MODULE-I

What is TOC?

In theoretical computer science, the **theory of computation** is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory.

In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine.

Automata theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.

This automaton consists of

- **states** (represented in the figure by circles),
- and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.

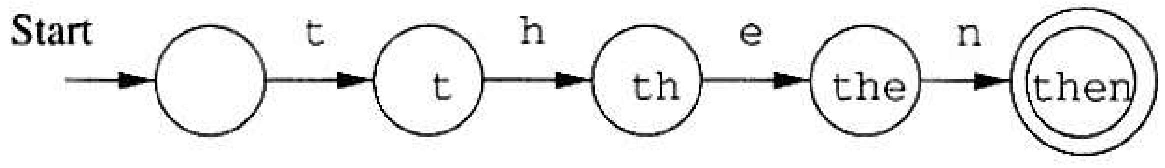


Figure 1.2: A finite automaton modeling recognition of then

Introduction to formal proof:

Basic Symbols used :

U – Union

∩- Conjunction

□ - Empty String

Φ – NULL set

7- negation

‘ – compliment

= > implies

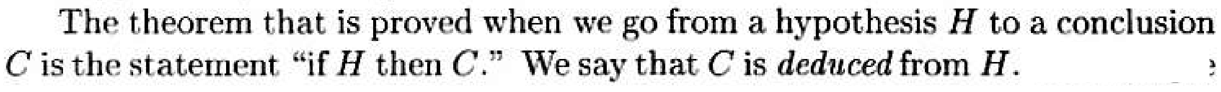
Additive inverse: $a+(-a)=0$
Multiplicative inverse: $a*1/a=1$
Universal set U={1,2,3,4,5}
Subset A={1,3}
A' ={2,4,5}
Absorption law: AU(A ∩B) = A, A∩(AUB) = A

De Morgan's Law:
(AUB)' =A' ∩ B'
(A∩B)' = A' U B'
Double compliment
(A')' =A
A ∩ A' = Φ

Logic relations:
a → b = > **7**a U b
7(a∩b)=**7**a U **7**b

Relations:
Let a and b be two sets a relation R contains aXb.
Relations used in TOC:
Reflexive: a = a
Symmetric: aRb = > bRa
Transition: aRb, bRc = > aRc
If a given relation is reflexive, symmentric and transitive then the relation is called equivalence relation.

Deductive proof: Consists of sequence of statements whose truth lead us from some initial statement called the hypothesis or the give statement to a conclusion statement.


The theorem that is proved when we go from a hypothesis H to a conclusion C is the statement "if H then C." We say that C is *deduced* from H.

Additional forms of proof:
Proof of sets
Proof by contradiction
Proof by counter example

Direct proof (AKA) Constructive proof:
If *p* is true then *q* is true
Eg: if a and b are odd numbers then product is also an odd number.
Odd number can be represented as 2n+1
a=2x+1, b=2y+1
product of a X b = (2x+1) X (2y+1)
= 2(2xy+x+y)+1 = 2z+1 (odd number)

Proof by contrapositive:

The *contrapositive* of the statement "if H then C" is "if not C then not H." A statement and its contrapositive are either both true or both false, so we can prove either to prove the other.

Theorem 1.10: $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

	Statement	Justification
1.	x is in $R \cup (S \cap T)$	Given
2.	x is in R or x is in $S \cap T$	(1) and definition of union
3.	x is in R or x is in both S and T	(2) and definition of intersection
4.	x is in $R \cup S$	(3) and definition of union
5.	x is in $R \cup T$	(3) and definition of union
6.	x is in $(R \cup S) \cap (R \cup T)$	(4), (5), and definition of intersection

Figure 1.5: Steps in the "if" part of Theorem 1.10

	Statement	Justification
1.	x is in $(R \cup S) \cap (R \cup T)$	Given
2.	x is in $R \cup S$	(1) and definition of intersection
3.	x is in $R \cup T$	(1) and definition of intersection
4.	x is in R or x is in both S and T	(2), (3), and reasoning about unions
5.	x is in R or x is in $S \cap T$	(4) and definition of intersection
6.	x is in $R \cup (S \cap T)$	(5) and definition of union

Figure 1.6: Steps in the "only-if" part of Theorem 1.10

To see why "if H then C" and "if not C then not H" are logically equivalent, first observe that there are four cases to consider:

1. H and C both true.

2. H true and C false.

3. C true and H false.

4. H and C both false.

Proof by Contradiction:

H and not C implies falsehood.

That is, start by assuming both the hypothesis H and the negation of the conclusion C. Complete the proof by showing that something known to be false follows logically from H and not C. This form of proof is called *proof by contradiction.*

It often is easier to prove that a statement is not a theorem than to prove it *is* a theorem. As we mentioned, if S is any statement, then the statement "S is not a theorem" is itself a statement without parameters, and thus can

Be regarded as an observation than a theorem.

Alleged Theorem 1.13: All primes are odd. (More formally, we might say: if integer x is a prime, then x is odd.)

DISPROOF: The integer 2 is a prime, but 2 is even. □

For any sets a,b,c if a∩b = Φ and c is a subset of b the prove that a∩c =Φ
Given : a∩b=Φ and c subset b
Assume: a∩c ≠ Φ
Then $\forall x,\ x\epsilon a \text{ and } x\epsilon c \Rightarrow x\epsilon b$
= > a∩b ≠Φ = > a∩c=Φ(i.e., the assumption is wrong)

Proof by mathematical Induction:

Suppose we are given a statement $S(n)$, about an integer n, to prove. One common approach is to prove two things:

1. The *basis*, where we show $S(i)$ for a particular integer i. Usually, $i = 0$ or $i = 1$, but there are examples where we want to start at some higher i, perhaps because the statement S is false for a few small integers.

2. The *inductive step*, where we assume $n \geq i$, where i is the basis integer, and we show that "if $S(n)$ then $S(n+1)$."

- ***The Induction Principle*: If we prove $S(i)$ and we prove that for all $n \geq i$, $S(n)$ implies $S(n+1)$, then we may conclude $S(n)$ for all $n \geq i$.**

Languages :

The languages we consider for our discussion is an abstraction of natural languages. That is, our focus here is on formal languages that need precise and formal definitions. Programming languages belong to this category.

Symbols :

Symbols are indivisible objects or entity that cannot be defined. That is, symbols are the atoms of the world of languages. A symbol is any single object such as ♠, *a*, 0, 1, #, begin, or do.

Alphabets :

An alphabet is a finite, nonempty set of symbols. The alphabet of a language is normally denoted by Σ. When more than one alphabets are considered for discussion, then subscripts may be used (e.g. Σ_1, Σ_2 etc) or sometimes other symbol like G may also be introduced.

$$\Sigma = \{0, 1\}$$
$$\Sigma = \{a, b, c\}$$
$$\Sigma = \{a, b, c, \&, z\}$$

Example : $\Sigma = \{\#, \nabla, ♠, \beta\}$

Strings or Words over Alphabet :

A string or word over an alphabet Σ is a finite sequence of concatenated symbols of Σ.

Example : 0110, 11, 001 are three strings over the binary alphabet { 0, 1 } .

aab, abcb, b, cc are four strings over the alphabet { a, b, c }.

It is not the case that a string over some alphabet should contain all the symbols from the alphabet. For example, the string cc over the alphabet { a, b, c } does not contain the symbols a and b. Hence, it is true that a string over an alphabet is also a string over any superset of that alphabet.

Length of a string :
The number of symbols in a string w is called its length, denoted by |w|.

Example : | 011 | = 4, |11| = 2, | b | = 1

Convention : We will use small case letters towards the beginning of the English alphabet to denote symbols of an alphabet and small case letters towards the end to denote strings over an alphabet. That is, $a, b, c \in \Sigma$ (symbols) and u, v, w, x, y, z are strings.

Some String Operations :
Let $x = a_1a_2a_3 \in a_n$ and $y = b_1b_2b_3 \in b_m$ be two strings. The concatenation of x and y denoted by xy, is the string $a_1a_2a_3 \cdots a_nb_1b_2b_3 \cdots b_m$. That is, the concatenation of x and y denoted by xy is the string that has a copy of x followed by a copy of y without any intervening space between them.

Example : Consider the string 011 over the binary alphabet. All the prefixes, suffixes and substrings of this string are listed below.

Prefixes: ε, 0, 01, 011.
Suffixes: ε, 1, 11, 011.
Substrings: ε, 0, 1, 01, 11, 011.

Note that x is a prefix (suffix or substring) to x, for any string x and ε is a prefix (suffix or substring) to any string.

A string x is a proper prefix (suffix) of string y if x is a prefix (suffix) of y and $x \neq y$.

In the above example, all prefixes except 011 are proper prefixes.

Powers of Strings : For any string x and integer $n \geq 0$, we use x^n to denote the string formed by sequentially concatenating n copies of x. We can also give an inductive definition of x^n as follows:
x^n = e, if n = 0 ; otherwise $x^n = xx^{n-1}$

Example : If x = 011, then x^3 = 011011011, x^1 = 011 and $x^0 = e$

Powers of Alphabets :

We write Σ^k (for some integer k) to denote the set of strings of length k with symbols from Σ. In other words,

Σ^k = { w | w is a string over Σ and | w | = k}. Hence, for any alphabet, Σ^0 denotes the set of all strings of length zero. That is, Σ^0 = { e }. For the binary alphabet { 0, 1 } we have the following.

$\Sigma^0 = \{e\}$.

$\Sigma^1 = \{0, 1\}$.

$\Sigma^2 = \{00, 01, 10, 11\}$.

$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

The set of all strings over an alphabet Σ is denoted by Σ^*. That is,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

The set Σ^* contains all the strings that can be generated by iteratively concatenating symbols from Σ any number of times.

Example : If Σ = { a, b }, then Σ^* = { ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, …}.

Please note that if $\Sigma = F$, then Σ^* that is $\phi^* = \{e\}$. It may look odd that one can proceed from the empty set to a non-empty set by iterated concatenation. But there is a reason for this and we accept this convention

The set of all nonempty strings over an alphabet Σ is denoted by Σ^*. That is,

$$\Sigma^+ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

Note that Σ^* is infinite. It contains no infinite strings but strings of arbitrary lengths.

Reversal :

For any string $w = a_1a_2a_3\cdots a_n$ the reversal of the string is $w^R = a_na_{n-1}\cdots a_3a_2a_1$.

An inductive definition of reversal can be given as follows:

Languages :
A language over an alphabet is a set of strings over that alphabet. Therefore, a language L is any subset of Σ^*. That is, any $L \subseteq \Sigma^*$ is a language.

Example :

1. F is the empty language.
2. Σ^* is a language for any Σ.
3. {e} is a language for any Σ. Note that, $\phi \neq \{e\}$. Because the language F does not contain any string but {e} contains one string of length zero.
4. The set of all strings over { 0, 1 } containing equal number of 0's and 1's.
5. The set of all strings over {a, b, c} that starts with a.

Convention : Capital letters A, B, C, L, etc. with or without subscripts are normally used to denote languages.

Set operations on languages : Since languages are set of strings we can apply set operations to languages. Here are some simple examples (though there is nothing new in it).

Union : A string $x \in L_1 \cup L_2$ iff $x \in L_1$ or $x \in L_2$

Example : { 0, 11, 01, 011 } $\cup$ { 1, 01, 110 } = { 0, 11, 01, 011, 111 }

Intersection : A string, $x \in L_1 \cap L_2$ iff $x \in L_1$ and $x \in L_2$.

Example : { 0, 11, 01, 011 } $\cap$ { 1, 01, 110 } = { 01 }

Complement **:** Usually, Σ^* is the universe that a complement is taken with respect to.
Thus for a language L, the complement is L(bar) = { $x \in \Sigma^*$ | $x \notin L$ }.

Example : Let L = { x | |x| is even }. Then its complement is the language { $x \in \Sigma^*$ | $|x|$ is odd }.
Similarly we can define other usual set operations on languages like relative complement, symmetric difference, etc.

Reversal of a language :
The reversal of a language *L*, denoted as L^R, is defined as: $L^R = \{w^R \mid w \in L\}$.

Example :

1. Let L = { 0, 11, 01, 011 }. Then L^R = { 0, 11, 10, 110 }.

2. Let L = { 1^n0^n | n is an integer }. Then L^R = { 1^n0^n | n is an integer }.

Language concatenation : The concatenation of languages L_1 and L_2 is defined as $L_1 L_2$ = { *xy* | $x \in L_1$ and $y \in L_2$ }.

Example : { *a*, *ab* }{ *b*, *ba* } = { *ab*, *aba*, *abb*, *abba* }.

Note that ,

1. $L_1 L_2 \neq L_2 L_1$ in general.
2. $L\Phi = \Phi$
3. $L\{\varepsilon\} = L = \{\varepsilon\}$

Iterated concatenation of languages : Since we can concatenate two languages, we also repeat this to concatenate any number of languages. Or we can concatenate a language with itself any number of times. The operation L^n denotes the concatenation of L with itself n times. This is defined formally as follows:

$L_0 = \{e\}$

$L^n = LL^{n-1}$

Example : Let L = { a, ab }. Then according to the definition, we have

$L_0 = \{e\}$

$L_1 = L\{e\} = L = \{a, ab\}$

$L_2 = L\,L_1 = \{a, ab\}\{a, ab\} = \{aa, aab, aba, abab\}$

$L_3 = L\,L_2 = \{a, ab\}\{aa, aab, aba, abab\}$

$\quad = \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\}$

and so on.

Kleene's Star operation : The Kleene star operation on a language L, denoted as L^* is defined as follows :

L^* = (Union n in N) L^n

= $L^0 \cup L^1 \cup L^2 \cup \cdots$

= { x | x is the concatenation of zero or more strings from L }

Thus L^* is the set of all strings derivable by any number of concatenations of strings in L. It is also useful to define

$L^+ = LL^*$, i.e., all strings derivable by one or more concatenations of strings in L. That is

L^+ = (Union n in N and n >0) L^n
$= L^1 \cup L^2 \cup L^3 \cup \cdots$

Example : Let L = { a, ab }. Then we have,

$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$

$= \{e\} \cup \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

$L^+ = L^1 \cup L^2 \cup L^3 \cup \cdots$

$= \{a, ab\} \cup \{aa, aab, aba, abab\} \cup \ldots$

Note : ε is in L^*, for every language L, including .

The previously introduced definition of Σ^* is an instance of Kleene star.

$$\begin{tikzcd}\text{Grammar} \arrow[r, "\text{(Generates)}"] & \text{Language} & \text{Automata} \arrow[l, "\text{(Recognizes)}"']\end{tikzcd}$$

Automata: A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string.

An automata is an abstract computing device (or machine). There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

Every Automaton fulfills the three basic requirements.

- Every automaton consists of some essential features as in real computers. It has a mechanism for reading input. The input is assumed to be a sequence of symbols over a given alphabet and is placed on an input tape(or written on an input file). The simpler automata can only read the input one symbol at a time from left to right but not change. Powerful versions can both read (from left to right or right to left) and change the input.

- The automaton can produce output of some form. If the output in response to an input string is binary (say, accept or reject), then it is called an accepter. If it produces an output sequence in response to an input sequence, then it is called a transducer(or automaton with output).
- The automaton may have a temporary storage, consisting of an unlimited number of cells, each capable of holding a symbol from an alphabet (whcih may be different from the input alphabet). The automaton can both read and change the contents of the storage cells in the temporary storage. The accusing capability of this storage varies depending on the type of the storage.
- The most important feature of the automaton is its control unit, which can be in any one of a finite number of interval states at any point. It can change state in some defined manner determined by a transition function.

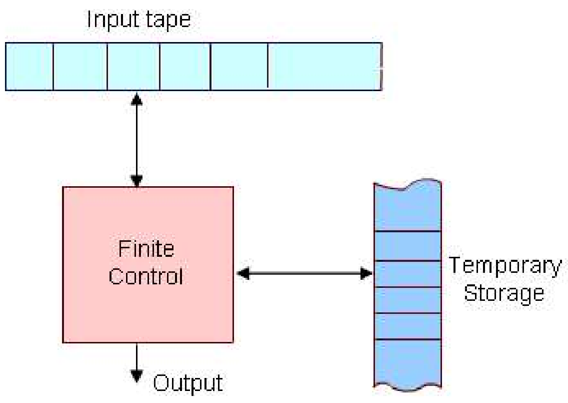


Figure 1: The figure above shows a diagrammatic representation of a generic automation.

Operation of the automation is defined as follows.
At any point of time the automaton is in some integral state and is reading a particular symbol from the input tape by using the mechanism for reading input. In the next time step the automaton then moves to some other integral (or remain in the same state) as defined by the transition function. The transition function is based on the current state, input symbol read, and the content of the temporary storage. At the same time the content of the storage may be changed and the input read may be modifed. The automation may also produce some output during this transition. The internal state, input and the content of storage at any point defines the configuration of the automaton at that point. The transition from one configuration to the next (as defined by the transition function) is called a *move.* Finite state machine or *Finite Automation* is the simplest type of abstract machine we consider. Any system that is at any point of time in one of a finite number of interval state and moves among these states in a defined manner in response to some input, can be modeled by a finite automaton. It doesnot have any temporary storage and hence a restricted model of computation.

Finite Automata

Automata (singular : automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons.

States, Transitions and Finite-State Transition System :

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.

Transitions are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction).

Some examples of state transition systems are: digital systems, vending machines, etc. A system

containing only a finite number of states and transitions among them is called a *finite-state transition system*.

Finite-state transition systems can be modeled abstractly by a mathematical model called *finite automation*

Deterministic Finite (-state) Automata

Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an input string -- one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far. A machine for which a deterministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

Thus, a DFA conceptually consists of 3 parts:

1. A *tape* to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from Σ.
2. A *tape head* for reading symbols from the tape
3. A *control* , which itself consists of 3 things:
    - finite number of states that the machine is allowed to be in (zero or more states are designated as *accept* or *final* states),
    - a current state, initially set to a start state,

- a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current tape cell and sends the symbol s found there to the control. Then the tape head moves to the next cell.
2. he control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined.

If it is an accept state , the input string is accepted ; otherwise, the string is rejected . Summarizing all the above we can formulate the following formal definition:

Deterministic Finite State Automaton : A Deterministic Finite State Automaton (DFA) is a 5-tuple : $M = (Q, \Sigma , \delta, q_0, F)$

- Q is a finite set of states.
- $\Sigma$is a finite set of input symbols or alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is the “next state” transition function (which is total). Intuitively, is δ a function that tells which state to move to in response to an input, i.e., if M is in state q and sees input a, it moves to state $\delta(q,a)$.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$is the set of accept or final states.

Acceptance of Strings :

A DFA accepts a string $w = a_1 a_2 \cdots a_n$ if there is a sequence of states $q_0, q_1, \cdots, q_n$ in Q such that

1. $q_0$is the start state.
2. $\delta(q_i, q_{i+1}) = a_{i+1}$ for all $0 < i < n$.
3. $q_n \in F$

Language Accepted or Recognized by a DFA :

The language accepted or recognized by a DFA M is the set of all strings accepted by M , and is denoted by $L(M)$ i.e. $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$. The notion of acceptance can also be made more precise by extending the transition function δ.

Extended transition function :

Extend $\delta : Q \times \Sigma \to Q$ (which is function on symbols) to a function on strings, i.e. .
$\hat{\delta} : Q \times \Sigma^* \to Q$

That is, $\hat{\delta}(q, w)$ is the state the automation reaches when it starts from the state q and finish processing the string w. Formally, we can give an inductive definition as follows:
The language of the DFA M is the set of strings that can take the start state to one of the accepting states i.e.

$$L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

$$= \{ w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F \}$$

Example 1 :

$M = (Q, \Sigma, \delta, q_0, F)$

$Q = \{q_0, q_1\}$

q_0 is the start state

$F = \{q_1\}$

$\delta(q_0, 0) = q_0$ $\delta(q_1, 0) = q_1$

$\delta(q_0, 1) = q_1$ $\delta(q_1, 1) = q_1$

It is a formal description of a DFA. But it is hard to comprehend. For ex. The language of the DFA is any string over { 0, 1} having at least one 1

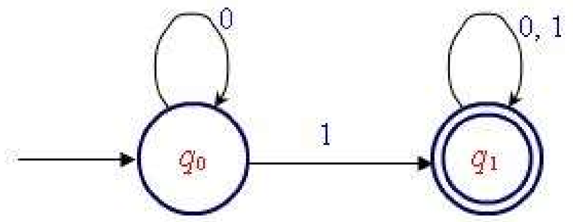
We can describe the same DFA by transition table or state transition diagram as following:

Transition Table :

	0	1
$\to q_0$	q_0	q_1

$*q_1$	q_1	q_1

It is easy to comprehend the transition diagram.



Explanation : We cannot reach find state q_1 w/0 or in the i/p string. There can be any no. of 0's at the beginning. (The self-loop at q_0 on label 0 indicates it). Similarly there can be any no. of 0's & 1's in any order at the end of the string.

Transition table :

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the "next state").

- Rows correspond to states,
- Columns correspond to input symbols,
- Entries correspond to next states
- The start state is marked with an arrow
- The accept states are marked with a star (*).

	0	1
$\rightarrow q_0$	q_0	q_1
$*q_1$	q_1	q_1

(State) Transition diagram :

A state transition diagram or simply a transition diagram is a directed graph which can be constructed as follows:

1. For each state in Q there is a node.
2. There is a directed edge from node q to node p labeled a iff $\delta(q, a) = p$. (If there are several input symbols that cause a transition, the edge is labeled by the list of these symbols.)
3. There is an arrow with no source into the start state.
4. Accepting states are indicated by double circle.