* problems, *decision* problems, *optimization* problems etc.

Example 6. Given an undirected graph G, search for a clique in it of size ≥ 5. There may be $0, 1$ or many such cliques in it. One needs to find one if there is any. This is an example of a search problem.

Given an undirected graph G and a positive integer k. One needs to find whether there is a clique of size k in G i.e. $< G, k > \stackrel{?}{\in}$ *CLIQUE*. This is a decision problem.

Given an undirected graph G find the *vertex cover* of minimal size. This is an optimization problem.

Another important computational task is to find the total number of solutions. It is a *counting problems*.

Example 7. Given a propositional Boolean formula we want to count the number of possible satisfying assignments.
Given an undirected graph G, count the number of cliques of sizes $\geq k$.

We define these problems more precisely.
A *search problem* after encoding may be viewed as a relation S over $\{0,1\}^*$ i.e. $S \subseteq \{0,1\}^* \times \{0,1\}^*$. For a problem instance $x \in \{0,1\}^*$, $S(x) = \{y : (x,y) \in S\}$. The set $S(x)$ may be empty.
As an example, if x is a Boolean formula, then $S(x)$ is the set of satisfying assignments.
A function $f_S : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ solves the search problem if

$$f_S(x) = \begin{cases} y \in S(x) & \text{if } S(x) \neq \emptyset, \\ \perp & \text{if } S(x) = \emptyset. \end{cases}$$

If x is a Boolean formula and it is *satisfiable*, then $f(x)$ is a satisfying assignment of x.

A *decision problem* is the encoded set $D \subseteq \{0,1\}^*$. The subset D may be viewed as *language*. The solution of a decision problem D computes a function $\mu_D : \{0,1\}^* \to \{0,1\}$, the *characteristic function* of D,

$$\mu_D(x) = \begin{cases} 1 & \text{if } x \in D, \\ 0 & \text{if } x \notin D. \end{cases}$$

An *optimization problem* makes sense if weights ($w : \{0,1\}^* \to \mathbb{R}$) are associated with different solutions. A function $f_O : \{0,1\}^* \to \{0,1\}^*$ solves an optimization problem, if

$$f_O(x) = \begin{cases} y \in S(x) & \text{if } w(y) = opt\{w(z) : z \in S(x)\}, \\ \perp & \text{if } S(x) = \emptyset. \end{cases}$$

A counting problem essentially finds the size of $S(x)$. It computes the function $f_C : \{0,1\}^* \to \mathbb{N}_0$, where $f_C(x) = |S(x)|$.

2.2 Algorithms: Uniform Models of Computation

The model of computing or algorithm is *uniform* in the sense that it can work on different instances (sizes) of a problem. It has the following components.

- A representation of the input.
- A finite set of simple operations.
- A finite sequence of these operations.
- An work space.
- At every step the operation acts locally. But the locality changes.
- At the end it stops and there is the representation of the output.

An algorithm A computes a function $f_A : \{0,1\}^* \to \{0,1\}^*$. If $f_A(x) = y$, the algorithm A presented with the input x, computes y and halts.
If it is a search problem then f_A is same as f_S (note the change in codomain). For a decision problem f_A is μ_D etc.

A concrete uniform model is the Turing machine (TM) invented before any modern computer was built. The main components of a Turing machine are the following.

- An infinite (potentially) sequence of cells, starting from the leftmost cell. Each one can hold a symbol of representation.
- A finite set of tape symbols (Σ).
- Current location pointer (h: head) of the sequence.
- Finite set of states (Q) of the machine. This includes an *initial* and *final/halt* states.
- A function δ is called a transition function from $Q \times \Sigma$ to $Q \times \Sigma \times \{-1, 0, +1\}$, where $\{-1, 0, +1\}$ change of current positions to *left*, *no move* and *right*.

There is a nondeterministic version of the model. Where there are two transition functions δ_0 and δ_1[7]. At every state the NTM (non-deterministic Turing machine) will non-deterministically choose one of the transition functions. So the possible sequence of configurations forms a binary tree with the *start configuration* at the root. If any of the computation sequence accepts the input, machine accepts the input.
An NTM can be simulated by a DTM (deterministic Turing machine). The DTM simulates all possible 1-step computations, all possible 2-step computations, $\cdots$. The simulation takes an exponential amount of time compared to computation time of the NTM.

The Church-Turing Thesis

The thesis asserts that whatever can be computed by a reasonable model of computing can also be computed by a Turing machine and vice versa.

[7] Any finite degree of non-determinism can be transformed to a binary non-determinism.

2.3 Universal Algorithm

There is a universal TM U that on input $(< M >, x)$, where $< M >$ a description of a Turing machine (finite) and x an input to M, halts with the output $M(x)$, if M halts on x, otherwise does not halt. That is $U(< M >, x) = M(x)$.
Given the description of of M and a configuration of M on the input x, U can perform a single step of M and generate the next configuration of M. The existence of a Universal machine makes the TM model *general purpose* or *uniform*. There is no basic difference between an algorithm $(< M >)$ and its input data (x). It depends on the interpretation.

2.4 Undecidable Sets

We have already tried to justify from the *cardinality* consideration, that there are *uncomputable* functions. Following is a more concrete demonstration of the claim.

A language $L \subseteq \{0,1\}^*$ is *decidable* or *recursive* if there is a Turing machine M_L that halts on all input (elements of $\{0,1\}^*$) and

$$M_L(x) = \begin{cases} 1 & \text{if } x \in L, \\ 0 & \text{if } x \notin L. \end{cases}$$

A language L is *recursively enumerable (r.e.)* or *Turing recognizable* if there is a Turing machine M_L that behaves as follows.

$$M_L(x) = \begin{cases} \text{1machine halts} & \text{if } x \in L, \\ \text{0 if } M \text{ halts, else does not halt} & \text{if } x \notin L. \end{cases}$$

Equivalently we may say that

$$M_L(x) = \begin{cases} \text{halts} & \text{if } x \in L, \\ \text{does not halt} & \text{if } x \notin L. \end{cases}$$

A Turing machine that halts with the output 0 can be modified to run forever. If a language is not *decidable* then it is called *undecidable*. Clearly a *decidable* language is also a *recursively enumerable* language. If a language L and its complement $\overline{L}(= \Sigma^* \setminus L)$ are both *recursively enumerable*, then it is *decidable*.

We show that there is a languages that is not *decidable*. In terms of functions, a Turing machine computes *partial functions* on $\mathbb{N}_0$ where a function may not be defined for all $n \in \mathbb{N}_0$.

Consider the language

$$L_{TM} = \{< M, x >: \ M \text{ is a Turing machine and } x \in L(M)\}.$$

We show that this language is *recursive enumerable* but not *recursive* or *decidable*. It is *undecidable*.
The proof is by contradiction. Suppose the language is decidable and is decided by the Turing machine M_{TM}. The behavior of M_{TM} is as follows:

$$M_{TM}(< M, x >) = \begin{cases} 1 & \text{if } M \text{ accepts } x, \\ 0 & \text{if } M \text{ does not accept } x. \end{cases}$$

We design another Turing machine D that uses M_{TM} as an *oracle* or a function call. D is defined as follows.

D : input $< M >$

1. Call M_{TM} with input $(< M, M >)$.

2. Output $(1 - M_{TM}(< M, M >))$ (flip the output of M_{TM}).

Clearly

$$D(< M >) = \begin{cases} 1 & \text{if } M_{TM}(< M, M >) = 0,\ M \text{ accepts its own description,} \\ 0 & \text{if } M_{TM}(< M, M >) = 1,\ M \text{ does not accept} \\ & \text{its own description.} \end{cases}$$

But then if run D on its own description $< D >$, then

$$D(< D >) = \begin{cases} 1 & \text{if } M_{TM}(< D, D >) = 0, \\ 0 & \text{if } M_{TM}(< D, D >) = 1. \end{cases}$$

This is a contradiction. So no such M_{TM} can exist.
It is easy to see that L_{TM} is a recursively *enumerable language*. Consider the following machine.

$$N_{TM}(< M, x >) = \begin{cases} 1 & \text{if } M(x) = 1,\ M \text{ accepts } x, \\ 0 & \text{if } M(x) = 0, \\ \textit{does not halt} & \text{if } M \text{ does not halt on } x. \end{cases}$$

We may view the outcome of application of a TM M_i on the code of a TM $< M_j >$ as a table. The $[i, j]^{th}$ entry is one (1) if M_i halts on the input $< M_j >$. Otherwise it is zero (0).

	$< M_1 >$	$< M_2 >$	$< M_3 >$	$\cdots$
M_1	0	1	1	$\cdots$
M_2	1	1	0	$\cdots$
M_3	1	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

The TM D flips the outcome of the diagonal elements. The contradiction comes for the *diagonal value* corresponding to D.

The complement of L_{TM} is not even *recursively enumerable*. The reason is simple. If both L_{TM} and $\overline{L_{TM}}$ are *recursively enumerable* then they are *recursive* or *decidable*.

2.5 Reduction

There is notion of *reducibility* of one language to another. It helps to characterize or solve a new problem.

Consider the language

$$L_H = \{< M, x >:\ M \text{ halts on } x\}.$$

We prove that this language is also *undecidable*. But the proof is through a reduction of L_{TM} to L_H. If there is a Turing machine M_H that decides L_H, then we can use it to decide L_{TM}. Our design of M_{TM}, a decider for L_{TM} using the decider for M_H is as follows:

M_{TM}: input $< M, x >$

1. Run M_H on $< M, x >$,
2. If $M_H(< M, x >) = 0$, output 0 and halt,
3. Simulate M on x until it halts (it will!),
4. If $M(x) = 1$, output 1, else, output 0.

Essentially if we know the answer to $< M, x > \overset{?}{\in} L_H$, we can decide L_{TM}. Note in this connection that the complement of L_H, $\overline{L_H}$ is not *recursively enumerable*.

2.5.1 Mapping Reducibility

A language $A \subseteq \Sigma^*$ is *mapping reducible* to a language $B \subseteq \Sigma^*$, if there is *computable function* $f : \Sigma^* \to \Sigma^*$ so that $x \in A$ if and only if $f(x) \in B$. We write $A \leq_m B$, "A is mapping reducible to B".
Mapping reducibility is very useful for *decision problems*.

It is clear that '$\leq_m$' is a binary relation on the set of languages. The relation is *reflexive* and *transitive*: for all language A, $A \leq_m A$ (by identity mapping), and if $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$ (by function composition).

The reduction function f transforms the membership question of A to B. Problem reduction shows a relation of complexity among problems. If $A \leq_m B$, then A cannot be more difficult than B or B cannot be simpler than A. Following are examples of conclusions one can draw from a *reduction*

Let $A \leq_m B$ equivalently $\overline{A} \leq_m \overline{B}$.

(a) If B is *decidable* then so is A.

(b) If B is *Turing recognizable* then so in A.

(c) If A is *not Turing decidable (recognizable)* then B is also *not Turing decidable (recognizable)*.

Example 8. Here is another example of *Reduction* to show that

$$Reg_{TM} = \{< M >: \ M \text{ is a Turing machine and } L(M) \text{ is a regular language}\},$$

is undecidable. The proof is by contradiction. We reduce L_H (we may as well use L_{TM}) to Reg_{TM}.
Suppose there is a decider R_M for Reg_{TM}. We use it to construct a decider of L_H as follows:
M_H: input $< M, x >$

1. Construct the following TM M_1.
   M_1: input y
   (a) If $y = y^R \in \{0, 1\}^*$, *accept*.
   (b) Otherwise, simulate M on x.
   If M halts on x, *accept*(y).
2. Simulate R_M with $< M_1 >$ as the input.
3. If R_M accepts then *accept*, else *reject*.

The language of M_1 is interesting.

$$L(M_1) = \begin{cases} \{0,1\}^*, & M \text{ halts on } x, \\ \{y \in \{0,1\}^* : \ y = y^R\}, & M \text{ does not halt on } x. \end{cases}$$

So $< M_1 > \in Reg_{TM}$ if and only if M halts on x.
$M_H(< M, x >)$ *accepts* if and only if Reg_{TM} *accepts* if and only if M halts on x.
So M_H is a decider of L_H - a contradiction.
Hence Reg_{TM} is undecidable. In this case the mapping is from an instance of L_H to an instance of Reg_{TM}, $f(< M, x >) = M_1$.

It is known that $\overline{L_H}$ is *not recursively enumerable*. If we can reduce $\overline{L_H}$ to some set A, then A is also *not recursively enumerable*.

Example 9. [DCK1]
Consider the languages $L_{fin} = \{< M >: \ L(M) \text{ is finite}\}$ and its complement $\overline{L_{fin}}$. We reduce $\overline{L_H}$ to both of them to show that none of them are Turing recognizable (recursively enumerable).
$\overline{L_H} \leq_m L_{fin}$:
Given $< M, x >$ the mapping function construct the following TM.
M_1 : input y

1. Simulate M on x.
2. If M halts on x, *accept*.

$$L(M_1) = \begin{cases} \{0,1\}^*, & M \text{ halts on } x, \\ \emptyset & M \text{ does not halt on } x. \end{cases}$$

$L(M_1)$ is finite if and only if M does not halt on x i.e. $< M_1 > \in L_{fin}$ if and only if $< M, x > \in \overline{L_H}$.
So L_{fin} is not Turing recognizable.

$\overline{L_H} \leq_m \overline{L_{fin}}$:
It is equivalent to say $L_H \leq_m L_{fin}$. We design the following TM M_2 such that $< M, x > \in L_H$ if and only if $< M_2 > \in L_{fin}$ i.e. M halts on x if and only if $L(M_2)$ is finite.
M_2 : input y

1. Simulate M on x for $|y|$ steps.
2. If M does not halt within $|y|$ steps, *accept*, otherwise *reject*.

There are two possible cases now:

(a) If M does not halt on x, then for no y, it will halt within $|y|$ steps and the y is accepted. The language of $L(M_2) = \Sigma^*$.

(b) Let M halts on x in k steps. Then for any y, $|y| < k$, the simulation will not halt and the y will be accepted. But for all y, $|y| \geq k$, the simulation of M will halt, and y will be rejected.
The number of y, $|y| < k$ is finite, and so is the language of $L(M_2)$.

The final conclusion is both L_{fin} o its complement are not Turing recognizable languages.

2.5.2 Rice's Theorem

Many undecidable problems e.g. Reg_{TM}, L_{fin} are special cases of a general property of a TM. Consider the collection *recursively enumerable (Turing recognizable)* languages $RE \subseteq 2^{\Sigma^*}$ over $\{0,1\}^*$. Each of these languages has a *finite description* (partial) using the corresponding TM. If L is recognized by the TM M, then $< M >$ is a *description*[8] of L. The encoding of this collection of TM codes form a Language $L_{re} = \{< M >: L(M) \in RE\}$.
A *property* P of the set RE is called *trivial* if no languages in RE have the property or all languages of RE have the the property. Otherwise the property is *nontrivial*. Any *nontrivial* property of RE languages gives a *proper* and *nonempty* subset L_P of L_{re} ($L_P \subsetneq L_{re}$).
As an example the collection of regular languages, a proper subset of RE, is a nontrivial property. The corresponding subset of L_{re} is $\{< M >:\ L(M)$ is regular$\}$.
Theorem 1. (*Rice's Theorem*) Any subset of L_{re} corresponding to a nontrivial property P is undecidable. In other words any nontrivial property of RE languages is undecidable.
Proof: Suppose the empty language ($\emptyset$) does not have the property P (afterward we shall see how to handle the situation if $\emptyset$ has the property)
Let $L \neq \emptyset$ in RE has the property P and M_L is a TM that *recognizes* L. Given a pair $< M, w >$, where M is any TM and w its input, the following TM can be *effectively* constructed. It may be viewed as a mapping, $< M, x > \mapsto < M_1 >$.

M_1: input x:

(i) Emulates M on w.

(ii) If M halts, emulate M_L on x.

(iii) If M_L accept (halts with 1), *accept (halt with 1).*

Clearly

$$L(M_1) = \begin{cases} L & \text{if } M \text{ halts on } w, \\ \emptyset & \text{otherwise.} \end{cases}$$

$L(M_1) = L$ i.e. $< M_1 > \in L_P$ *if and only if* M halts on w ($\emptyset \notin L_P$ - assumption).
Suppose L_P is *decidable* and D_P is a decider of L_P. We construct the following TM to decide L_H, the *halting* problem.

D_h: input $< M, w >$:

(i) Condtruct the TM M_1 with the input $< M, w >$ (shown above).

(ii) Emulate D_P on $< M_1 >$.

(iii) If D_P halts with 1, **halt** with 1.

(iv) If D_P halts with 0, **halt** with 0.

[8]The description is *partial* because if $x \notin L$, the corresponding machine may not report that.

D_P halts with 1 *if and only if* $L(M_1) \in L_P$ *if and only if* M halts on w. So D_h decides L_H.
But that is a *contradiction* as we already know that *halting problem* is undecidable. So there is no D_P for L_P.

We started with the assumption that $\emptyset$ does not have the property P. If $\emptyset$ has the property P. We consider $\neg P$ and $L_{\neg P}$, the collection of codes of TM whose languages does not have the property P. But then if $L_{\neg P}$ is decidable, then so is L_P (complement language), and the proof follows. QED.

Note the following:

1. There is a computable function f that maps $< M, x > \mapsto < M_1 >$ such that $< M, x > \in L_H$ if and only if $L(M_1) \in L_P$.

2. In the construction of D_h, the TM D_P (if it exists) is used as an *oracle*.

2.6 Time Complexity

Undecidable problems, non-computable functions are objects beyond the scope of symbolic computing. In complexity theory, we are more interested about resource bounded (e.g. time) computable problems. There are two measures of time related to a problem. One is the *time complexity* of an *algorithm* of a problem and the other is the inherent *time complexity* of a *problem*. The worst case time complexity of *insertion sort* is $O(n^2)$. On the other hand sorting by comparison and exchange algorithm cannot have worst case time complexity better than $n \log n$[9].

We consider Turing machines that halt on all input (an algorithm). The number of *basic steps* taken by such a machine A on each input is a function $f_A : \{0,1\}^* \to \mathbb{N}$. This is the *time complexity* of the algorithm.

Different input of same length may take different amount of time and often we are interested about a function $T_A : \mathbb{N} \to \mathbb{N}$, where $T_A(n) = \max_{x \in \{0,1\}^n}\{f_A(x)\}$. This is called the worst case time complexity of the algorithm A.

The complexity theory if more concerned about the time complexity of a problem Π, the *upper bound* and *lower bound* of possible algorithms to solve Π. By the *upper bound* we roughly mean the time complexity of the fastest possible algorithm[10]. The lower bound tells that any algorithm of Π requires at least that much of time in worst case[11].
The complexity of a problem may depend on the specific model of computation where the algorithm is implemented.

Cobham-Edmonds Thesis

If the *time complexities* of a problem Π in any two reasonable models of computation are T_1 and T_2, then they are related by a polynomial i.e. $T_2 = poly(T_1)$.

Example 10. The decision problem of $\{w \in \{0,1\}^* : \ w = w^R\}$ on a 2-tape TM can be solved in linear time. But it takes quadratic time on a single-tape TM.

[9] There are $n!$ permutations of n distinct objects. Every stage of comparison and decision to exchange a pair of object can be viewed as a binary decision tree. The minimum height of a balanced binary tree with $n!$ leaf nodes is $\log_2(n!) \approx n \log_2 n$. In the worst case an algorithm is required to make $\Omega(n \log_2 n)$ comparisons.

[10] The notion of *fastest* algorithm may not be justified etc.

[11] As an example $\Omega(n \log n)$ in case of sorting by comparison and exchange.

Any problem Π that has time complexity $O(n)$ on a multi-tape TM can be solved in time $O(n^2)$ on a single-tape TM.
The thesis says that, a problem Π has time complexity t in some reasonable model of computing *if and only if* its time complexity is *poly*(t) on a single-tape Turing machine.

2.7 Efficient Algorithms

Algorithms that has polynomial time upper bound ($t = \text{poly}(n)$) are considered to be efficient due to following reasons.

(a) The upper bound cannot be less than $O(n)$ as the input is to be read.

(b) Polynomials are moderately growing[12] functions.

(c) Polynomials are closed under composition. So the composition of two polynomial time bounded algorithms is also bounded by a polynomial.

(d) Polynomial time bounded algorithm remains to be polynomial time bounded on different models.

(e) Exponential algorithms of exhaustive search will always overshoot in computation time of any polynomial time algorithm on large data.

Time Bounded Universal Turing Machine

A *time-bounded universal TM* U takes $(< M >, x, t)$ as input. It simulates a M on the input x for t steps. If M halts within t steps with the output y, then U halts with output y. If M does not halt within t steps, then also U halts with a special output $\perp$. It computes a total function.
For every step of simulation, a time bounded U not only reads the transition table of M and updates the configuration of M, but also keeps track of the time t.

Space Complexity and Multi-Tape

The amount of memory usage during computation is another measure of resource. Here we shall not consider the space required to store the input or the output. This model of TM has three tapes, (i) *read-only* input tape, (ii) *write-only* output tape, and (iii) *read-write* work tape.
The space complexity of such TM is defined as the number of cells of the work tape as a function of input length: $S_M(n) = \max_{x \in \{0,1\}} s_M(x)$, where $s_M(x)$ is the number of cells used in the work tape.

Oracle Turing Machine

We have already come across to Turing machines that invoke some other Turing machine during its computation. As an example the TM D_h (for the halting problem) invokes D_P, the decider for property P. Such a machine sends its *queries* ($x \in \{0,1\}^*$) and gets replies $f(x) \in \{0,1\}^*$, where $f : \{0,1\}^* \to \{0,1\}^*$ is the function computed by the invoked machine.

An *Oracle Turing Machine* has an additional tape for writing down the query (x) and getting the reply ($f(x)$). It also has two new states, corresponding to oracle-call and oracle-return.

The output of a Turing machine M with an oracle function f, on input x is written as $M^f(x)$.

[12] Algorithm with very high degree polynomial upper-bound has little practical use. But for theoretical analysis unbounded degree is better than large finiteness.

2.8 Non-Uniform Models of Computation

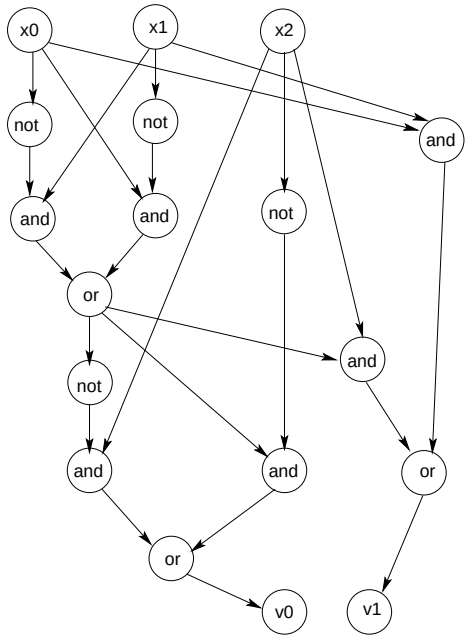
A non-uniform model uses different combination of devices for computation on input of different sizes (lengths). The size of the collection of devices for an *infinite language* is infinite. Though each device is of finite in size, a collection of devices for some language may not have any *finite description*.

2.8.1 Boolean Circuits

Computers are made of *Boolean Circuits* and algorithms run on them. Computation of a TM can be simulated on a Boolean circuits. It may be a better model to understand the *limitation* of *efficient* computation.

A *Boolean Circuit* is a *directed acyclic graph* with three types of *labeled vertices*: *source*, *sink* and *internal*.

Example 11. Following is an example of a Boolean circuit.



(a) *Source vertices:* These vertices do not have any incoming edge. They are labeled with the input variables[13]. We may use indices of the variables (natural numbers) to label them (i stands for x_i). More than one source vertices may have the same label.

(b) *Sink vertices:* These are the output terminals with one incoming edge and no outgoing edge, labeled by the output variables or their indices. But unlike the *source* vertex, no two *sink* vertices have the same label.

(c) *Internal vertices:* They corresponds to *Boolean gates* of the circuit and are typically labeled with $(\neg, \wedge, \vee)$ corresponding to *(not, and, or)* gates. *Out-degree* of these gates are one or more (*fanout*). The in-degree of '$\neg$' is one, and for other two or more[14].

[13]It may be necessary to put a constant *true* or *false* as a input. Otherwise there will be problem of null string as an input to a circuit!

[14]Often, 2 input and 1 output edges for *and, or* gates.

The *fan-in* of each internal node may be bounded to at most two (bounded fan-in). The set of logic gates should be functionally complete for Boolean functions.

A restricted version of *Boolean circuit*, called a *Boolean formula*. It has one out-degree for internal nodes. This makes the DAG a tree.

Two restricted versions of Boolean formula and the corresponding circuits are *conjunctive normal form (CNF)* and *disjunctive normal form (DNF)*.
CNF formula is a *conjunction (and)* of *clauses*. A *clause* is a *disjunction (or)* of *literals*. A *literal* is a *variable* or the *negation* of a *variable*.
CNF forms a layered circuit, *negations, disjunctions* and *conjunction*.
Every Boolean formula can be converted into an equivalent CNF formula. But the equivalent formula may be exponentially larger in size than the original formula.
If there are at most *k-literals* in every clauses of a CNF, It is called k-CNF.

A *DNF* formula is *disjunction* of *conjunction* of literals. If there are at most *k-literals* in every conjunction, it is called a k-DNF.

Example 12. Consider the following *truth-table* of two Boolean functions

a	b	c	f	g
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

For the construction of the DNF we look at the output 1 of the rows.
$f(a,b,c) = \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}\overline{c} + abc$. It is enough for one conjunct to be true.
For the construction of CNF we look at the output 0 of the rows.
$f(a,b,c) = (a+b+c)(a+\overline{b}+\overline{c})(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+c)$. But in this case everu clause is to be true.

A *Boolean circuit* with n different input labels and m output terminals computes a function $f : \{0,1\}^n \to \{0,1\}^m$ as follows.
If the values of the children of a gate g of *in-degree* d are $v_1, \cdots, v_d$, then the value of g,

$$v_g = \begin{cases} \wedge_{i=1}^{d} v_i & \text{if } g \text{ is an } \textit{and} \text{ gate,} \\ \vee_{i=1}^{d} v_i & \text{if } g \text{ is a } \textit{or} \text{ gate,} \\ \neg v_1 & \text{if } g \text{ is a } \textit{not} \text{ gate and } d = 1. \end{cases}$$

Definition 1. A *circuit family* $\mathcal{C} = \{C_n\}_{n \in \mathbb{N}_0}$, where C_n has n input.
A language L is decided by a *circuit family* $\mathcal{C}$ where each C_n has n input and one output. It computes a function from $\{0,1\}^n \to \{0,1\}$ such that for all $x \in \{0,1\}^n$, $C_n(x) = 1$ *if and only if* $x \in L$. In general $C_{|x|}(x) = 1$ *if and only if* $x \in L$.

Circuit Size Complexity

The size of a circuit is its number of *edges* (or the number of internal gates if the *fan-in* and *fan-out* are bounded).
The size of a circuit family $\{C_n\}$ is a function $s : \mathbb{N}_0 \to \mathbb{N}_0$, where $s(n) = |C_n|$.

A language $L \in \mathbf{SIZE}(s(n))$ if there is a circuit family of size $s(n)$ such that

$$x \in L \text{ if and only if } C_{|x|}(x) = 1.$$

The circuit complexity of a language L is $s_L : \mathbb{N}_0 \to \mathbb{N}_0$, where $s_L(n)$ is the size of the smallest circuit C_n such that for all $x \in \{0,1\}^n$, $C_n(x) = 1$ *if and only if* $x \in L$.

The class $\mathbf{P/poly}$ is the collection of languages decided by polynomial size circuit families i.e. $\mathbf{P/poly} = \bigcup_{k\in\mathbb{N}} \mathbf{SIZE}(n^k)$.

Following are a few facts about circuit complexity.

(a) Every Boolean function $f_n : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size $O(n2^n)$ (*truth table*). If there are n variables, the truth table has 2^n rows. Say in CNF there are at most 2^n clauses and each clause has n literals.

(b) Every subset A of $\{0,1\}^*$ gives rise to a function $f_A : \{0,1\}^* \to \{0,1\}$. Restriction of f_A for each $n \in \mathbb{N}$ gives rise to a n variable Boolean function and a circuit family.
There are *uncountably* many subsets of $\{0,1\}^*$. So there are uncountably many circuit families. Most of them cannot have finite representation and undecidable languages also have circuit families.

(c) Any function that has a time complexity t has a circuit complexity *poly*(t). Note that there are $O(t(n))$ configurations of the Turing machine that computes the function. Each configuration is of length $t(n)+k$, where k is a constant. The transitions from one configuration to the next configuration can be emuleted by circuit of size $O(t(n))$.

(d) If the time complexity of a language $t(n) =$ *poly*(n), then its circuit complexity is also *poly*(n).

(e) Most of the Boolean functions have exponential circuit complexity. The size of the domain of the n variable Boolean function is 2^n. So there are 2^{2^n} boolean functions.
On the other hand a circuit of size s may be viewed as a *directed acyclic graph (DAG)*. It consists of n input vertices (0-indegree), internal vertices of 'and', 'or' or 'not' gates (at most 2-indegree and 1-outdegree) and one output vertex (1-indegree).
The description of each input node and internal vertices can be written as a binary string of length $\log(s+n)$. The sources of two incoming input to a gate can be encoded as a binary string of length $2\log(s+n)$. The description for s internal nodes is of length $2s\log(s+n)$. Taking $s > O(n)$, the description length is $O(s\log s) = O(s^2)$. With this length of description we can describe at most $2^{O(s^2)}$ different circuits and corresponding functions. So to compute 2^{2^n} Boolean functions $s^2 \approx 2^n$ i.e. $s = 2^{O(n)}$. If $s = n^k$, $k \in \mathbb{N}$, the number of circuits are $2^{n^k} \ll 2^{2^n}$.

2.9 Probabilistic Turing Machine

There are simpler and efficient algorithms for different problems that use *randomness (pseudo randomness)*. Such an algorithm has a probability of error

that can be reduced to a very small value. Testing whether a positive integer is *prime* is a typical example. The *Miller-Rabin* test on input n takes $O((\log n)^3)$ bit operations with error probability bounded by $\frac{1}{4}$. It can be lowered to $\frac{1}{4^k}$ by k repeations. This kind of algorithms are known as *Monte Carlo algorithm*[15]. What is the power of a Turing machine that can toss a coin? The model of a time bounded probabilistic Turing machine (PTM) is as follows:
The Turing machine M has two transition functions, δ_0 and δ_1. On an input x, at every stage of computation, it chooses either δ_0 or δ_1 with a probability $\frac{1}{2}$. The choices is independent of previous choices (uniformly random). Irrespective of the random choices, it halts within $T(|x|)$ steps with the outcome *accept (1)* or *reject (0)*. The output of the TM is a random variable $M(x)$.

This computation has 2^t branches ($t = T(|x|)$), each one is taken with a probability $\frac{1}{2^t}$. So the probability of the success, $Pr[M(x) = 1]$, is the fraction of branches that ends with output 1.
The model and its computation looks very similar to a time bounded nondeterministic Turing machine.
But the interpretation of the outcome is different. In an NTM, input x is accepted if there is one branch with output 1. But in case of PTM input x is accepted if $Pr[M(x) = 1] > \frac{1}{2}$.

2.10 Time and Space Bounded Classes

We are interested about Turing machines that halts on all input. In such a machine the computation time or work-space used can be expressed as a function of the size of input. We already have introduced the basic concepts of time and space complexity. A complexity class is a collection of problems that can be solved within the time (or space) bounded by some function of the input size.

Definition 2. A k-tape deterministic Turing machine M halts on every input $x \in \{0,1\}^*$. It takes at most $T(n)$ steps for all but finitely many inputs of length n, where $T : \mathbb{N}_0 \to \mathbb{N}_0$. The machine M is called *$T(n)$-time bounded*.

A nondeterministic Turing machine is also time bounded in a similar way. But in this case computations on all possible paths, due to nondeterministic choice, must come to halt within $T(n)$ steps. The acceptance condition is different. If there is computation-path that comes to *accept halt (1)*, the input is accepted (decision problem).

As a computer of a function a deterministic machine M starts with $x \in \{0,1\}^*$ on the input tape. When it halts, $f(x)$ is on the output tape. The TM computes the function $f : \{0,1\}^* \to \{0,1\}^*$ in $T(|x|)$ time.
If M is a *decider*, then $x \in L(M)$ is decided in $T(|x|)$ time.

Definition 3. A k-tape deterministic Turing machine M halts on every input $x \in \Sigma^*$, and uses at most $s(n)$ cells on the work-tapes, for all but finitely many inputs of length n, then M is called *$s(n)$-space bounded*.

A nondeterministic Turing machine is also space bounded in a similar way. But in this case computations on each of its possible paths should not use more than $s(n)$ cells of work-tapes. If M is a decider, then $L(M)$ is decided in $s(n)$ space.

It is necessary that the space and time binding functions should be *non-decreasing* and *constructible*.

[15]There is another type known as *Las Vegas algorithm* that always produces a correct answer. But randomness affects the runtime. It may not even terminate.

Definition 4. A function $T : \mathbb{N}_0 \to \mathbb{N}_0$ is said to be *time constructible* if $T(n) \geq n$ and there is k-tape Turing machine that for any x of length n, computes the binary representation of $T(x)$ in time $O(T(n))$.

A function $s : \mathbb{N}_0 \to \mathbb{N}_0$, where $s(n) \geq O(\log n)$, is called *space constructible*, if there is a Turing machine that takes x of length n as input and computes $s(n)$ using $O(s(n))$ work-tape cells.

The class of *constructible* or *proper* functions includes all reasonable functions such as $n, n \log n, n^k, 2^n$ etc. It can be proved that if functions f and g are constructible, then $g \circ f, f + g, f \times g$, 2^f and many more are constructible.

Let $t : \mathbb{N}_0 \to \mathbb{N}_0$ and $t : \mathbb{N}_0 \to \mathbb{N}_0$ be proper functions. We define the following complexity classes. Our model is a k tape Turing machine.

Definition 5.

(a) $DTIME(t(n)) = \{L$: there is a $t(n)$ time bounded deterministic Turing machine (DTM) M so that $L = L(M)\}$.

(b) $NTIME(t(n)) = \{L$: there is a $t(n)$ time bounded nondeterministic Turing machine (NTM) M so that $L = L(M)\}$.

(c) $DSPACE(s(n)) = \{L$: there is a $s(n)$ space bounded DTM M so that $L = L(M)\}$.

(d) $NSPACE(s(n)) = \{L$: there is a $s(n)$ space bounded NTM M so that $L = L(M)\}$.

(e) $\mathbf{L} = DSPACE(\log n)$.

(f) $\mathbf{NL} = NSPACE(\log n)$.

(g) $\mathbf{P} = \cup_{i \geq 1} DTIME(n^i)$.

(h) $\mathbf{NP} = \cup_{i \geq 1} NTIME(n^i)$.

(i) $\mathbf{PSPACE} = \cup_{i \geq 1} DSPACE(n^i)$.

(j) $\mathbf{NPSPACE} = \cup_{i \geq 1} NSPACE(n^i)$.

(k) $\mathbf{EXP}(\mathbf{EXPTIME}) = \cup_{i \geq 1} DTIME(2^{n^i})$.

(l) $\mathbf{NEXP}(\mathbf{NEXPTIME}) = \cup_{i \geq 1} NTIME(2^{n^i})$.

(m) $\mathbf{EXPSPACE} = \cup_{i \geq 1} DSPACE(2^{n^i})$.

(n) $\mathbf{NEXPSPACE} = \cup_{i \geq 1} NSPACE(2^{n^i})$.

Some inclusion relations among the classes are straight forward.

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{NPSPACE} \subseteq \mathbf{EXP}.$$

A DTM is an NDTM with the degree of non-determinism 1. An NDTM can be simulated using a DTM following the *depth first search* of the NDTM's computation tree. As the depth is bounded by $f(n)$, the space is also bounded by $f(n)$.

We have the following claim about the last inclusion:

$$NSPACE(f(n)) \subseteq DTIME(2^{O(f(n))})$$

In a k-tape non-deterministic $f(n)$ space bounded TM on input x of length

References

[OD] Computational Complexity *A Conceptual Perspective* by *Oded Goldreich*, Pub. Cambridge University Press, 2008, ISBN 978-0-521-88473-0.

[MS] *Theory of Computation* by Michael Sipser, (3rd. ed.) Pub. Cengage Learning, 2013, ISBN 978-81-315-2529-6a .

[SABB] *Computational Complexity, A Modern Approach* by Sanjeev Arora & Boaz Barak, Pub. Cambridge University Press, 2009, ISBN 978-0-521-42426-4.

[CHP] Computational Complexity by *Christos H Papadimitriou*, Pub. Addision-Wesley, 1994, ISBN 0-201-53082-1.

[AW] Mathematics + Computation *A Theory Revolutionizing Technology and Science* by *Avi Wigderson*, Pub. Princeton University Press, 2019, ISBN 978-0-691-18913-0.

[DCK1] *Theory of Computation* by Dexter C Kozen, Pub. Springer, 2006, ISBN 978-81-8128-696-3.