Addis Ababa University

College of Natural and Computational Sciences

Department of Computer Science

Compiler and Complexity Module

Part I: Automata and Complexity Theory

Part II: Compiler Design

May 2024

Addis Ababa,

Ethiopia

Automata and Complexity Theory

Theory of Automata

The theory of automata is a theoretical branch of computer science and mathematics. It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the automata. The main motivation behind developing the automata theory was to develop methods to describe and analyze the dynamic behavior of discrete systems.

This automaton consists of **states** and **transitions**. The **State** is represented by **circles**, and the **Transition** is represented by **arrows**. Automata is the kind of machine that takes some string as input and this input goes through a finite number of states and may enter the **final state**.

The basic terminologies used in Automata Theory:- Alphabets, Strings, Languages, and Grammar.

Symbols: Symbols are an entity or individual objects, which can be any letter, alphabet or any picture. E.g.: 1, a, b, #

Alphabets:

An alphabet finite set of non-empty symbols. It is denoted by ∑ (sigma).

Examples:

- ∑ = {a, b} is the binary alphabet
- ∑ = {A, B, C, D} is the set of uppercase letters
- ∑ = {(,)} is the set of open and closed brackets

String:

It is a finite collection of symbols from the alphabet. The string is denoted by w.

Example 1:

If ∑ = {a, b}, various string that can be generated from ∑ are {ab, aa, aaa, bb, bbb, ba, aba .... }.

- A string with zero occurrences of symbols is known as an empty string. It is represented by ε.
- The number of symbols in a string w is called the length of a string. It is denoted by |w|.

Example 2:

- w = 010; Number of Sting |w| = 3
- abcbz is a string over ? = {a, b, c, d… z}
- 11001011 is a string over ? = {0, 1}
-)) ()(() is a string over ? = {(,)}

Language:

A language is a collection of appropriate string. A language which is formed over Σ can be **Finite** or **Infinite**.

Example: 1

L1 = {Set of string of length 2}

= {aa, bb, ba, bb} **Finite Language**

Example: 2

L2 = {Set of all strings starts with 'a'}

= {a, aa, aaa, abb, abbb, ababb}

Infinite Language

Grammar

A grammar defines a set of rules, and with the use of these rules, valid sentences in a language are generated. Similar to English language we have set of rules in automata, which help us in creating sentences. A popular way to derive a grammar recursively is phrase-structure grammar.

Finite Automata

- Finite automata are used to recognize patterns.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.
- Finite automata have two states, **Accept state** or **Reject state**. When the input string is processed successfully, and the automata reached its final state, then it will accept.

Formal Definition of FA

A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: **final** state
5. δ: Transition function

Finite Automata Model: Finite automata can be represented by input tape and finite control.

Input tape: It is a linear tape having some number of cells. Each input symbol is placed in each cell.

Finite control: The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.

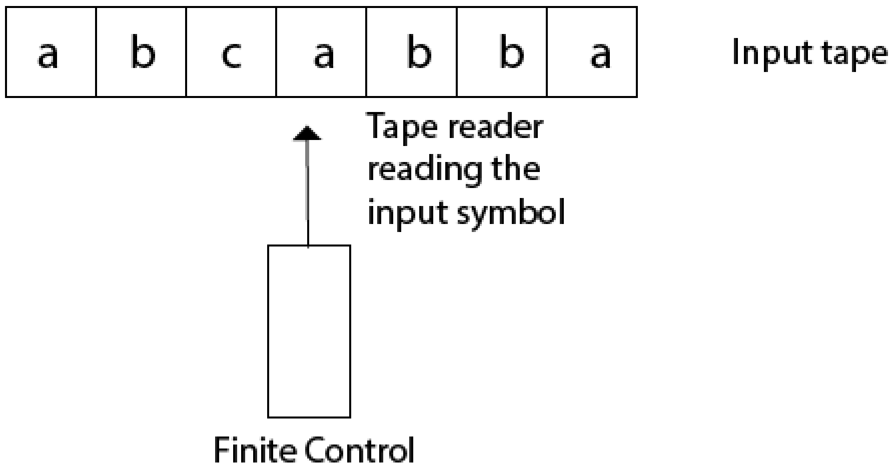
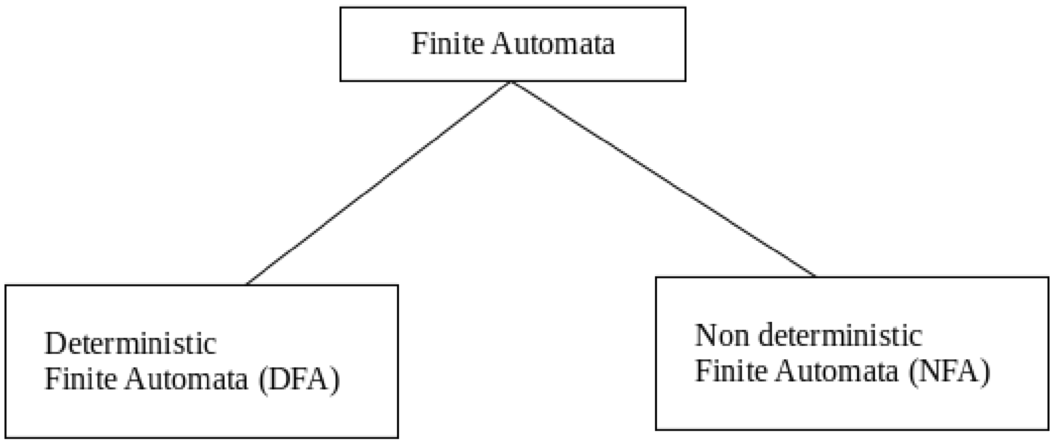


Fig: - Finite automata model

Types of Automata:

There are two types of finite automata:

1. DFA(deterministic finite automata)
2. NFA(non-deterministic finite automata)


Finite Automata
Deterministic Finite Automata (DFA)
Non deterministic Finite Automata (NFA)

1. DFA

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

2. NFA

NFA stands for non-deterministic finite automata. It is used to transmit any number of states for a particular input. It can accept the null move.

Some important points about DFA and NFA:

1. Every DFA is NFA, but NFA is not DFA.
2. There can be multiple final states in both NFA and DFA.
3. DFA is used in Lexical Analysis in Compiler.
4. NFA is more of a theoretical concept.

Regular Expression

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expression are referred to as Regular languages.
- A regular expression can also be described as a sequence of pattern that defines a string.
- Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

For instance:

In a regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx, }

In a regular expression, x^+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx, ...}

Operations on Regular Language

The various operations on regular language are:

Union: If L and M are two regular languages then their union L U M is also a union.

1. 1. L U M = {s | s is in L or s is in M}

Intersection: If L and M are two regular languages then their intersection is also an intersection.

1. 1. L ∩ M = {st | s is in L and t is in M}

Kleen closure: If L is a regular language then its Kleen closure L1* will also be a regular language.

1. 1. L* = Zero or more occurrence of language L.

Conversion of RE to FA

To convert the RE to FA, we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below:

Step 1: Design a transition diagram for given regular expression, using NFA with ε moves.

Step 2: Convert this NFA with ε to NFA without ε.

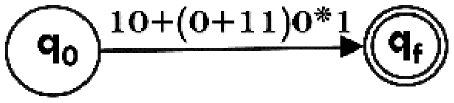
Step 3: Convert the obtained NFA to equivalent DFA.

Example 1:

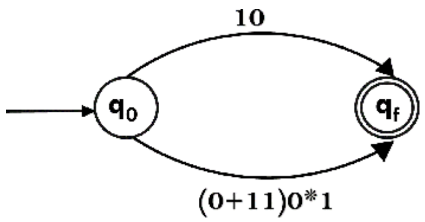
Design a FA from given regular expression 10 + (0 + 11)0* 1.

Solution: First we will construct the transition diagram for a given regular expression.

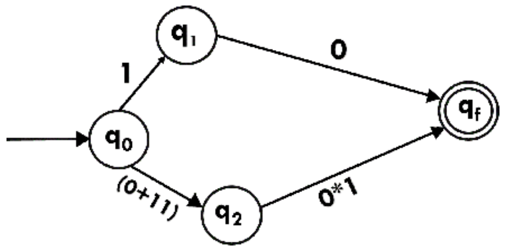
Step 1:



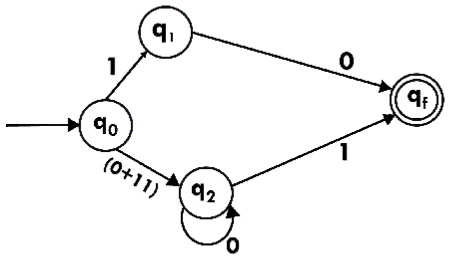
Step 2:



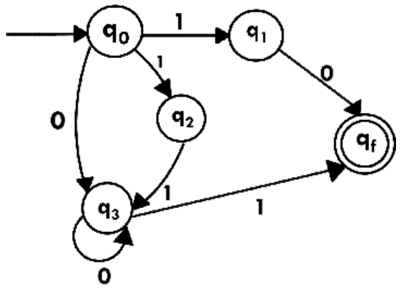
Step 3:



Step 4:



Step 5:



Now we have got NFA without ε. Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

State	0	1
→q0	q3	{q1, q2}
q1	qf	ϕ
q2	ϕ	q3
q3	q3	qf
*qf	ϕ	ϕ

The equivalent DFA will be:

State	0	1
→[q0]	[q3]	[q1, q2]
[q1]	[qf]	ϕ
[q2]	ϕ	[q3]
[q3]	[q3]	[qf]
[q1, q2]	[qf]	[qf]
*[qf]	ϕ	ϕ

Regular Languages

A language is regular if it can be described by a regular expression.

The Regular Languages (L_{REG}) is the set of all languages that can be represented by a regular expression, i.e. set of set of strings.

A language is said to be a Regular language if and only if some finite state machine recognizes it

So what Language are not Regular?

The language

- which is not recognized by any FMS and
- which require memory
    - Memory of FMS is very limited
    - It can't store and count string

The languages accepted by all DFAs form the family of regular languages.

The language defined by regular grammar is known as regular language. Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language.

There are two Pumping Lemmas, which are defined for 1. Regular Languages and 2. Context – Free Languages Pumping Lemma for Regular Languages For any regular language L, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists $u, v, w \in \Sigma^*$, such that $x = uvw$, and (1) $|uv| \leq n$ (2) $|v| \geq 1$ (3) for all $i \geq 0$: $uviw \in L$ In simple terms, this means that if a string v is 'pumped', i.e., if v is inserted any number of times, the resultant string still remains in L. Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular. The opposite of this may not always be true. That is, if Pumping Lemma holds, it does not mean that the language is regular.

Context free languages

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

Context-free grammars have a start symbol and set of terminals, non-terminals, and production rule components.

A context-free grammar can be described by a four-element tuple (V, Σ, R, S), where

- V is a finite set of variables (which are non-terminal);
- Σ is a finite set (disjoint from V) of terminal symbols;
- R is a set of production rules where each production rule maps a variable to a string $s \in (V \cup \Sigma)*$;
- *S* (which is in *V*) which is a start symbol.

Context-free grammars can be modelled as parse trees. The nodes of the tree represent the symbols and the edges represent the use of production rules. The leaves of the tree are the end result (terminal symbols) that make up the string the grammar is generating with that particular sequence of symbols and production rules. Grammar can be implemented with multiple parse trees to get the same resulting string, this is said to be ambiguous.

A sentential form is any string derivable from the start symbol. Thus, in the derivation of a + a * a, E + T * F and E + F * a and F + a * a are all sentential forms as are E and a + a * a themselves. A sentence is a sentential form consisting only of terminals such as a + a * a.

The derivation tree is a graphical representation of the given production rules of context-free grammar (CFG). It is a way to show how the derivation can be done to obtain some string from a given set of production rules. Two types of derivation; in the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives, we read the input string from left to right, and in the rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in most right derivatives, we read the input string from right to left.

In CFG, all the grammar is not always optimized which means the grammar may consist of some extra symbols (non-terminal). Having additional symbols, unnecessarily increase the length of grammar. Simplification of grammar means a reduction of grammar by removing useless symbols. Simplification essentially comprises the following steps; Reduction of CFG, Removal of Unit Productions, and Removal of Null Productions.

Pushdown automata

Pushdown automata are nondeterministic finite state machines augmented with additional memory in the form of a stack, which is why the term “pushdown” is used, as elements are pushed down onto the stack. Pushdown automata are computational models—theoretical computer-like machines—that can do more than a finite state machine, but less than a Turing machine. A pushdown automaton is formally defined as a 7-tuple: $(Q, \Sigma, \Gamma, \delta, q_0, Z, F)$.

Pushdown automata accept context-free languages, which include the set of regular languages. The language that describes strings that have matching parentheses is a context-free language. Say that a programmer has written some code, and in order for the code to be valid, any parentheses must be matched. One way to do this would be to feed the code (as strings) into a pushdown automaton programmed with transition functions that implement the context-free grammar for the language of balanced parentheses. If the code is valid and all parentheses are matched, the pushdown automata will "accept" the code. If there are unbalanced parentheses, the pushdown automaton will be able to return to the programmer that the code is not valid. This is one of the more theoretical ideas behind computer parsers and compilers.

Pushdown automata can be useful when thinking about parser design and any area where context-free grammars are used, such as in computer language design. Since pushdown automata are equal in power to context-free languages, there are two ways of proving that a language is context-free: provide the context-free grammar or provide a pushdown automaton for the language.

A non-deterministic pushdown automaton (NPDA), or just pushdown automaton (PDA) is a variation on the idea of a non-deterministic finite automaton (NDFA). Unlike an NDFA, a PDA is associated with a stack (hence the name pushdown). The transition function must also take into account the “state” of the stack.

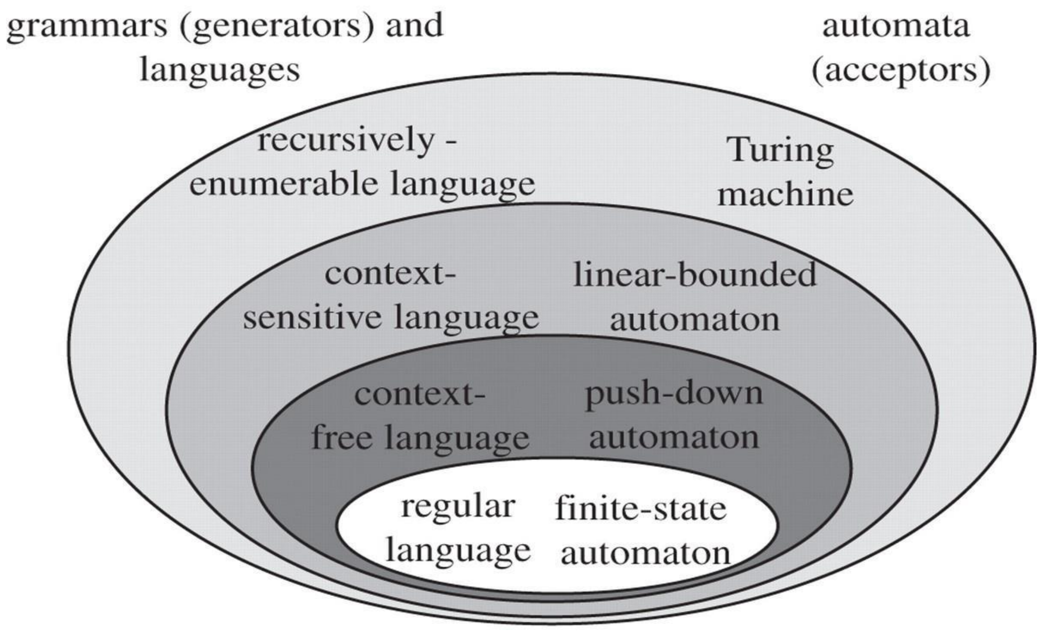


Fig: The traditional Chomsky hierarchy

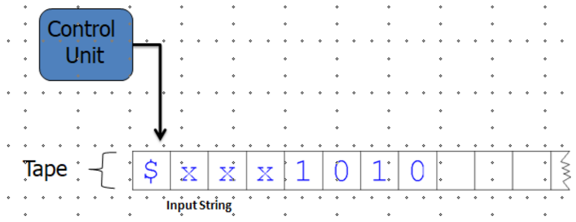
TURING MACHINE

STANDARD TURING MACHINE

Turing machines, first described by Alan Turing, are **simple abstract computational devices intended to help investigate the extent and limitations of what can be computed**.

- Turing machine is a model of general purpose computer.
- A Turing machine can do everything that a real computer can do.
- The Turing machine model uses an infinite tape as its unlimited memory.
- It has a tape head that can read and write symbols and move around on the tape.
- Initially the tape contains only the input string and is blank everywhere else.

- The Turing machine model uses an infinite tape as its unlimited memory.
- It has a tape head that can read and write symbols and move around on the tape.
- Initially the tape contains only the input string and is blank everywhere else.



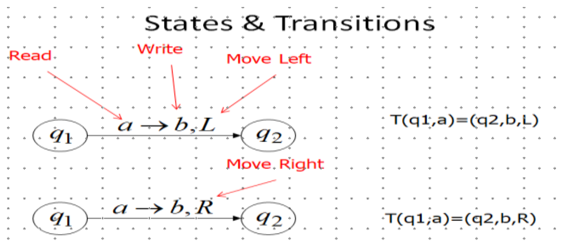
Turing machine model

A Turing machine M is a **7-tuple**, namely **(Q, ∑, ┌, δ, qo, b, F)**, where

1. **Q** is a finite nonempty set of states;
2. ∑ is a nonempty set of input symbols and is subset of ┌ and b∉∑;
3. ┌ is a finite nonempty set of tape symbols;
4. **b**∉ ┌is the blank symbol;
5. **δ** is the **transition function**, mapping (**q**, **x**) onto (**q'**, **y**, **D**) where **D** denotes the direction of **movement of R/W head**: **D = L or R** according as the movement is to the left or right.
6. **qo**∈ **Q** is the initial state , and
7. **F**⊆**Q** is the set of final states.

   NB - The acceptability of a string is decided by the reachability from the initial state to some final state, and such states are called the accepting states.

   - δ may not be defined for some elements of **QX ┌.**



Turing Machines are deterministic: for each state there is only one unique Transition on each symbol.

Partial Transition Function: no transition for all input symbol

Representation of Turing Machine

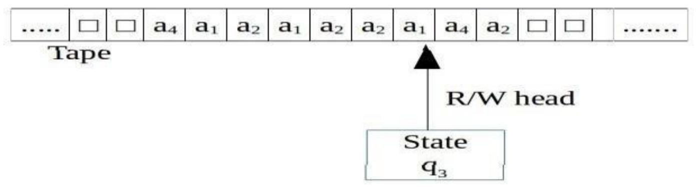
- We can describe a Turing machine by:

1. **Instantaneous descriptions**
2. **Transition table**, and
3. **Transition diagram** (**transition graph**).

Instantaneous descriptions using move-relations ($\vdash$)

- Machine which is defined in terms of the entire ***input string*** and the ***current state***.
- An **ID** of a Turing machine $\boldsymbol{M}$ is a string $\boldsymbol{\alpha\beta\gamma}$, where
- $\boldsymbol{\beta}$ is the present state of $\boldsymbol{M}$, and
- the entire input string is split as $\boldsymbol{\alpha\gamma}$, the first symbol of $\boldsymbol{\gamma}$ is the current symbol $\boldsymbol{a}$ under the R/W head and $\boldsymbol{\gamma}$ has all the subsequent symbols of the input string, and
- The string $\boldsymbol{\alpha}$ is the substring of the input string formed by all the symbols to the left of $\boldsymbol{a}$.

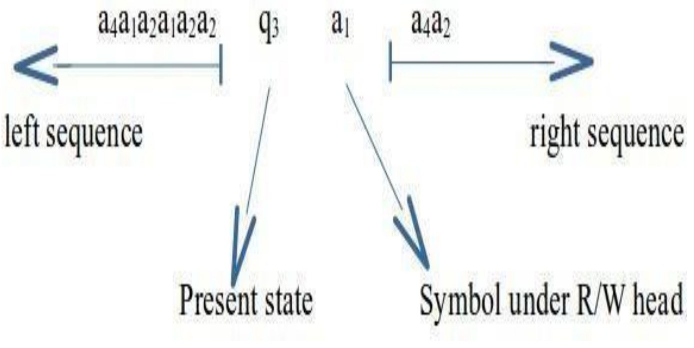
Example: A snapshot of Turing machine is shown in figure below. Obtain the instantaneous description?



Solution:

- The present symbol under the R/W head is a_1.
- The present state is q_3. So a_1 is written to the right of q_3.
- The non-blank symbols to the left of a_1 form the string $a_4a_1a_2a_1a_2a_2$, which is written to the left of q_3.
- The sequence of non-blank symbols to the right of a_1 is a_4a_2.

Thus the ID is as given in the figure below.



NB. - For constructing the **ID**, we simply insert the current state in the input string to the left of the symbol under the R/W head.

- We observe that the blank symbol may occur as part of the left or right substring.

Transition table

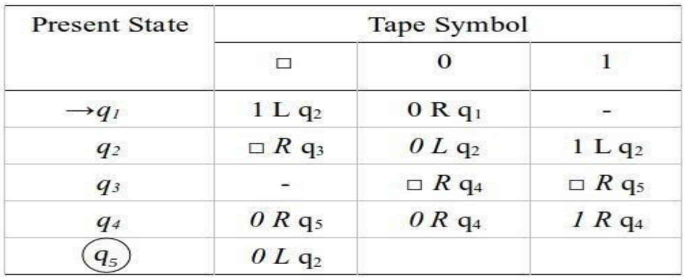
We give the definition of transition function $\boldsymbol{\delta}$ in the form of a table called the ***transition table***.

– If $\delta(q, a) = (\gamma, \alpha, \beta)$, we write $\alpha\beta\gamma$ under the α–column and in the q–row.

– So if we get $\alpha\beta\gamma$ in the table, it means that α is written in the current cell, β gives the movement of the head (L or R) and γ denotes the new state into which the Turing machine enters.

– Consider, for example, a Turing machine with five states q_1, ..., q_5, where $q_1$is the initial state and q_5 is the (only) final state, and the tape symbols(G) are 0, 1 and □.

– The transition table given in the Table below describes δ.



Present State	Tape Symbol		
	□	0	1
$\rightarrow q_1$	1 L q_2	0 R q_1	-
q_2	□ R q_3	0 L q_2	1 L q_2
q_3	-	□ R q_4	□ R q_5
q_4	0 R q_5	0 R q_4	1 R q_4
$\textcircled{q_5}$	0 L q_2		

#**Note**: The initial state is marked with → and the final state is marked with a circle.

● We describe the computation sequence in terms of the contents of the tape and the current state.

Transition diagram (transition graph).

We can use the transition systems (diagrams) to represent Turing machines.

– The states are represented by vertices.

– Directed edges are used to represent transition of states.
– The labels are triples of the form (α, β, γ), where α, β Î Γ (where Γis set of tape symbols) and γ Î{L, R}.

When there is a directed edge from q_i to q_j with label (α,β,γ),it means that $\delta(q_i, \alpha) = (q_j, \beta, \gamma)$

– During the processing of an input string, suppose the Turing machine enters q_i and the R/W head scans the (present) symbol α.
– As a result the symbol is β written in the cell under the R/W head.
– The R/W head moves to the left or to the right depending on γ, and the new state is q_j.
– Every edge in the transition system can be represented by a 5-tuple ($q_i, \alpha, \beta, \gamma, q_j$).
– So each Turing machine can be described by the sequence of 5-tuples representing all the directed edges.
– The initial state is indicated by → and any final state is marked with O.

There are **three possible outcomes** of executing a Turing machine over a given input.

(1) Halt and accept the input
(2) Halt and reject the input
(3) Never halt

Undecidable problem	Decidable problem
-It Is impossible to construct a single algorithm that always leads to a correct yes-or-no answer. -Undecidable problems are those which are **<u>not recursive.</u>**	-A language is **decidable** if there is a Turing machine (decider) which accepts and halts on every input string. Also known as ***<u>recursive languages</u>***
-There is no Turing machine to solve an undecidable problem. We have not said that undecidable means we don't know of a solution today but might find one tomorrow.	- Every decidable language is Turing-Acceptable.

A Turing machine that **accepts** the language (input): If machine halts in a final state.

A Turing machine that **rejects** the language (input): If machine halts in **a non-final state** or If machine enters an ***infinite loop***.

Because of the **infinite loop**: -The final state **cannot** be **reached**, the machine **never halts** and the input is **not accepted**

A function **F** is **computable**if there is a Turing Machine with a transition from **initial** state to **final** state.

Decidable and Undecidable problem

Partially Computable Languages: If the characteristic function is partially computable, then the language is said to be partially computable language.

Computability

Recursive function

A function f is recursive if it can be obtained from the initial functions by a finite number of applications of composition, recursion and minimization over regular functions.

Recursive Functions is one that calls upon itself to determine the solution.

Example: $F(0) = 3$, $F(1) = 5$ $F(n+1) = F(n) + F(n-1)$ for all $n > 0$Can be utilized to resolve many problems.

Recursive Language and recursively enumerable languages

Recursivelanguages	Recursive enumerablelanguages
TMs that *always* halt, no matter accepting or non-accepting. **DECIDABLE** PROBLEMS	TM halts sometimes & may not halt sometimes.
A set X is ***recursive*** if we have an algorithm to determine whether a given element belongs to X or not.	A language that is **not** Recursively Enumerable that language is **not accepted** by **any** Turing Machine.
All recursive set is **recursively enumerable**.	

Computational complexity

Big-O notations

- **Algorithm analysis** is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any **algorithm** which solves a given computational problem.
- This resource can be expressed in terms of execution time (***time efficiency***, the most common factor) or memory (***space efficiency***).

- **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n.
- **Space Complexity:** Determine the approximate memory required to solve a problem of size n.
- These estimates provide an insight into reasonable directions of search for efficient **algorithms**.
- The algorithm analysis can be expressed using **Big O notation**.

Big-O notation: is the most commonly used notation for specifying asymptotic complexity, that is, for estimating the rate of growth of complexity functions.

- The function f(n) is O(g(n)) if there exist positive numbers c and N such that $f(n) \leq c.g(n)$ for all $n \geq N$.
- **Example**: $f(n) = n^2 + 5n$, **$f(n)=O(n^2)$**
  Big O Notation Classes
  Constant Time Complexity O(1): Simplest of all complexities Or **not** complex at all.
- If an operation always completes in the **same** amount of CPU time regardless of the input size, it is called **a constant time operation.**
- If it always uses the same amount of memory regardless of the input size, it is called **a constant space operation.**
- **Example**: if an algorithm **increment** each number of length n , this algorithm runs in O(n) time and performs O(1) work for each elements.

Logarithmic time : log n

- An algorithm is said to take **logarithmic time** when $T(n)$ = **O(log *n*)**.
- Where the time complexity of a function only grows logarithmically in relation to the input.

An **example** of logarithmic effort is the **binary search** for a specific element in a sorted array of size *n*.

Linear time O (n):

- An algorithm is said to take **linear time**, the running time increases at most linearly with the size of the input.
- More precisely, this means that there is a constant *c* such that the running time is at most *cn* for every input of size *n*.
- If an algorithm's time/space usage only grows linearly with the number of elements in the input, then it has linear time/space complexity.

- Linear time is the best possible time complexity in situations where the algorithm has to sequentially read its entire input.

Quasilinear Time O(n log n) :

- The effort grows slightly faster than linear because the linear component is multiplied by a logarithmic one.

Quadratic Time O(n^2) :

- The time grows linearly to the square of the number of input elements: If the number of input elements *n* doubles, then the time simply multiplies.

Exponential time O(2^n) :

- An algorithm is said to be **exponential time**, if $T(n)$ is upper bounded by $2^{(n)}$, where (n) is some polynomial in *n*.
- More formally, an algorithm is exponential time if $T(n)$ is bounded by O(2^{nk}) for some constant *k*.

Big O Notation Order

Here are, once again, the described complexity classes, sorted in ascending order of complexity (for sufficiently large values of *n*):

1. *O(1)* – constant time
2. *O(log n)* – logarithmic time
3. *O(n)* – linear time
4. *O(n log n)* – quasilinear time
5. *O(n^2)* – quadratic time

Polynomial and Non Polynomial Class

Polynomial	Non Polynomial
P problems are set of problems which can be solved in polynomial time by deterministic algorithms.	NP problems are the problems which can be solved in non-deterministic polynomial time.
The problem belongs to class P if it's easy to find a solution for the problem.	The problem belongs to NP, if it's easy to check a solution that may have been very tedious to find.
P Problems can be solved and verified in polynomial time. P problems are subset of NP problems.	Solution to NP problems cannot be obtained in polynomial time. but
P problems are subset of NP problems.	NP problems are superset of P problems.