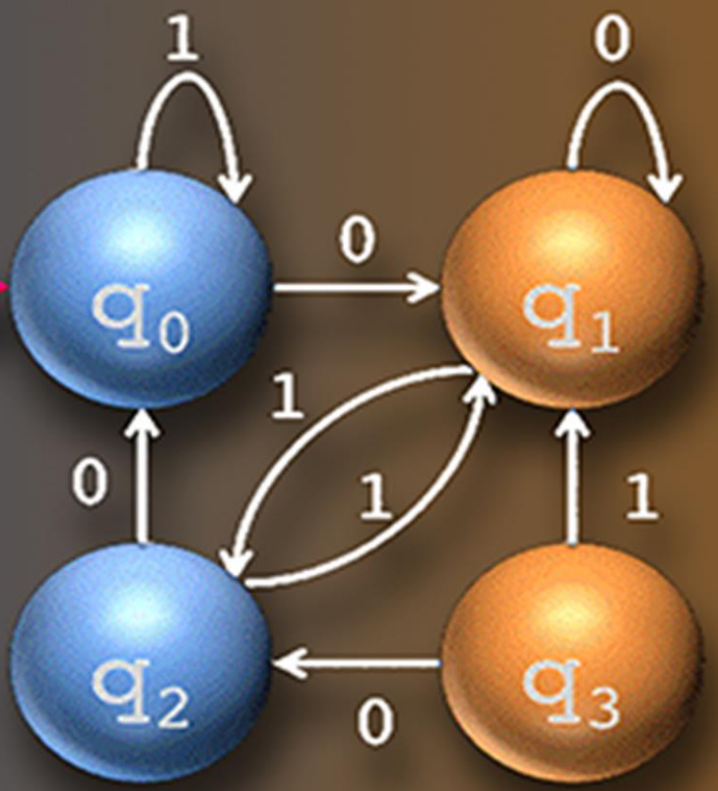
Theory of COMPUTATION



COMPILED BY:
ER. SHIVA RAM DAM

1. Brief Review of Set

1.1 Definition of Set

Sets are represented as a collection of well-defined objects or elements. A set is represented by a capital letter. The number of elements in the finite set is known as the cardinal number of a set.

1.2 Elements of a Set

Let us take an example:

A = {1, 2, 3, 4, 5}

Since a set is usually represented by the capital letter. Thus, A is the set and 1, 2, 3, 4, 5 are the elements of the set or members of the set. The elements that are written in the set can be in any order but cannot be repeated. All the set elements are represented in small letter in case of alphabets. The cardinal number of the set is 5. Some commonly used sets are as follows:

N: Set of all natural numbers
Z: Set of all integers
Q: Set of all rational numbers
R: Set of all real numbers
Z+: Set of all positive integers

1.3 Order of Sets

The order of a set defines the number of elements a set is having. It describes the size of a set. The order of set is also known as the cardinality.

1.4 Representation of Sets

The sets are represented in curly braces, {}. For example, {2,3,4} or {a,b,c} or {Bat, Ball, Wickets}. The elements in the sets are depicted in either the Statement form, Roster Form or Set Builder Form.

1. **Statement Form**
   In statement form, the well-defined descriptions of a member of a set are written and enclosed in the curly brackets.
   For example, the set of even numbers less than 15.
   In statement form, it can be written as {even numbers less than 15}.

2. **Roster Form**
   In Roster form, all the elements of a set are listed.
   For example, the set of natural numbers less than 5.
   Natural Number = 1, 2, 3, 4, 5, 6, 7, 8,…….
   Natural Number less than 5 = 1, 2, 3, 4
   Therefore, the set is N = { 1, 2, 3, 4 }

3. **Set Builder Form**
   The general form is, A = { x : property }
   Example: Write the following sets in set builder form: A={2, 4, 6, 8}
   The set builder form is A = {x: x=2n, n ∈ N and 1 ≤ n ≤ 4}

1.5 Types of Sets

We have several types of sets in math. They are empty set, finite and infinite sets, proper set, equal sets, etc. Let us go through the classification of sets here.

Empty Set	A set which does not contain any element is called an empty set or void set or null set. It is denoted by { } or Ø.
Singleton Set	A set which contains a single element is called a singleton set. Example: There is only one apple in a basket of grapes.
Finite set	A set which consists of a definite number of elements is called a finite set. Example: A set of natural numbers up to 10. A = {1,2,3,4,5,6,7,8,9,10}
Infinite set	A set which is not finite is called an infinite set. Example: A set of all natural numbers. A = {1,2,3,4,5,6,7,8,9……}

Equivalent set	If the number of elements is the same for two different sets, then they are called equivalent sets. The order of sets does not matter here. It is represented as: n(A) = n(B) where A and B are two different sets with the same number of elements. Example: If A = {1,2,3,4} and B = {Red, Blue, Green, Black} In set A, there are four elements and in set B also there are four elements. Therefore, set A and set B are equivalent
Equal sets	The two sets A and B are said to be equal if they have exactly the same elements, the order of elements do not matter. Example: A = {1,2,3,4} and B = {4,3,2,1} Here, A = B
Disjoint Sets	The two sets A and B are said to be disjoint if the set does not contain any common element. Example: Set A = {1,2,3,4} and set B = {5,6,7,8} are disjoint sets, because there is no common element between them.
Subsets	A set 'A' is said to be a subset of B if every element of A is also an element of B, denoted as A ⊆ B. Even the null set is considered to be the subset of another set. In general, a subset is a part of another set. Example: A = {1,2,3} Then {1,2} ⊆ A. Similarly, other subsets of set A are: {1},{2},{3},{1,2},{2,3},{1,3},{1,2,3},{}. If A is not a subset of B, then it is denoted as A⊄B.
Proper Subset	If A ⊆ B and A ≠ B, then A is called the proper subset of B and it can be written as A⊂B. Example: If A = {2,5,7} is a subset of B = {2,5,7} then it is not a proper subset of B = {2,5,7} But, A = {2,5} is a subset of B = {2,5,7} and is a proper subset also.
Superset	Set A is said to be the superset of B if all the elements of set B are the elements of set A. It is represented as A ⊃ B.
Universal Set	For example, if set A = {1, 2, 3, 4} and set B = {1, 3, 4}, then set A is the superset of B. A set which contains all the sets relevant to a certain condition is called the universal set. It is the set of all possible values. Example: If A = {1,2,3} and B {2,3,4,5}, then universal set here will be: U = {1,2,3,4,5}
Power set	The power set is a set which includes all the subsets including the empty set and the original set itself. If set A = {x, y, z} is a set, then all its subsets {x}, {y}, {z}, {x, y}, {y, z}, {x, z}, {x, y, z} and {} are the elements of power set, such as: Power set of A, P(A) = { {x}, {y}, {z}, {x, y}, {y, z}, {x, z}, {x, y, z}, {} }

1.6 Operations on Sets

In set theory, the operations of the sets are carried when two or more sets combine to form a single set under some of the given conditions. The basic operations on sets are:

- Union of sets
- Intersection of sets
- A complement of a set
- Cartesian product of sets.
- Set difference

Union of Sets	If set A and set B are two sets, then A union B is the set that contains all the elements of set A and set B. It is denoted as A ∪ B. Example: Set A = {1,2,3} and B = {4,5,6}, then A union B is: A ∪ B = {1,2,3,4,5,6}

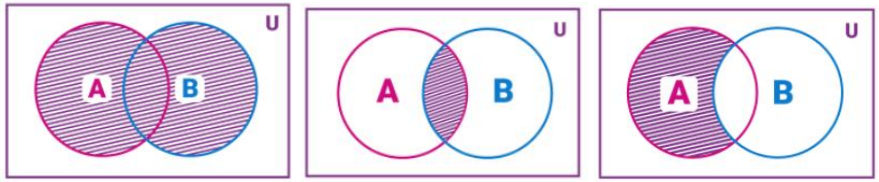
Intersection of Sets	If set A and set B are two sets, then A intersection B is the set that contains only the common elements between set A and set B. It is denoted as A ∩ B. Example: Set A = {1,2,3} and B = {4,5,6}, then A intersection B is: A ∩ B = { } or Ø
Complement of Sets	The complement of any set, say P, is the set of all elements in the universal set that are not in set P. It is denoted by P'. Properties of Complement sets 1. P ∪ P' = U 2. P ∩ P' = Φ 3. Law of double complement : (P')' = P 4. Laws of empty/null set(Φ) and universal set(U), Φ' = U and U' = Φ.
Cartesian Product of sets	If set A and set B are two sets then the cartesian product of set A and set B is a set of all ordered pairs (a,b), such that a is an element of A and b is an element of B. It is denoted by A × B. We can represent it in set-builder form, such as: A × B = {(a, b) : a ∈ A and b ∈ B} Example: set A = {1,2,3} and set B = {4,5}, then; A × B = {(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)} B × A = {(4,1),(4,2),(4,3),(5,1),(5,2),(5,3)}
Difference of Sets	If set A and set B are two sets, then set A difference set B is a set which has elements of A but no elements of B. It is denoted as A – B. Example: A = {1,2,3} and B = {2,3,4} A – B = {1}

1.7 Properties of Sets

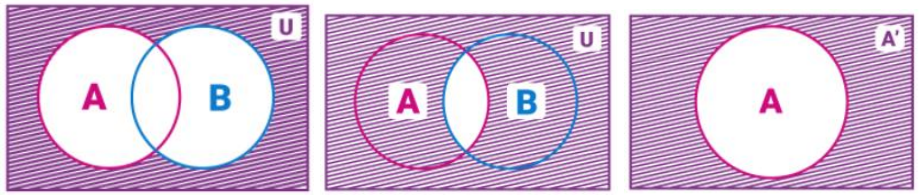
Commutative Property :	• AUB = BUA • A∩B = B∩A
Associative Property :	• A ∪ (B ∪ C) = (A ∪ B) ∪ C • A ∩ (B ∩ C) = (A ∩ B) ∩ C
Distributive Property :	• A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C) • A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C)
De morgan's Law :	• Law of union : (A ∪ B)' = A' ∩ B' • Law of intersection : (A ∩ B)' = A' ∪ B'
Complement Law :	• A ∪ A' = A' ∪ A =U • A ∩ A' = Ø
Idempotent Law And Law of a null and universal set :	For any finite set A • A ∪ A = A • A ∩ A = A • Ø' = U • Ø = U'

1.8 Venn diagram

Venn diagrams are the diagrams that are used to represent the sets, relation between the sets and operation performed on them, in a pictorial way.



AUB A∩B A-B

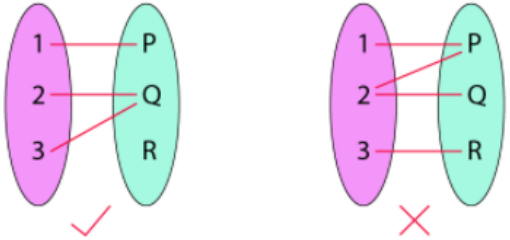


(AUB)' (A∩B)' A'

2. Function and Relation

2.1 Function

A function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output. Let A & B be any two non-empty sets, mapping from A to B will be a function only when every element in set A has one end only one image in set B.
Example:



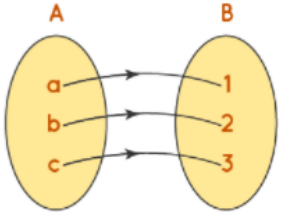
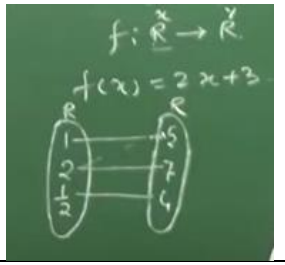
In case of function: for (x, y) , each x has only one y.
In case of relation: for (x, y) , one, some or all x can have more than y.

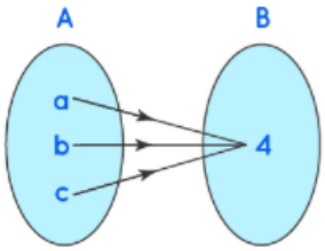
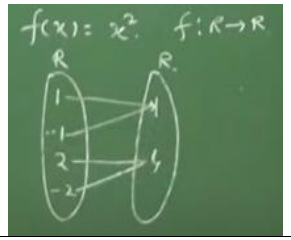
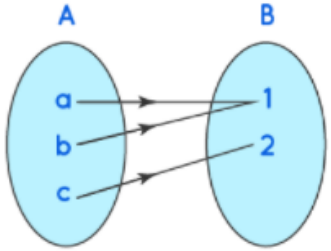
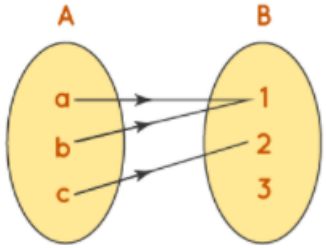
2.2 Domain and Range

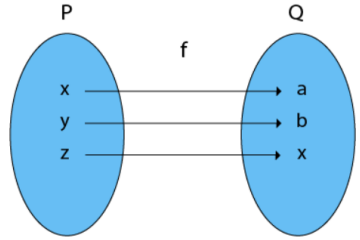
Domain:
For a function, y = f(x), the set of all the values of x is called the domain of the function. It refers to the set of possible input values.

Range:
Range of y = f(x) is a collection of all outputs f(x) corresponding to each real number in the domain. Range is the set of all the values of y. It refers to the set of possible output values.
For example, consider the following relation.
{(2, 3), (4, 5), (6, 7)}
Here Domain = {2, 4, 6}
Range = { 3, 5, 7}

2.3 Types of functions

Types	Details	Mapping
One-to-One Functions (Injective)	A one-to-one function is defined by f: A → B such that every element of set A is connected to a distinct element in set B.	 

Many-to-One function	A many to one function is defined by the function f: A → B, such that more than one element of the set A are connected to the same element in the set B. In a many to one function, more than one element has the same co-domain or image.	 
Onto Function (Surjective)	In an onto function, every codomain element is related to the domain element. For a function defined by f: A → B, such that every element in set B has a pre-image in set A. The onto function is also called a subjective function.	
Into function	The into function is exactly opposite in properties to an onto function. Here there are certain elements in the co-domain that do not have any pre-image. The elements in the set B are excess and are not connected to any elements in the set A.	

One-to-One Onto Function (Bijective)	A function which is both injective (one to - one) and surjective (onto) is called bijective (One-to-One Onto) Function.	

2.4 Relation

A relation R, from a non-empty set P to another non-empty set Q, is a subset of P X Q.
For example, Let P = {a, b, c} and Q = {3, 4} and
Let R = {(a, 3), (a, 4), (b, 3), (b, 4), (c, 3), (c, 4)}
Here R is a subset of A x B. Therefore, R is a relation from P to Q.

2.5 Properties of relation

Let R be a relation on A, and let x,y,z ∈ A.

A relation R is ...	**if ...**
reflexive	xRx
symmetric	xRy implies yRx
transitive	xRy and yRz implies xRz
Irreflexive	xRy implies x≠y
antisymmetric	xRy and yRx implies x=y

2.6 Relation's types

Reflexive relation	Reflexive relation is a relation of elements of a set A such that each element of the set is related to itself. Let A={1,2,3}

	Then R={(1,1), (2,2) , (3,3)} is reflexive relation defined on set A.
Symmetric relation	Relation R is symmetric for all a and b in A if (a,b) ∈ R, then also (b,a) ∈ R Let A={1,2,3} Then R={(1,2)(2,1) , (1,3) (3,1)} is symmetric relation defined on set A.
Transitive relation	Relation R is transitive if for all a,b and c in A(a,b) ∈ R and (b,c) ∈ R, then (a,c) ∈ R Let A={1,2,3} Then ={(1,2), (2,3) , (1,3)} is transitive relation defined on set A.
Equivalence relation	A relation is an Equivalence Relation if it is reflexive, symmetric, and transitive. i.e. relation R={(1,1),(2,2),(3,3),(1,2),(2,1),(2,3),(3,2),(1,3),(3,1)} on set A={1,2,3} is equivalence relation as it is reflexive, symmetric, and transitive.
Partial order relation	Let R be a relation defined on set A, then R is partial order relation if it is reflexive, anti-symmetric and transitive. Let A={1,2,3} Then R={(1,1)(2,2) , (3,3) (1,2), (2,3), (1,3)} is partial order relation defined on set A.
Total order relation	A partial order relation is Total order relation if for all a and b in A , either (a,b) ∈ R or (b,a) ∈ R. Let A={1,2,3} Then R={(1,1)(2,2) , (3,3) (2,1), (2,3), (1,3)} is total order relation defined on set A.

3. Alphabets and Languages

3.1 Alphabets

Theory of computation is entirely based on symbols. These symbols are generally letters and digits.

Alphabets are defined as *a finite set of symbols.*

Examples:

∑ = {0, 1} is an alphabet of binary digits

∑ = {A, B, C, …., Z} is an alphabet.

3.2 String

A string is a finite sequence of symbols selected from some alphabet. It is generally denoted as w. For example, for alphabet ∑ = {0, 1} w = 010101 is a string.

Length of a string is denoted as |w| and is defined as the number of positions for the symbol in the string. For the above example length is 6.

The empty string is the string with zero occurrence of symbols. This string is represented as ϵ or λ.

The set of strings, including the empty string, over an alphabet ∑ is denoted by ∑*.

For ∑ = {0, 1} we have set of strings as ∑* = {ϵ, 0, 1, 01, 10, 00, 11, 10101,…}. and ∑1 = {0, 1} , ∑2 = {00, 01, 10, 11} and so on.

3.3 Power of alphabet

If Σ is an alphabet, the set of all strings can be expressed as a certain length from that alphabet by using exponential notation. The power of an alphabet is denoted by Σk and is the set of strings of length k.

For example,

- Σ ={0,1}
- Σ1= {0,1}
- Σ2= {00,01,10,11}
- Σ3= {000,001,010,011,100,101,110,111}

3.4 Concatenation of strings

Let w1 and w2 be two strings then w1w2 denotes their concatenation w. The concatenation is formed by making a copy of w1 and followed by a copy of w2.
For example, w1 = 001, w2 = 101
then w = w1w2 = 001101

3.5 Kleen closure

If S is a set of words then by S* we mean the set of all finite strings formed by concatenating words from S, where any word may be used as often we like, and where the null string is also included.
S* is the Kleen closure for S. We can think of kleen star (S*) as an operation that makes an infinite language of strings of letters out of an alphabet

For example, for ∑ = {a}
∑* = {ϵ, a, aa, aaa, ….}

3.6 Positive closure

The set of all strings over an alphabet ∑ except the empty string is called positive closure. It is denoted by ∑+.

∑+ =∑1 U ∑2 U ∑3 U ……

For example, for ∑ = {a}
∑+ = {a, aa, aaa, ….}

3.7 Reversal of string

For any string W, its reversal denoted by W^R us a sting spelled backward.
Eg. W=1010
W^R=0101

3.8 Language

A language is a set of string all of which are chosen from some ∑*, where ∑ is a particular alphabet. This means that language L is subset of ∑*.
Given : L={w ∈ {a, b}* : w has odd number of a}
This means : L={ a, ab, aaba, ……..}

The relation between Regular expression and the language they represent is established by a function L, such that if α is any regular expression, then L(α) is language represented by α.

The function L is defined as follows:

1. L(Φ)= Φ and L (a) ={a} for each a∈∑
2. If α and β are regular expressions, then L(α U β) =L(α) U L(β)
3. If α and β are regular expressions, then L(αβ) =L(α). L(β)
4. If α is regular expression, then L(α*) =L(α)*

What language is represented by following RE: L(((aUb)*a))

Here , L(((aUb)*a))	=	L((aUb)*) L(a)	From Rule 3
	=	L((aUb)*) {a}	From Rule 1
	=	L((aUb))* {a}	From Rule 4
	=	(L(a) U L(b))* {a}	From Rule 2
	=	({a} U {b})* {a}	From Rule 1
	=	(a,b)*{a}	
	=	{w∈{a,b}* : w ends with an a}	

3.9 Regular expressions

Regular expression is a formula representing a language in terms of a form that uses three operations : concatenation, union and kleen closure. Regular Expressions are used to denote regular languages. An expression is regular if:

- ϕ is a regular expression for regular language ϕ.
- ε is a regular expression for regular language {ε}.

- If a ∈ Σ (Σ represents the input alphabet), a is regular expression with language {a}.
- If a and b are regular expression, a + b is also a regular expression with language {a,b}.
- If a and b are regular expression, ab (concatenation of a and b) is also regular.
- If a is regular expression, a* (0 or more times a) is also regular.

Regular Expressions	Regular Set
(0 + 10*)	L = { 0, 1, 10, 100, 1000, 10000, … }
(0*10*)	L = {1, 01, 10, 010, 0010, …}
(0 + ε)(1 + ε)	L = {ε, 0, 1, 01}
(a+b)*	Set of strings of a's and b's of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….}
(a+b)*abb	Set of strings of a's and b's ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, ……..}
(11)*	Set consisting of even number of 1's including empty string, So L= {ε, 11, 1111, 111111, …….}
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, ……..}
(aa + ab + ba + bb)*	String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, ..}
(aa)* (bb)*b	L={ w ∈{a,b}* : w has even no. of a followed by odd no. of b} i.e. L= {b, aab, aabbb, aaaab, ……}
(0U1)*00 Or (0+1)*00	L={ w ∈{0, 1}* : w has strings of 0 and 1 ending in 00} L= {0100, 110100, 100 , …….}
0(0U1)*1 or 0(0+1)*1	L={ w ∈{0, 1}* : w has strings of 0 and 1 beginning with 0 and ending with 1} L= {01 ,001, 0011…….}
(((00)*1*) + (01 – 0))*	Set of strings of 0 and 1 with even numbers of 0 i.e. 00,001, 00 , 0001, 1010, ….
0*(10*10*)*10*	Language containing odd no. of 1 i.e. 1,01, 01101, 0111, 111, ….

Given alphabet Σ=(a,b), Write Regular expressions for below:

S.No.	language	Regular Expression
1.	Start with ab	ab(a+b)*
2.	Start with bba	bba(a+b)*
3.	Ends with abb	(a+b)*abb
4.	Contains a substring aab	(a+b)*abb (a+b)*
5.	Start and ends with a	a+a(a+b)*a
6.	Starts and ends with same symbol	a+a(a+b)*a + b + b(a+b)*b
7.	Starts and ends with different symbol	a(a+b)*b+b(a+b)*a
8.	|w|=3	(a+b) (a+b) (a+b) OR $(a+b)^3$
9.	|w|≥3	$(a+b)^3$ (a+b)*

10.	$\|w\| \leq 3$	$\varepsilon + (a+b) + (a+b)^2 + (a+b)^3$ OR $(a+b+\varepsilon)^3$
11.	$\|w\|_a = 2$	b*ab*ab*
12.	$\|w\|_a \geq 2$	(a+b)*a(a+b)* (a+b)*
13.	$\|w\|_a \leq 2$	b*(a+ ε)b*(a+ ε)b* OR b*+b*ab*+b*ab*ab*
14.	3rd symbol from left end is b	$(a+b)^2$ b (a+b)*
15.	28th symbol from right end is a	$(a+b)^*a(a+b)^{27}$
16.	$\|w\| = 0 \pmod 3$ i.e. length of string divided by 3 is 0. 0,3,6,9,……	$((a+b)^3)^*$
17.	$\|w\| = 2 \pmod 3$	$a+b)^2((a+b)^3)^*$
18.	$\|w\|_b = 0 \pmod 2$ i.e. no of b divisible by 2	a* + (a*ba*ba*)*
19.	$\|w\|_a = 1 \pmod 3$	b*ab*(b*ab*ab*ab*)*
20.	$\|w\|_b = 2 \pmod 3$	a*ba*ba*(a*ba*ba*ba*)*

Describe the following sets by Regular expressions.

1.	{101}	101
2.	{abba}	Abba
3.	{01, 10}	01+10
4.	{^ ,ab}	^+ab
5.	{abb, a,b, bba}	abb+ a+ b+ bba
6.	{^, 0,00,000,….}	0*
7.	{1,11,111,….}	11* OR 1^+

End of Chapter 1

2.1 Theory of Computation

Automata theory (aka Theory of Computation) is a theoretical branch of Computer Science and Mathematics, which mainly deals with the logic of computation with respect to simple machines, referred to as automata.

Automata* enables the scientists to understand how machines compute the functions and solve problems. The main motivation behind developing Automata Theory was to develop methods to describe and analyze the dynamic behavior of discrete systems. Automata is originated from the word "Automaton" which is closely related to "Automation".

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.

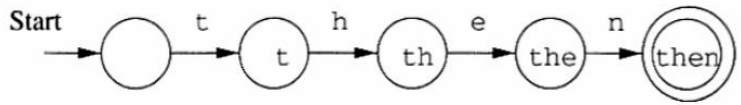
Automata theory

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.



2.2 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:

- Q: finite set of states
- ∑: finite set of the input symbol
- q0: initial state
- F: final state
- 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.

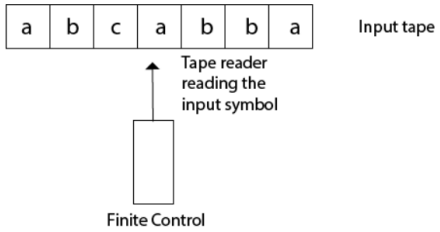


Fig :- Finite automata model

Transition Diagram

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

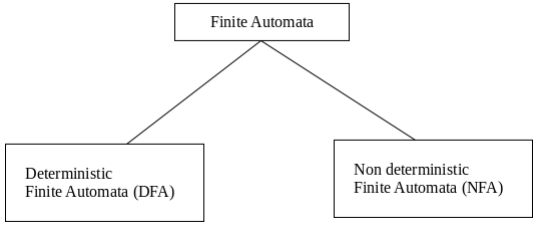
- There is a node for each state in Q, which is represented by the circle.
- There is a directed edge from node q to node p labeled a if δ(q, a) = p.
- In the start state, there is an arrow with no source.
- Accepting states or final states are indicating by a double circle.

Some Notations that are used in the transition diagram:

Types of Automata:

There are two types of finite automata:

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



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.

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:

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

DFA vs NDFA

The following table lists the differences between DFA and NDFA.

DFA	NFA
All transitions are deterministic.	Transitions can be non-deterministic.
Each transition leads to exactly one state.	A transition could lead to a subset of states.
For each state, transitions on all possible symbols(alphabets) should be defined	For each state, not all symbols necessarily have to be defined in the transition function.

Accepts input if the last states in in F	Accepts input if one of the last sates is in F
Sometimes harder to construct because of the numbers of states.	Generally easier thana DFA to construct.
Practical implementation is feasible.	Practical implementation has to be deterministic (so needs conversion to DFA)
Back tracking is allowed in DFA	Back tracking is not allowed in NFA
Requires more memory	Less memory

2.3 DFA

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

Formal Definition of a DFA

A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

- Q is a finite set of states.
- ∑ is a finite set of symbols called the alphabet.
- δ is the transition function where δ: Q × ∑ → Q
- q0 is the initial state from where any input is processed (q0 ∈ Q).
- F is a set of final state/states of Q (F ⊆ Q).

Graphical Representation of a DFA

- A DFA is represented by digraphs called state diagram.
- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example

Let a deterministic finite automaton be →

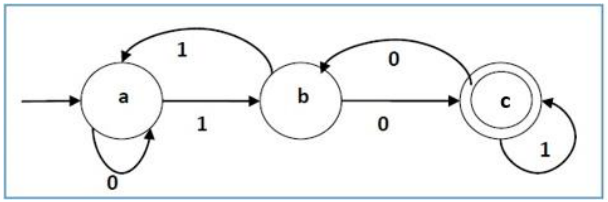
Q = {a, b, c},
∑ = {0, 1},
q_0 = {a},
F = {c}, and

Transition function δ as shown by the following table −

Q	0	1
a	a	b
b	c	a
c	b	c

Or the transition function can be written as:
δ(a, 0) → a
δ(a, 1) → b
δ(b, 0) → c
δ(b, 1) → a
δ(c, 0) → b

Its graphical representation would be as follows –

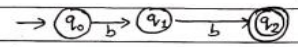


2.4 DFA Numerical Problems:

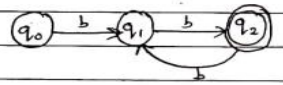
1. Design a DFA that accepts the language given by: L= {w ∈ {a, b}* : w has even numbers of b}

Sol^{n}: Here, L = { abb , bb , bbbb , bab, bba, bbabb , baabaabb, ... }

First let us design for bb



Again design for bbbb



Hence the DFA is

M = { Q, Σ, δ, q_0, F }

where,

Q = { q_0, q_1, q_2 }

Σ = { a, b }

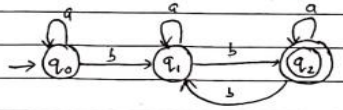
q_0 = { q_0 }

F = { q_2 }

δ :

Q	a	b
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_1

State diagram is :



2. Design a DFA that accepts the language given by: L= {w ∈ {1, 0}* : w ends with 100 }

Here,

L = { 100, 1100, 0100, 10100, 010100, ... }

Let M = { Q, Σ, δ, q_0, F } where

Q = { q_0, q_1, q_2, q_3 }

Σ = { 0, 1 }

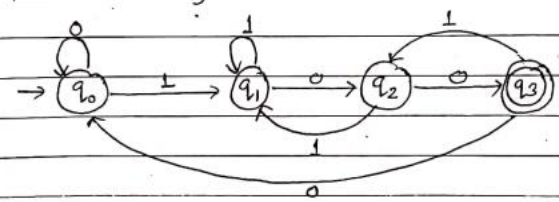
q_0 = { q_0 }

F = { q_3 }

δ :

Q	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_3	q_1
q_3	q_0	q_1

Hence, the state diagram is:



Verification:

10100 → Verified ends at q_3, accepted

10101 → ends at q_1, rejected

3. Construct a DFA starting with aba, ∑ = a, b.

Here,

L = { aba , abaa , abab, ... }

Let M = { Q, Σ, δ, q_0, F }

where,

Q = { q_0, q_1, q_2, q_3, q_ϕ }

Σ = { a, b }

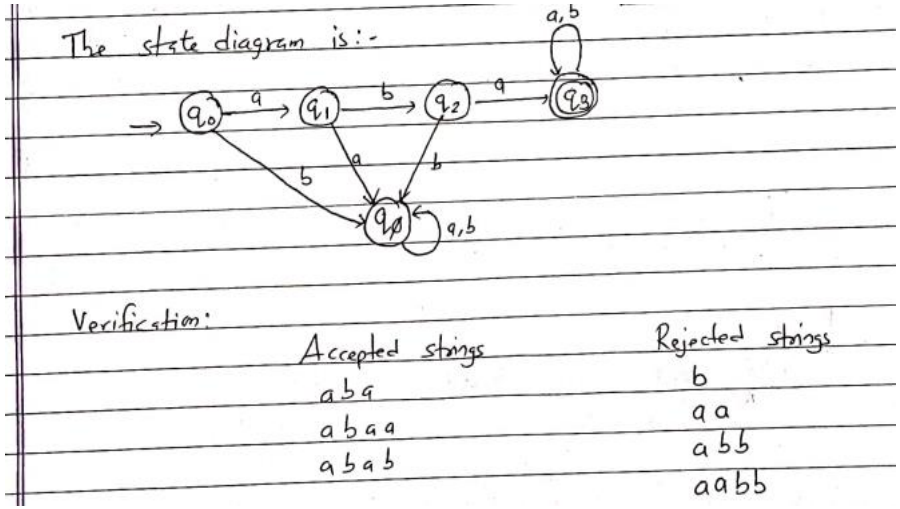
q_0 = { q_0 }

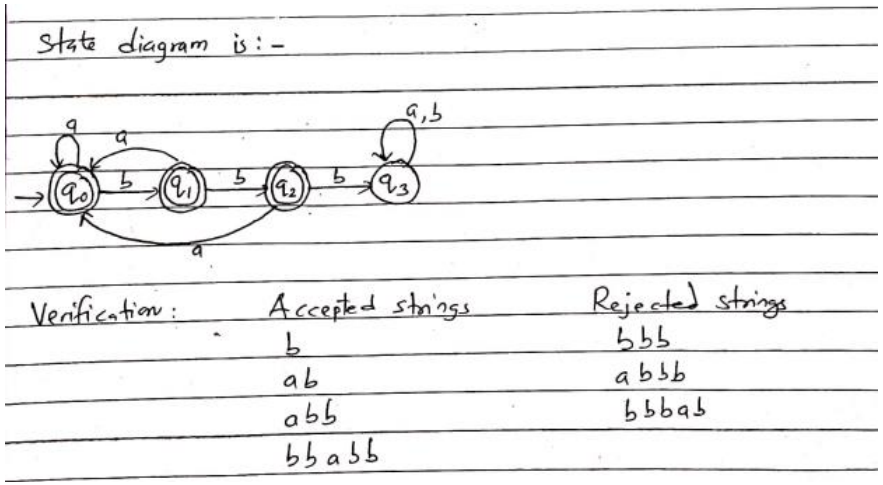
F = { q_3 }

and

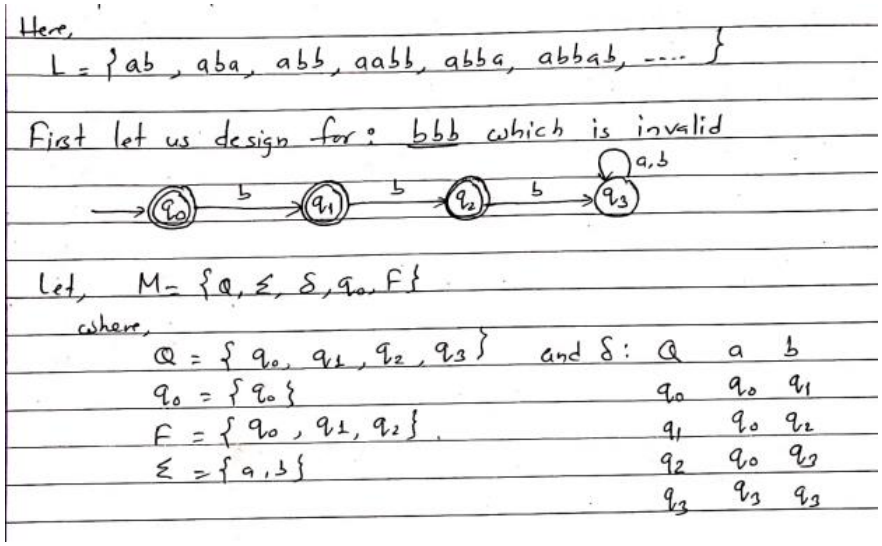
δ :

Q	a	b
q_0	q_1	q_ϕ
q_1	q_ϕ	q_2
q_2	q_3	q_ϕ
q_3	q_3	q_3
q_ϕ	q_ϕ	q_ϕ

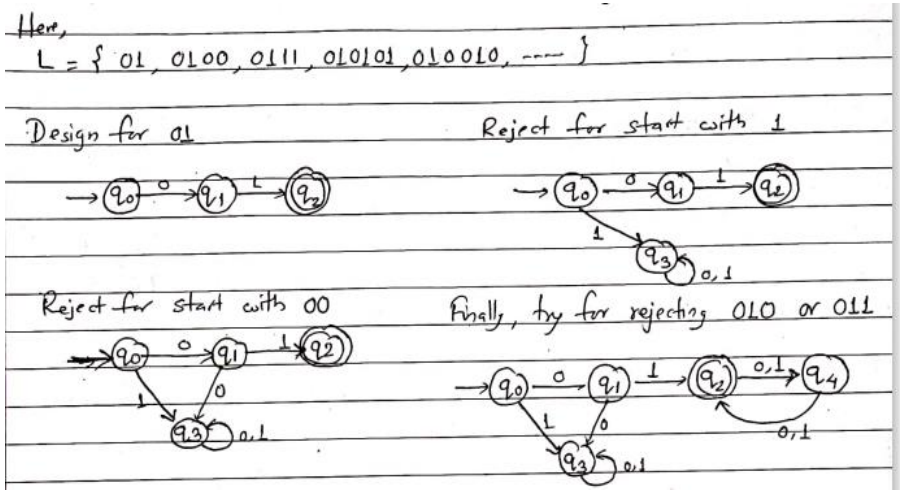




4. Design a DFA that accepts the language given by: L= {w ∈ {a, b}* : w does not contain three consecutive b }



5. Design a DFA that accepts the language given by: L= {w ∈ {0, 1}* : w starts with 01 having even length}



Hence, DFA is:

$M = \{Q, \Sigma, \delta, q_0, F\}$

where,

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

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

$q_0 = \{q_0\}$

$F = \{q_2\}$

and

δ:

Q	0	1
q_0	q_1	q_3
q_1	q_3	q_2
q_2	q_4	q_4
q_3	q_3	q_3
q_4	q_2	q_2

Verification:

$0111 \Rightarrow \delta(q_0, 0111)$
$\vdash (q_1, 111)$
$\vdash (q_2, 11)$
$\vdash (q_4, 1)$
$\vdash (q_2, \epsilon)$

Since q_2 lies in final state, DFA accepts

$010 \Rightarrow \delta(q_0, 010)$
$\vdash (q_1, 10)$
$\vdash (q_2, 0)$
$\vdash (q_4, \epsilon)$

Since q_4 is not a final state, DFA rejects

6. Design a DFA that accepts the language given by: L= {w ∈ {0, 1}* : w starts with 01 }

Here,

$L = \{01, 0100, 010, 0111, 010111 \ldots\}$

The DFA is:

$M = \{Q, \Sigma, \delta, q_0, F\}$

where,

$Q = \{$

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

$q_0 = q_0$

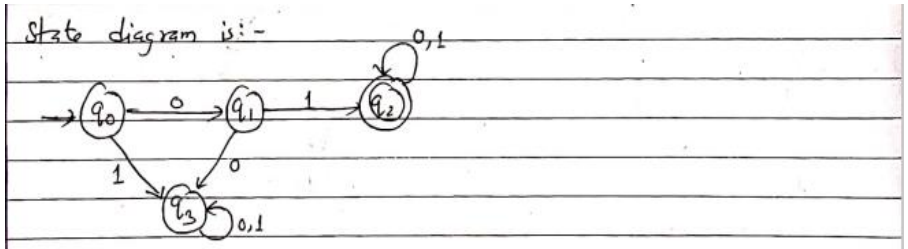
$F = \{q_2\}$

and

δ:

Q	0	1
q_0	q_1	q_3
q_1	q_3	q_2
q_2	q_2	q_2
q_3	q_3	q_3

State diagram is:-



Verification:

Accepted string	Rejected string
0101	001
$\delta(q_0, 0101)$	$\delta(q_0, 001)$
$\vdash (q_1, 101)$	$\vdash (q_1, 01)$
$\vdash (q_1, 01)$	$\vdash (q_3, 1)$
$\vdash (q_2, 1)$	$\vdash (q_3, \epsilon)$
$\vdash (q_2, \epsilon)$	
Since, q_2 is final state, DFA accepts	Since, q_3 is not final state, DFA rejects

Tutorial:

Page no: 29 : Example no: 2.1 – 2.11

2.5 NDFA

n NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. As it has finite number of states, the machine is called Non-deterministic Finite Machine or Non-deterministic Finite Automaton.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where –

- **Q** is a finite set of states.
- **∑** is a finite set of symbols called the alphabets.
- **δ** is the transition function where δ: Q × ∑ → 2^Q
  (Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- **q_0** is the initial state from where any input is processed (q_0 ∈ Q).
- **F** is a set of final state/states of Q (F ⊆ Q).

Graphical Representation of an NDFA: (same as DFA)

- An NDFA is represented by digraphs called state diagram.
- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example

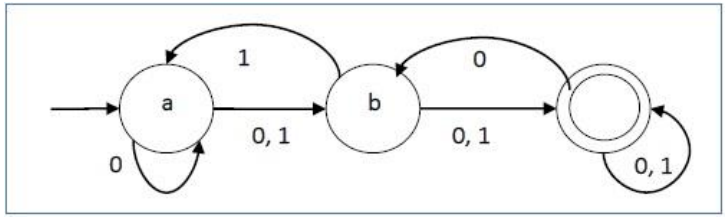
Let a non-deterministic finite automaton be →

Q = {a, b, c}
∑ = {0, 1}
q_0 = {a}
F = {c}

The transition function δ as shown below –

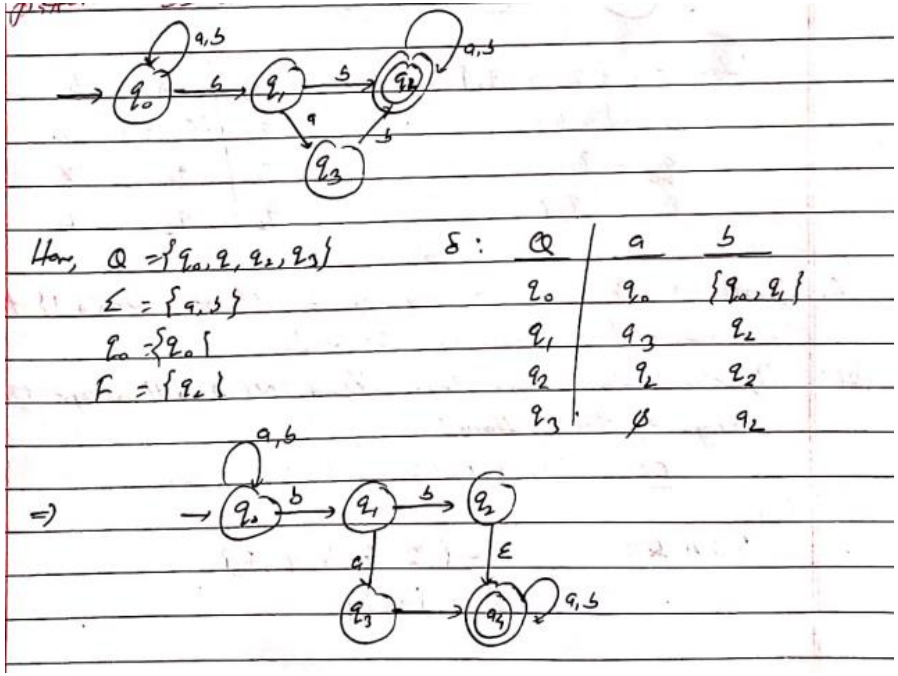
Present State	Next State for Input 0	Next State for Input 1
a	a, b	b
b	c	a, c
c	b, c	c

Its graphical representation would be as follows –

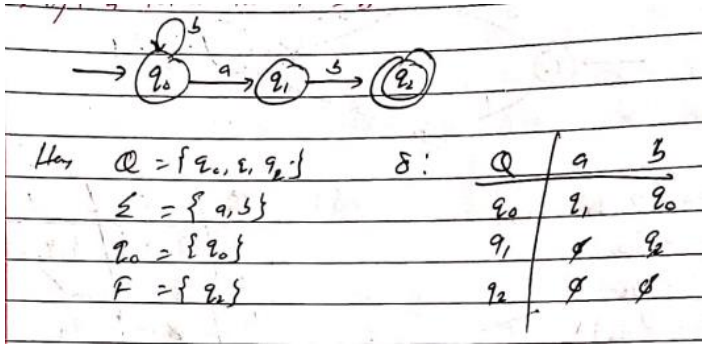


2.6 NFA Numerical Problems

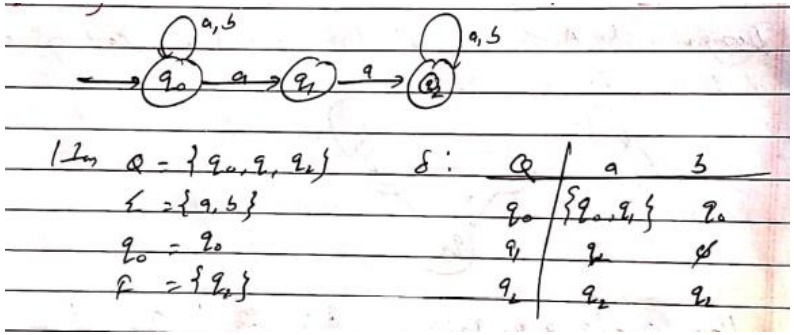
1. **Design a NFA that accepts the set of strings containing occurrence of pattern bb or bab.**



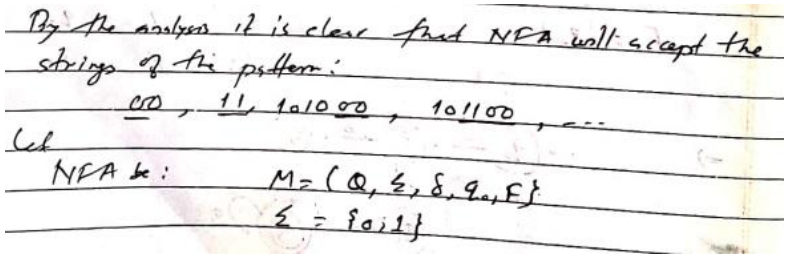
2. Design a NFA for R= b*ab

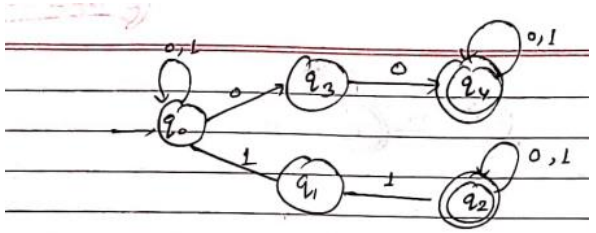


3. Design a NFA over ∑ = {a, b} that accepts strings having aa as substring.

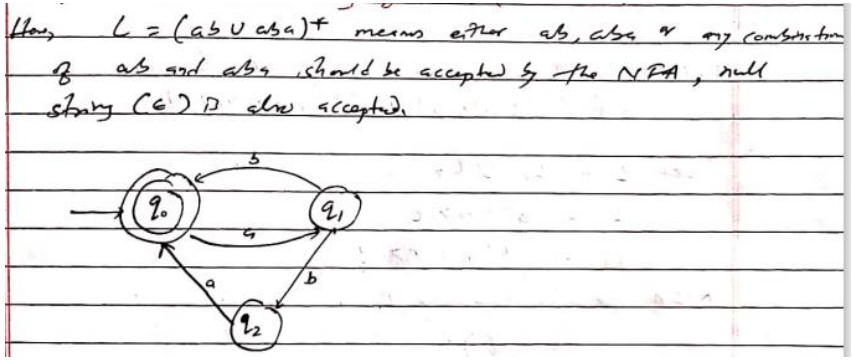


4. Design a NFA for the language L=all strings over {0, 1} that have at least two consecutive 0s and 1s.

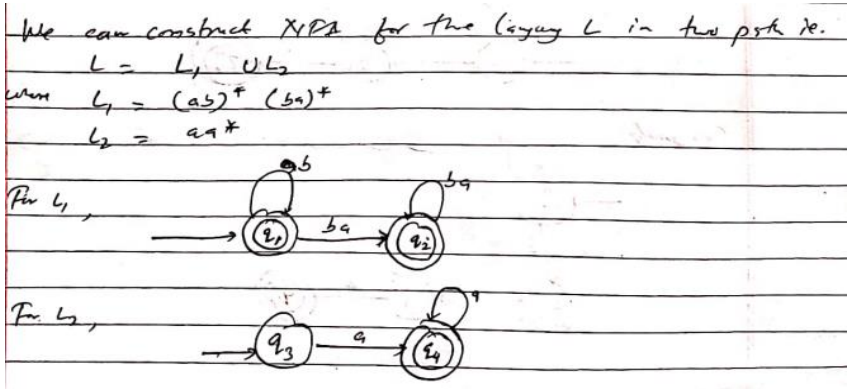


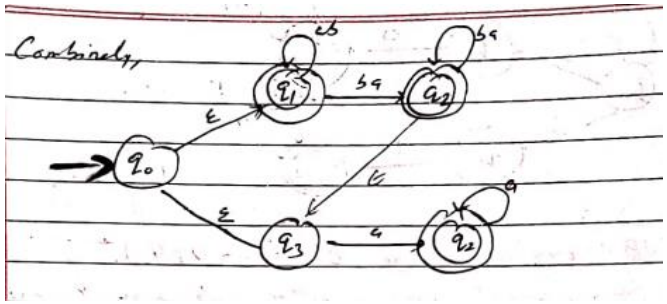


5. Design a NFA for the language L= (ab U aba)*

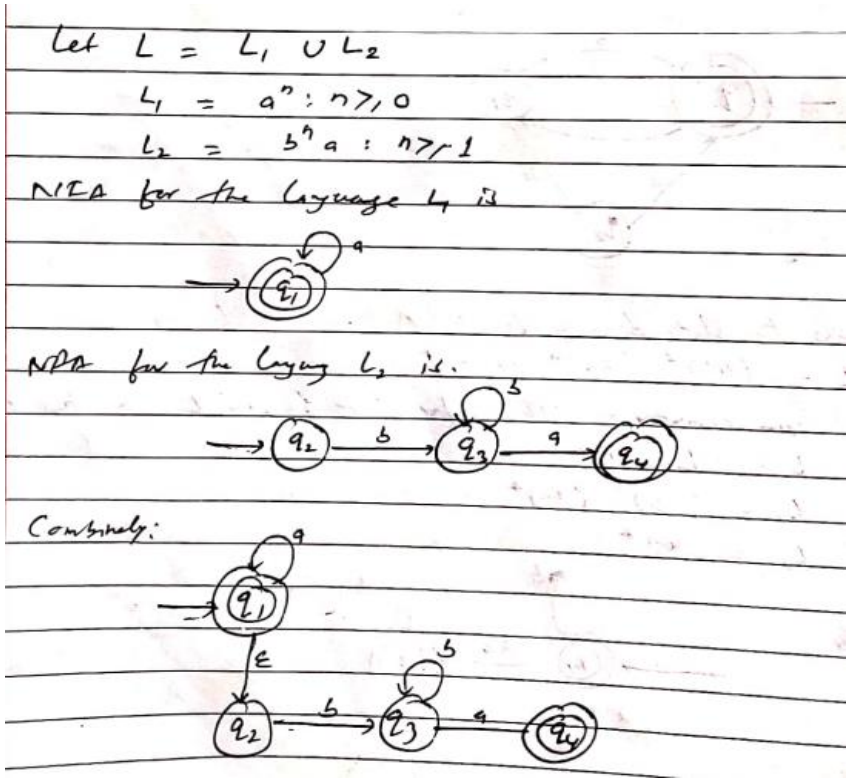


6. Draw the state diagram for NFA accepting the language L= (ab)*(ba)* U aa*





7. Find NFA with four state for the language L= { (a^n : n ≥ 0) U ($b^n a$: n ≥1)}



2.7 Equivalence of DFA and NFA

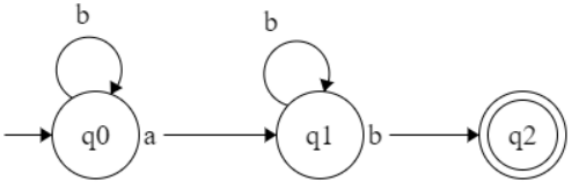
In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol. However, for each NFA, there is an equivalent DFA.

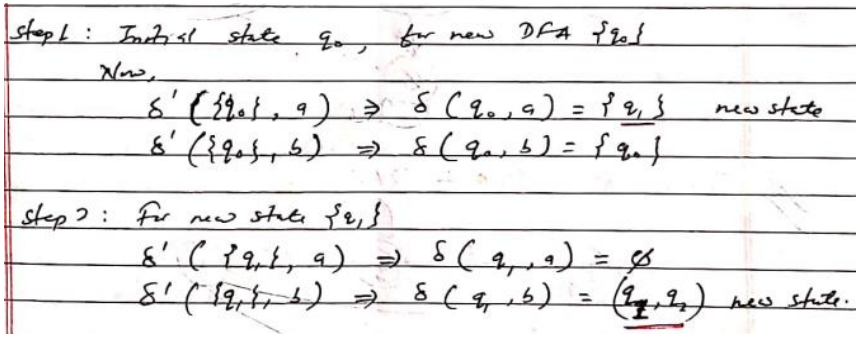
Steps for converting NFA to DFA:

1. Observe all the transitions from starting state q0 for every symbol in alphabet (∑).
2. For new states from step 1, check all transitions for every ∑.
3. Repeat step 2 till new states are appeared.

2.8 NFA to DFA Numerical Problems

1. Convert below NFA to DFA.





Step 3: For new state $\{q_1, q_2\}$

$\delta'(\{q_1, q_2\}, a) \Rightarrow \delta(q_1, a) \cup \delta(q_2, a)$

$\Rightarrow \phi \cup \phi$

$\Rightarrow \phi$

$\delta'(\{q_1, q_2\}, b) \Rightarrow \delta(q_1, b) \cup \delta(q_2, b)$

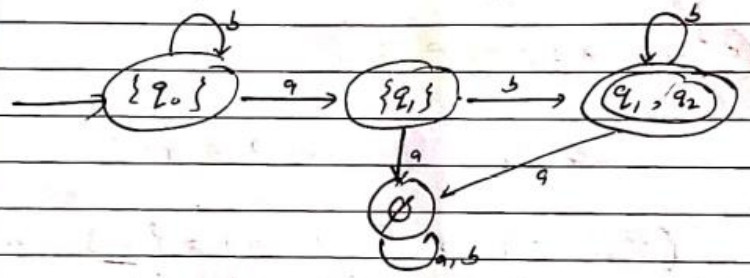
$\Rightarrow \{q_1, q_2\} \cup \phi$

$= \{q_1, q_2\}$

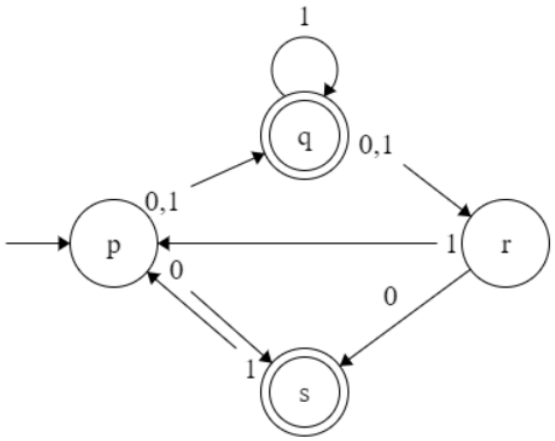
Now, transition table is:

Q	a	b
$\{q_0\}$	$\{q_1\}$	$\{q_0\}$
$\{q_1\}$	ϕ	$\{q_1, q_2\}$
$\{q_1, q_2\}$	ϕ	$\{q_1, q_2\}$

Here, Equivalent DPA is:-



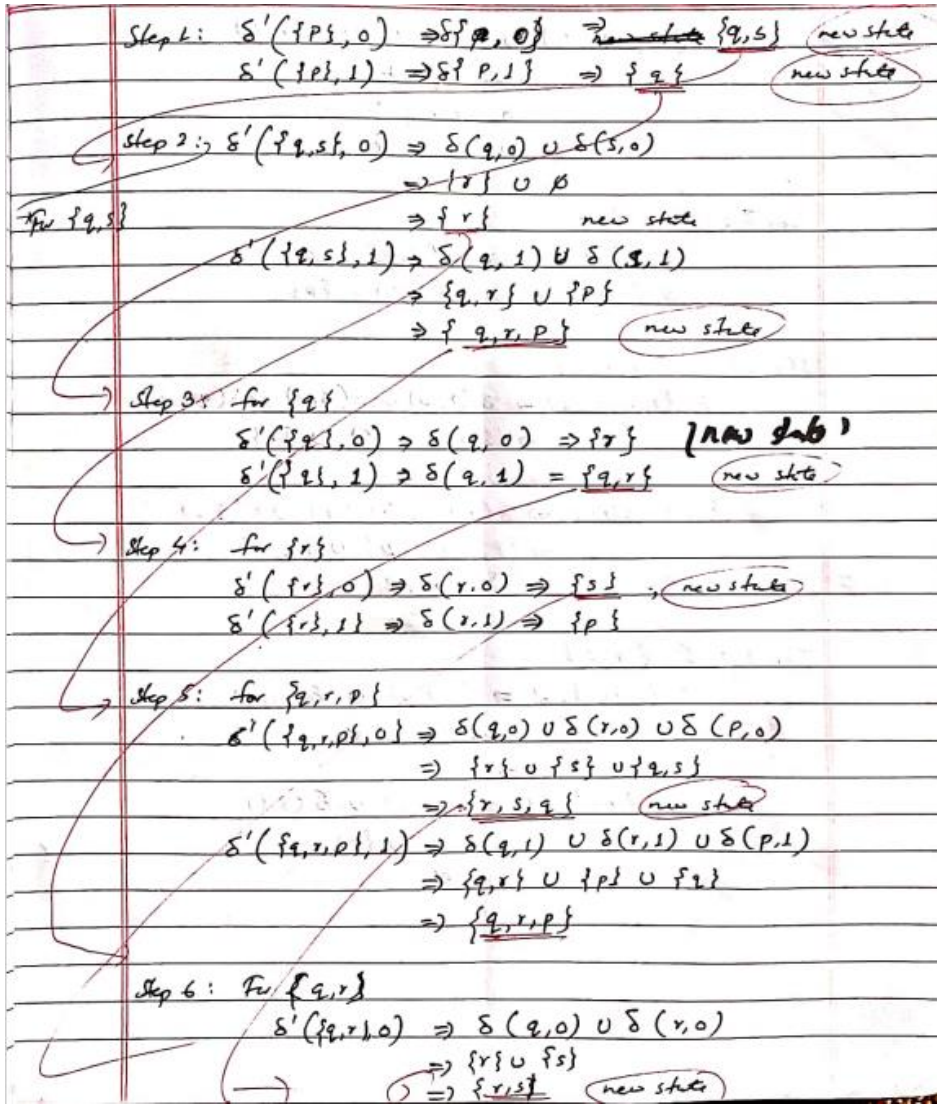
2. Transform below NFA to DFA:

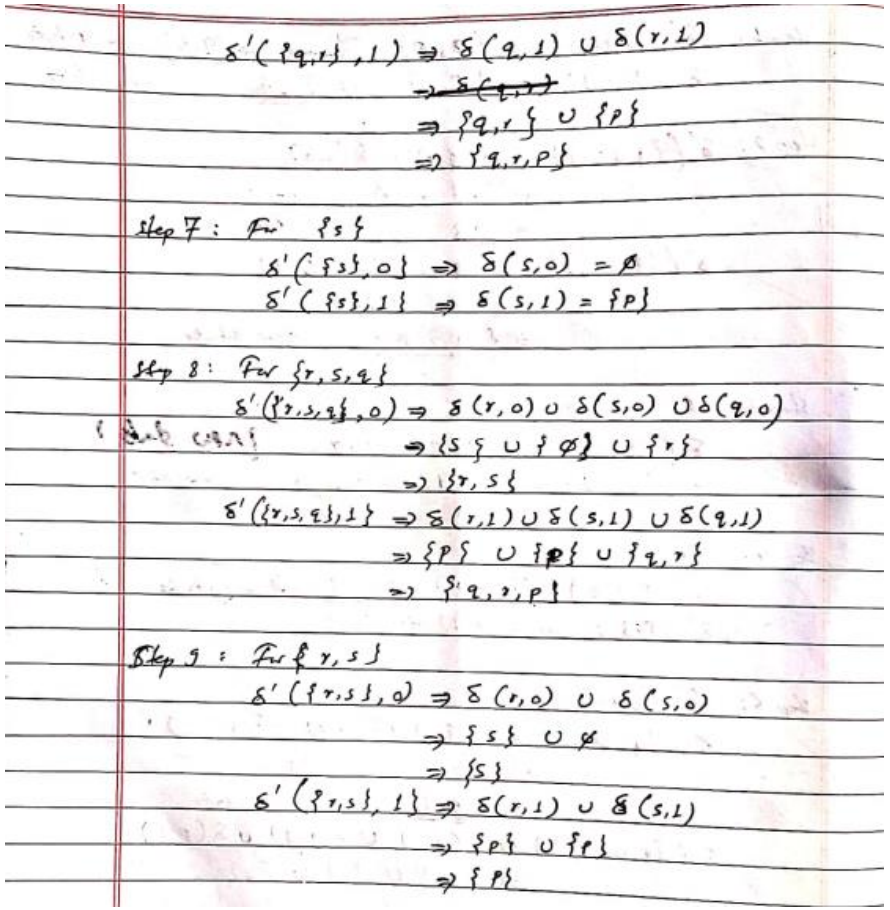


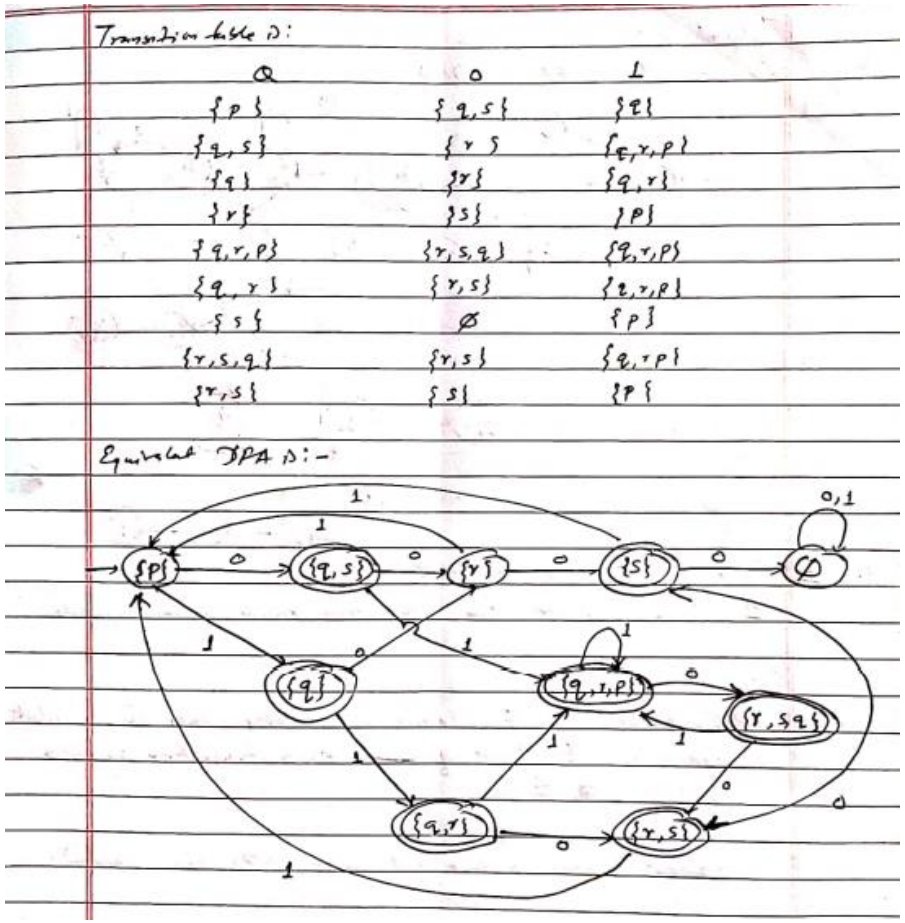
Here, Initial state of NFA is p

For DFA, initial state is {p}

Now,

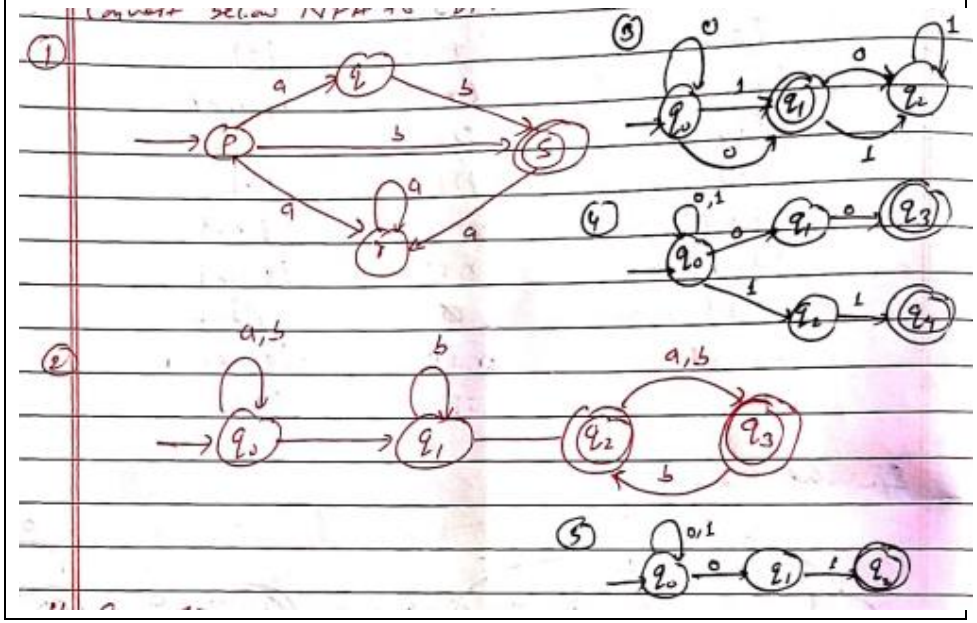

Step 1: δ'({p},0) ⇒ δ{p,0} ⇒ {q,s} new state
δ'({p},1) ⇒ δ{p,1} ⇒ {q} new state
Step 2: δ'({q,s},0) ⇒ δ(q,0) ∪ δ(s,0)
⇒ {r} ∪ ∅
For {q,s} ⇒ {r} new state
δ'({q,s},1) ⇒ δ(q,1) ∪ δ(s,1)
⇒ {q,r} ∪ {p}
⇒ {q,r,p} new state
Step 3: for {q}
δ'({q},0) ⇒ δ(q,0) ⇒ {r} [new state]
δ'({q},1) ⇒ δ(q,1) = {q,r} new state
Step 4: for {r}
δ'({r},0) ⇒ δ(r,0) ⇒ {s} new state
δ'({r},1) ⇒ δ(r,1) ⇒ {p}
Step 5: for {q,r,p}
δ'({q,r,p},0) ⇒ δ(q,0) ∪ δ(r,0) ∪ δ(p,0)
⇒ {r} ∪ {s} ∪ {q,s}
⇒ {r,s,q} new state
δ'({q,r,p},1) ⇒ δ(q,1) ∪ δ(r,1) ∪ δ(p,1)
⇒ {q,r} ∪ {p} ∪ {q}
⇒ {q,r,p}
Step 6: For {q,r}
δ'({q,r},0) ⇒ δ(q,0) ∪ δ(r,0)
⇒ {r} ∪ {s}
⇒ {r,s} new state


δ'({q,r},1) ⇒ δ(q,1) ∪ δ(r,1)
⇒ {q,r} ∪ {p}
⇒ {q,r,p}
Step 7: For {s}
δ'({s},0) ⇒ δ(s,0) = ∅
δ'({s},1) ⇒ δ(s,1) = {p}
Step 8: For {r,s,q}
δ'({r,s,q},0) ⇒ δ(r,0) ∪ δ(s,0) ∪ δ(q,0)
⇒ {s} ∪ {∅} ∪ {r}
⇒ {r,s}
δ'({r,s,q},1) ⇒ δ(r,1) ∪ δ(s,1) ∪ δ(q,1)
⇒ {p} ∪ {p} ∪ {q,r}
⇒ {q,r,p}
Step 9: For {r,s}
δ'({r,s},0) ⇒ δ(r,0) ∪ δ(s,0)
⇒ {s} ∪ ∅
⇒ {s}
δ'({r,s},1) ⇒ δ(r,1) ∪ δ(s,1)
⇒ {p} ∪ {p}
⇒ {p}



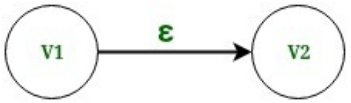
Tutorial:

Convert below NFA to DFA:



2.9 Ɛ -NFA

The NFA with epsilon-transition is a finite state machine in which the transition from one state to another state is allowed without any input symbol i.e. empty string ε. Here in the figure below, State V1 and v2 have an epsilon move.



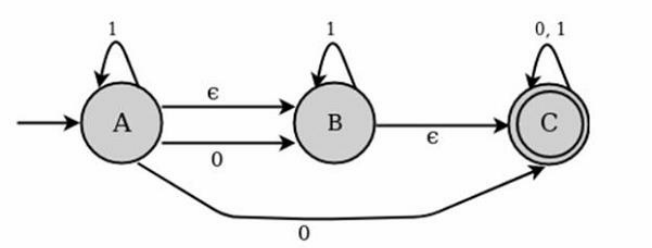
Epsilon (ε) - closure

Epsilon closure for a given state X is a set of states which can be reached from the states X with only (null) or ε moves including the state X itself.

In other words, ε -closure for a state can be obtained by union operation of the ε -closure of the states which can be reached from X with a single ε move in a recursive manner.

Example

Consider the following figure of NFA with ε move −



The transition state table for the above NFA is as follows −

State	0	1	epsilon
A	B, C	A	B
B	-	B	C
C	C	C	-

For the above example, ε closure are as follows −

E closure(A) : {A, B,C}

E closure(B) : {B,C}

E closure(C) : {C}

Converting E-NFA to DFA

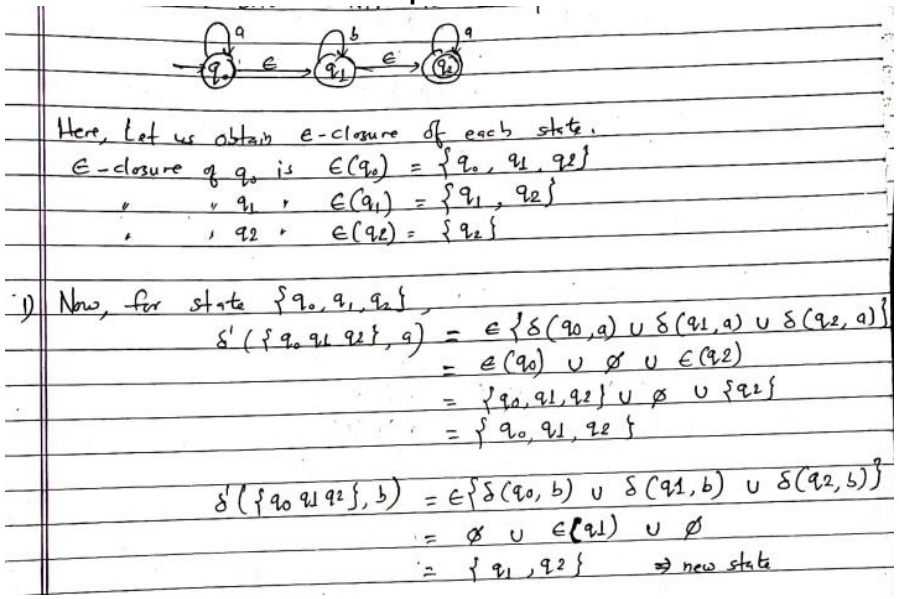
Steps:

1. Find out the ε-closure of starting state, and calculate union of ε-closure of each transition foe every symbol of alphabet (∑).
2. Repeat same process till new states are approved.

Here, ε -closure of state q can be obtained by following all transitions out of q the are labelled ε.

Numerical:

1. **Convert the below ε-NFA to its equivalent DFA.**



2) For state $\{q_1, q_2\}$

$\delta'(\{q_1, q_2\}, a) = \in\{\delta(q_1, a) \cup \delta(q_2, a)\}$
$= \emptyset \cup \in(q_2)$
$= \{q_2\}$ ⇒ new state

$\delta'(\{q_1, q_2\}, b) = \in\{\delta(q_1, b) \cup \delta(q_2, b)\}$
$= \in(q_1) \cup \emptyset$
$= \{q_1, q_2\}$

3) For state $\{q_2\}$:

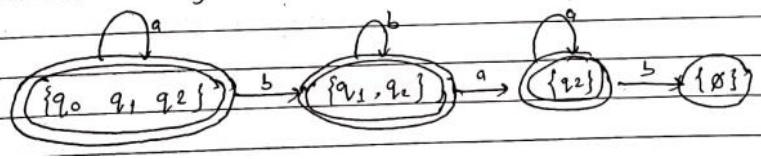
$\delta'(\{q_2\}, a) = \in\{\delta(q_2, a)\}$
$= \in(q_2)$
$= \{q_2\}$

$\delta'(\{q_2\}, b) = \in\{\delta(q_2, b)\}$
$= \emptyset$

Hence, transition table is :-

Q	a	b
$\{q_0 q_1 q_2\}$	$\{q_0 q_1 q_2\}$	$\{q_1, q_2\}$
$\{q_1 q_2\}$	$\{q_2\}$	$\{q_1 q_2\}$
$\{q_3\}$	$\{q_2\}$	$\emptyset$

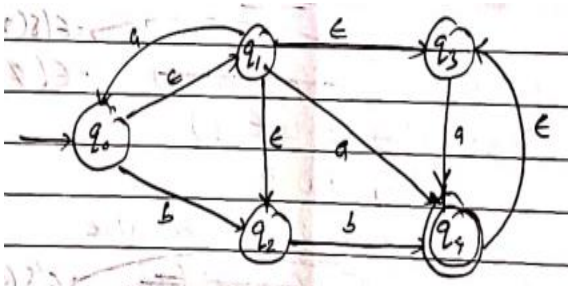
The state diagram of DFA is :-



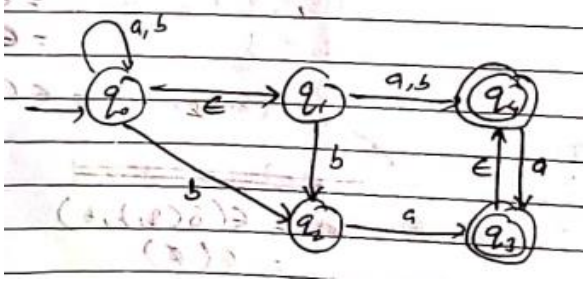
Tutorial:

Convert below ε-NFA to its equivalent DFA.

1)



2)

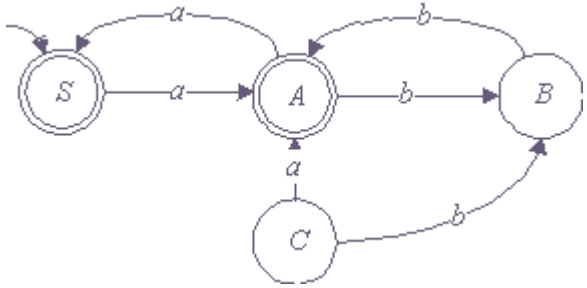


2.10 State Minimization (DFA Minimization)

DFA minimization stands for converting a given DFA to its equivalent DFA with minimum number of states. Minimization of DFA thus reduces the number of states from given FA. Thus, we get the FSM (finite state machine) with redundant states after minimizing the FSM.
In this process we minimize the given DFA by removing unreachable states and by merging similar equivalent states.

Inaccessible states:
All the states which can never be reached from initial state are inaccessible/unreachable states. In contrast, Dead states are the states from which no final state is reachable.

For eg:



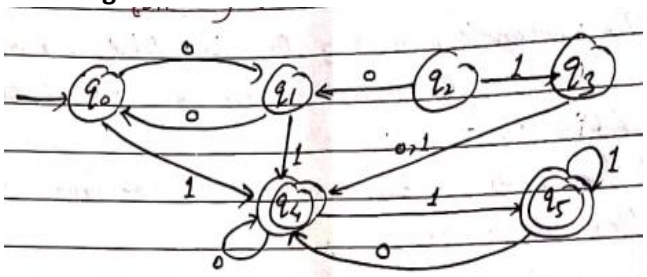
Here, state C is never reached from the initial state S. So state C is unreachable state.

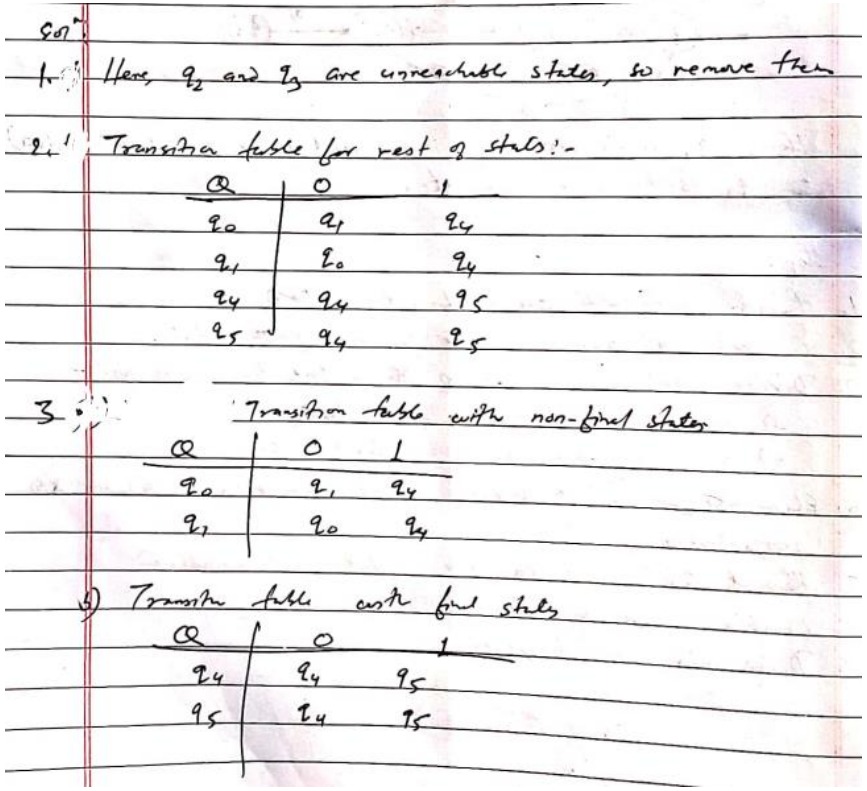
State minimization steps:

1. Remove unreachable states from given DFA.
2. Construct a transition table for rest of the states.
3. Divide the transition table of step 2 into two parts:
   a) Rows that start with final states.
   b) Rows that start with non-final states.
4. Eliminate one of the equivalent/similar states from table : 3a and 3b respectively.
5. Repeat step 4 until equivalent states are appeared.
6. Combine/merge updated tables.
7. Draw minimized state diagram.

2.11 Numerical

1. **Minimize following DFA.**





4. Eliminate equivalent/similar rows in 5th table q_3 and q_5 so, we have. 3(b) as: $(q_4 = q_5)$.

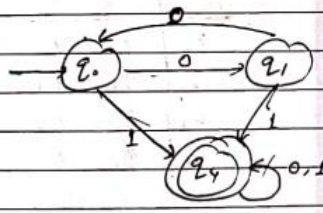
Q	0	1
q_3	q_4	q_4

5. Merge updated table: we get.

Q	0	1
q_0	q_1	q_4
q_1	q_0	q_4
q_4	q_4	q_4

$\therefore (q_4 = q_5)$

6. Final minimized DFA D:-

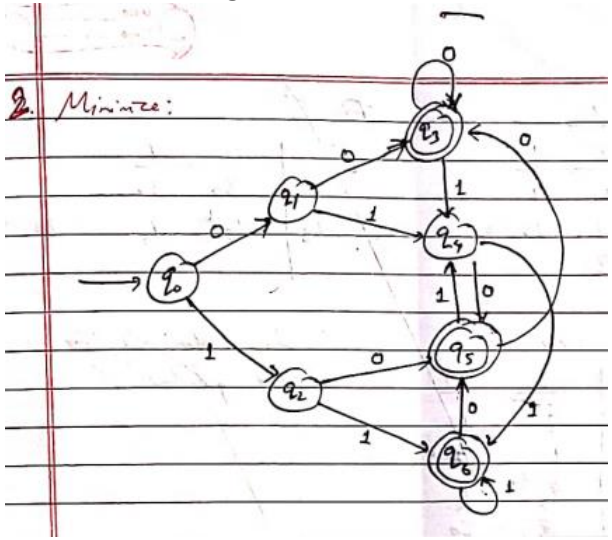


Test: 00100 1

$\Rightarrow \delta(q_0, 00100 1)$
$\vdash (q_1, 0100 1)$
$\vdash (q_0, 100 1)$
$\vdash (q_4, 00 1)$
$\vdash (q_4, 01)$
$\vdash (q_4, 1)$
$\vdash (q_4, \epsilon)$

$\therefore Q_4$ is in final state, so DFA accepted.

2. Minimize following DFA



Solution:

1. No unreachable states.
2. Transition table:

Q	0	1
q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_5	q_6
q_3	q_3	q_4
q_4	q_5	q_6
q_5	q_3	q_4
q_6	q_5	q_6

3. Splitting into transition tables of Non-final and final states,

a) Non-final state Transition table

Q	0	1
q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_5	q_6
q_4	q_5	q_6

b) Final state Transition table

Q	0	1
q_3	q_3	q_4
q_5	q_3	q_4
q_6	q_5	q_6

4. Eliminating one of the equivalent rows.

a)

Q	0	1
q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_5	q_6
~~q_4~~	~~q_5~~	~~q_6~~

Here, Now, $q_2 = q_4$

⇓

Q	0	1	
q_0	q_1	q_2	
q_1	q_3	q_2	$\because q_4 = q_2$
q_2	q_3	q_6	$\because q_5 = q_3$

b)

Q	0	1
q_3	q_3	q_4
~~q_5~~	~~q_3~~	~~q_4~~
q_6	q_5	q_6

Here, Now, $q_3 = q_5$

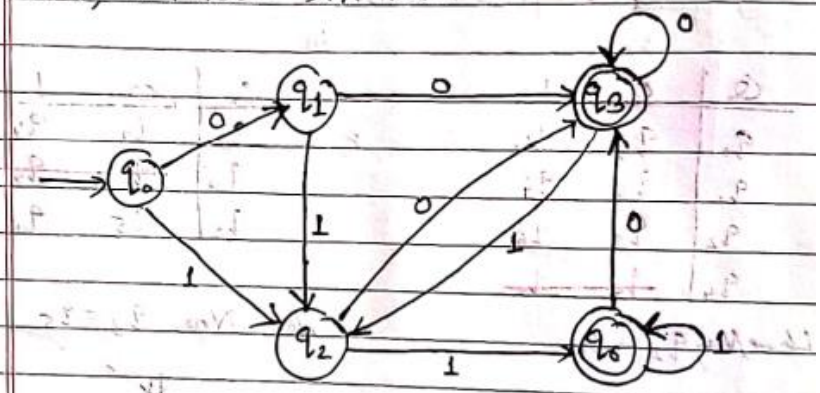
⇓

Q	0	1	
q_3	q_3	q_2	$\because q_4 = q_2$
q_6	q_3	q_6	$\because q_5 = q_3$

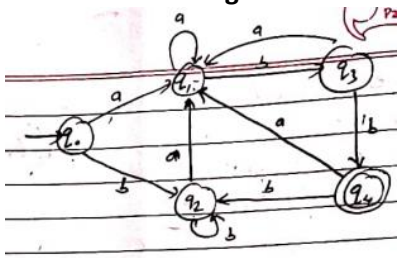
5. Merging the tables,

Q	0	1
q_0	q_1	q_2
q_1	q_3	q_2
q_2	q_3	q_6
q_3	q_3	q_2
q_6	q_3	q_6

6. Hence, minimized DFA is:-



3. Minimize following DFA.



Solution:

1. No unreachable states

2. Transition table:

Q	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
q_4	q_1	q_2

3. a) Non-final

Q	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4

b) Final

Q	a	b
q_4	q_1	q_2

4. Eliminate equivalent rows $q_0 \equiv q_2$

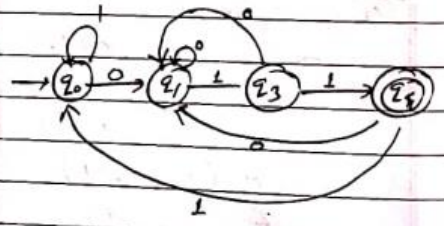
Q	a	b
q_0	q_1	q_0

5. Merge table:

Q	a	b
q_0	q_1	q_0
q_1	q_1	q_3
q_3	q_1	q_4
q_4	q_1	q_0

$\therefore q_0 = q_2$

6. State diagram:

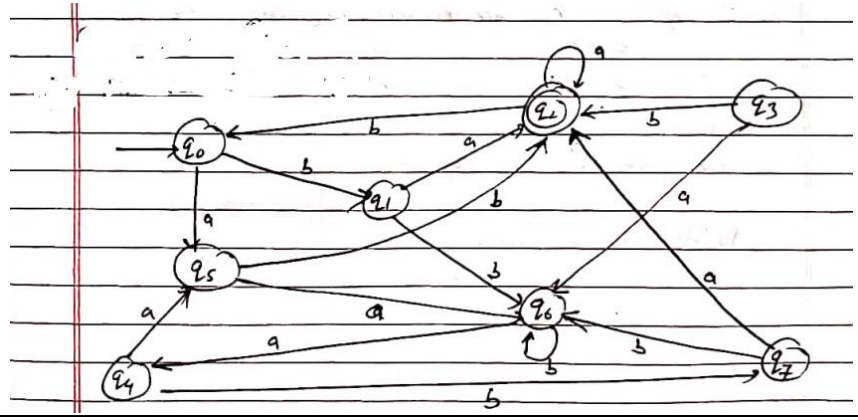


Tutorial:

1. **Minimize following DFA by using State Minimization method, where: → represents initial state and * represents final state.**

δ/Σ	0	1
→ q_0	q_1	q_2
* q_1	q_1	q_3
q_2	q_2	q_2
* q_3	q_5	q_2
* q_4	q_4	q_2
* q_5	q_4	q_2
q_6	q_5	q_6
q_7	q_5	q_6

2. **Minimize below DFA:**



2.12 Regular Expressions and Regular Language

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, ...}

Definition of Regular Expression:

The set of regular expression is defined by the following rules"

1. Every letter of ∑ can be made into a regular expression, null string, ε itself is a regular expression.
2. If r1 and r2 are regular expression, then:
   i) (r1)
   ii) r1r2
   iii) r1 + r2
   iv) r1*
   v) r1+ are also regular expression.
3. Nothing else is regular expression.

Regular Language:

A language is regular if it can be expressed in terms of regular expression.

Regular Expression	Regular language
a	{a}
B	{b}
a+b	{a,b}
a··b	{ab}
a∗a∗	{єє,a,aa,aaa,aaaa,.....}
b+b+	{b,bb,bbb,bbbb,.....}

2.13 Inductive Steps

There are four parts to the inductive step, one for each of three operators and one for the introduction of parentheses.

1. If r1 and r2 are regular expressions, then r1 +r2 is a regular expression denoting the union of L(r1) and L(r2).
   i.e. L(r1+r2)= L(r1) U L(r2)
2. If r1 and r2 are regular expressions, then r1 r2 is a regular expression denoting the concatenation of L(r1) and L(r2).
   i.e. L(r1r2)= L(r1) . L(r2)
3. If r is a regular expressions, then r* is a regular expression denoting the closure of L(r).
   i.e. L(r*)= (L(r))*
4. If r is a regular expressions, then (r) , a parenthesized r, is also a regular expression denoting the same language as r.
   i.e. L((r))= L(r)

2.14 Application of Regular language

1. Validation:	Determines that a string compiles a set of formatting constraints like email validation, password validation, etc.
2. Search and Selection:	Identifying a subset of items from larger set on the basis of a pattern match
3. Tokenization:	Converting a sequence of characters into words, tokens for later interpretation.

2.15 Algebraic laws for RE

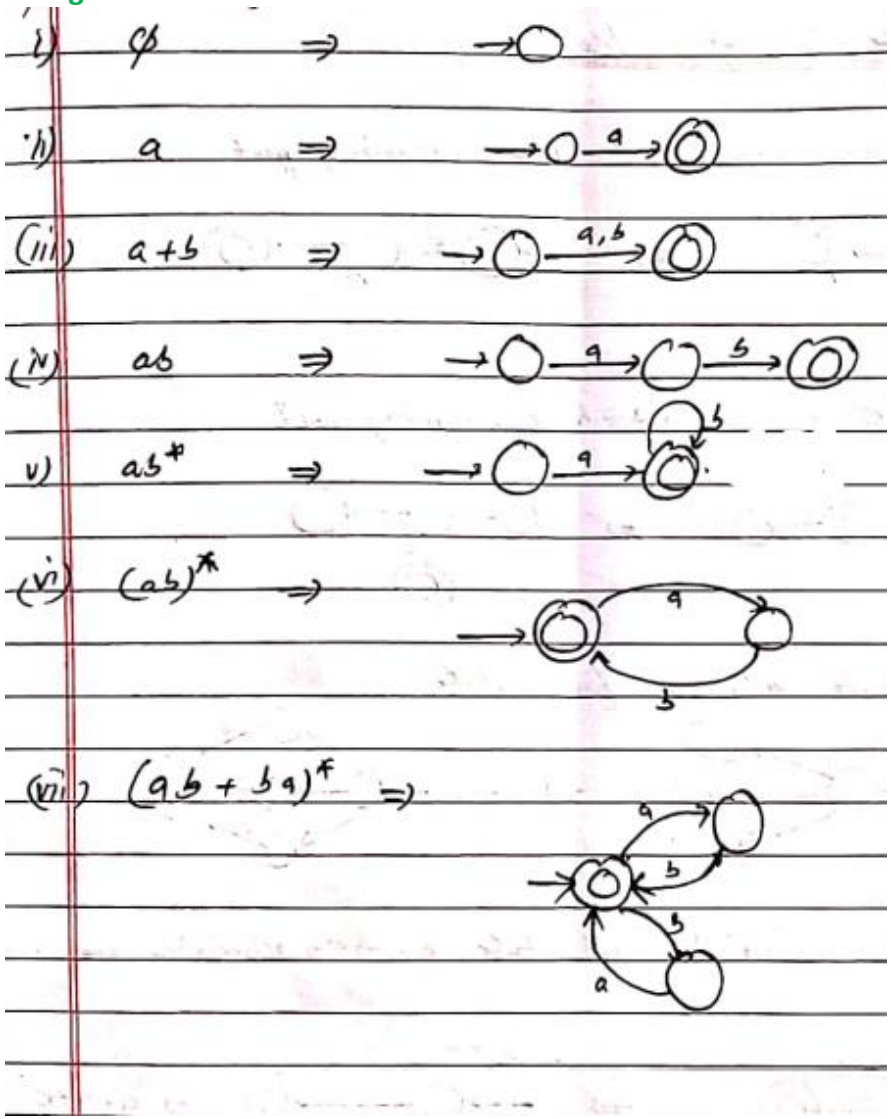
If r, r1, r2 and r3 are regular expressions, then:

Commutativity Law	• r1 U r2 = r2 U r1 • r1.r2 ≠ r2. r1
Associativity Law	• r1 U (r2 U r3) = (r1 U r2) U r3 • r1.(r2 .r3) = (r1. r2). r3
Distributive Law	• r1 (r2 U r3) =r1 r2 U r1 r3 • (r2 U r3) r1 = r2r1 U r3r1
Identity Law	• r1 U Φ = Φ U r1 = r1 • r1. ε = ε. r1 = r1
Annihilator Law	• Φ. r= r. Φ = Φ
Idempotent Law	• r U r =r

2.16 Finite Automata and RE

We can convert each of regular expression into finite automata. Also, from given automata, we can find out the regular expressions.

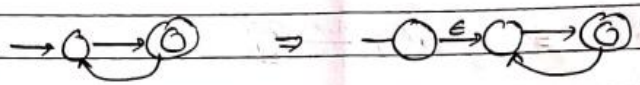
From Regex to FA



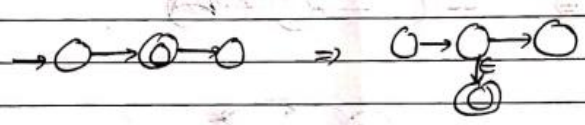
From FA to Regex

i) Using State Elimination Method

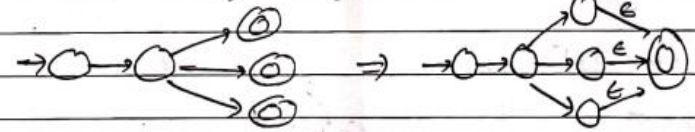
a) Initial state should not have incoming part.



b) Final state should not have outgoing part

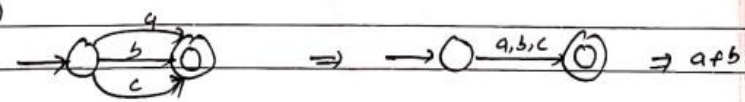


c) Should have only one final state

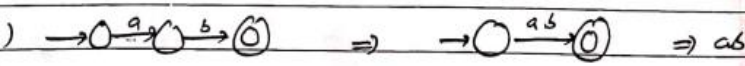


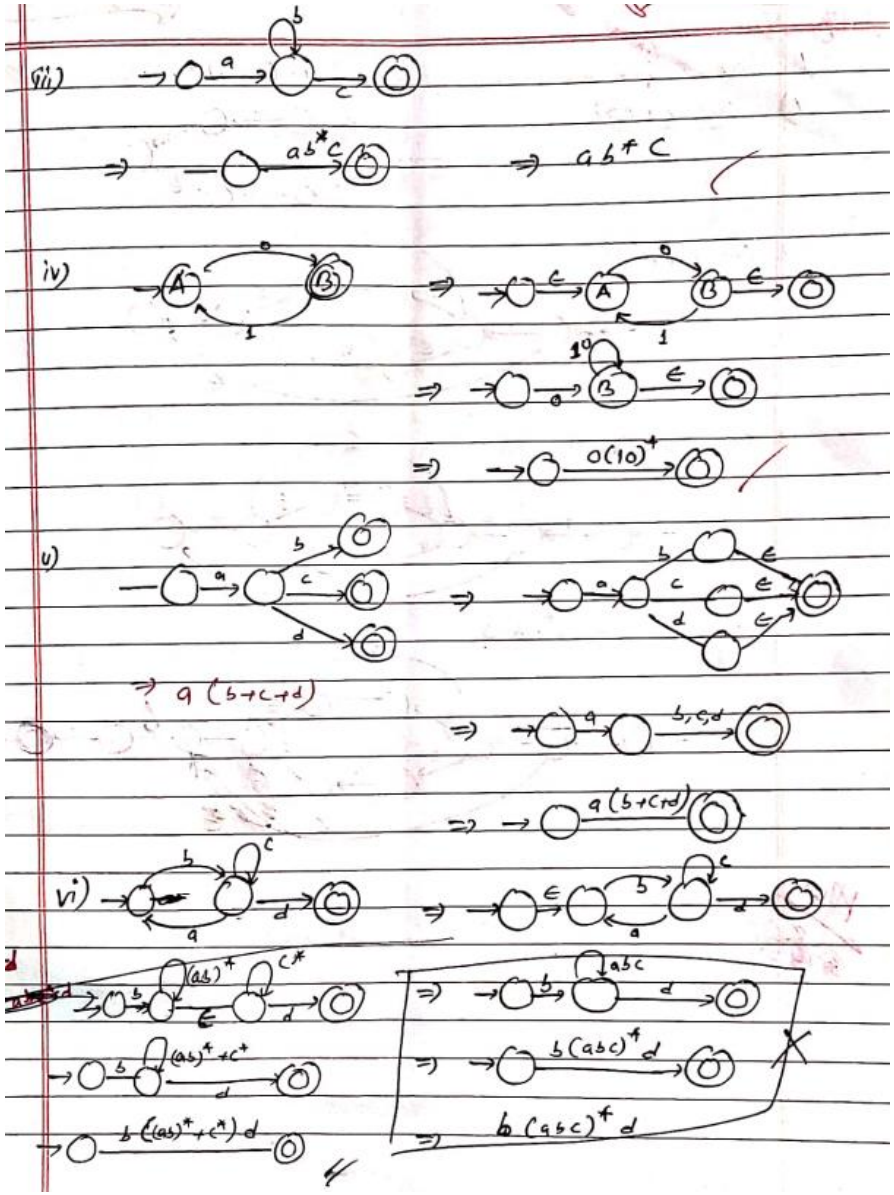
d) other than initial & final state, eliminate other state one by one

(i)



(ii)





2.17 Arden's Theorem

- "If P and Q are two RE over alphabet Σ, and if P does not contain null string ϵ, then the following equation in R is given by $R = Q + RP$ has a unique solution, ie. $R = QP^*$."
- It means, eqn in the form of $R = Q + RP$ can be replaced by $R = QP^*$.

Proof:

a) Proof of solution.

Let us take the equation, $R = Q + RP$ --①

Now, replacing R by $R = QP^*$,

$$R = Q + QP^*P$$
$$= Q(\epsilon + P^*P)$$
$$\therefore R = QP^* \quad \text{proved} \quad (\because \epsilon + R^*R = R^*)$$

b) Proof for the only solution

Consider, $R = Q + RP$

$$= Q + (Q + RP)P \quad (\because R = Q + RP)$$
$$= Q + QP + RP^2$$

Again replace R by $Q + RP$

$$R = Q + QP + (Q + RP)P^2$$
$$= Q + QP + QP^2 + RP^3$$
$$\vdots$$
$$= Q + QP + QP^2 + \cdots QP^n + RP^{n+1}$$

GURUKUL

$$= Q + QP + QP^2 + \cdots QP^n + QP^*P^{n+1} \quad (\because R + QP^*)$$
$$= Q(\epsilon + P + P^2 + \cdots P^n + P^*P^{n+1})$$
$$= QP^* \quad (\because P^* \text{ is the represent the closure of } P)$$

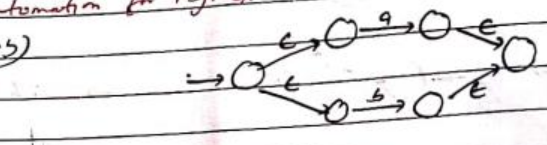
Hence proved

2.18 Numerical

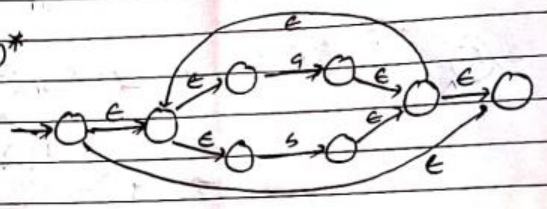
Numerical:

1. Find the GNFA for reg. ex $a \cdot (a+b)^* \cdot b \cdot b$

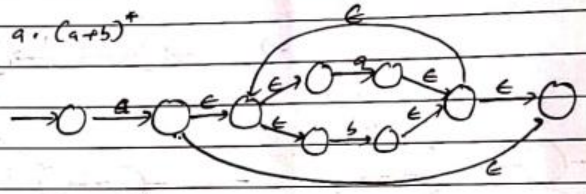
i) For $(a+b)$



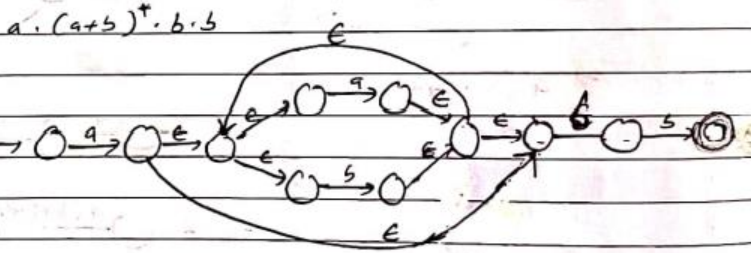
ii) For $(a+b)^*$



iii) For. $a \cdot (a+b)^*$

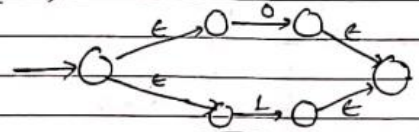


iv) For $a \cdot (a+b)^* \cdot b \cdot b$

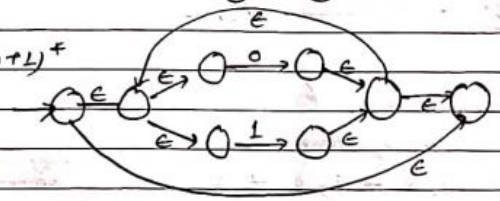


2. Construct ϵ-NFA for the reg ex $(0+1)^* 1 (0+1)$

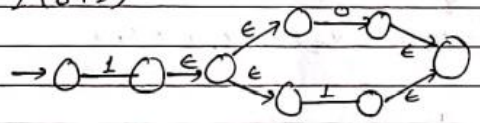
① For $(0+1)$



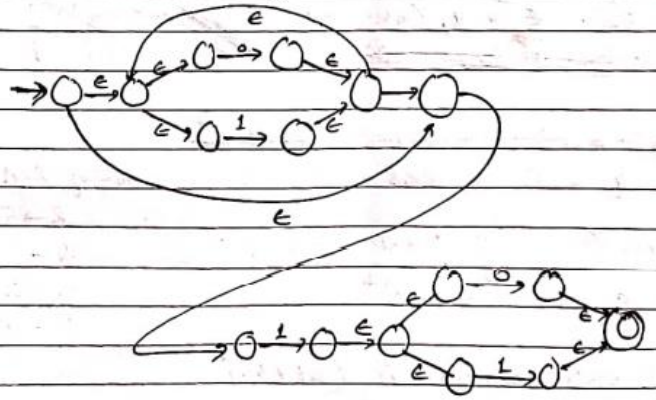
② For $(0+1)^*$



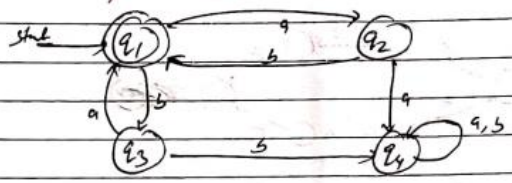
③ For $1(0+1)$



④ For $(0+1)^* 1 (0+1)$



3) Find the regular expression for transition diagram given in.



Sol: Using Arden's theorem,

Let us form the equations: (Considering Incoming edges)

$$q_1 = q_2 b + q_3 a + \epsilon$$
$$q_2 = q_1 a$$
$$q_3 = q_1 b$$
$$q_4 = q_2 a + q_3 b + q_4 a + q_4 b$$

Put q_2 and q_3 in q_1 as:

$$q_1 = q_1 ab + q_1 ba + \epsilon$$
$$q_1 = \epsilon + q_1 (ab + ba)$$

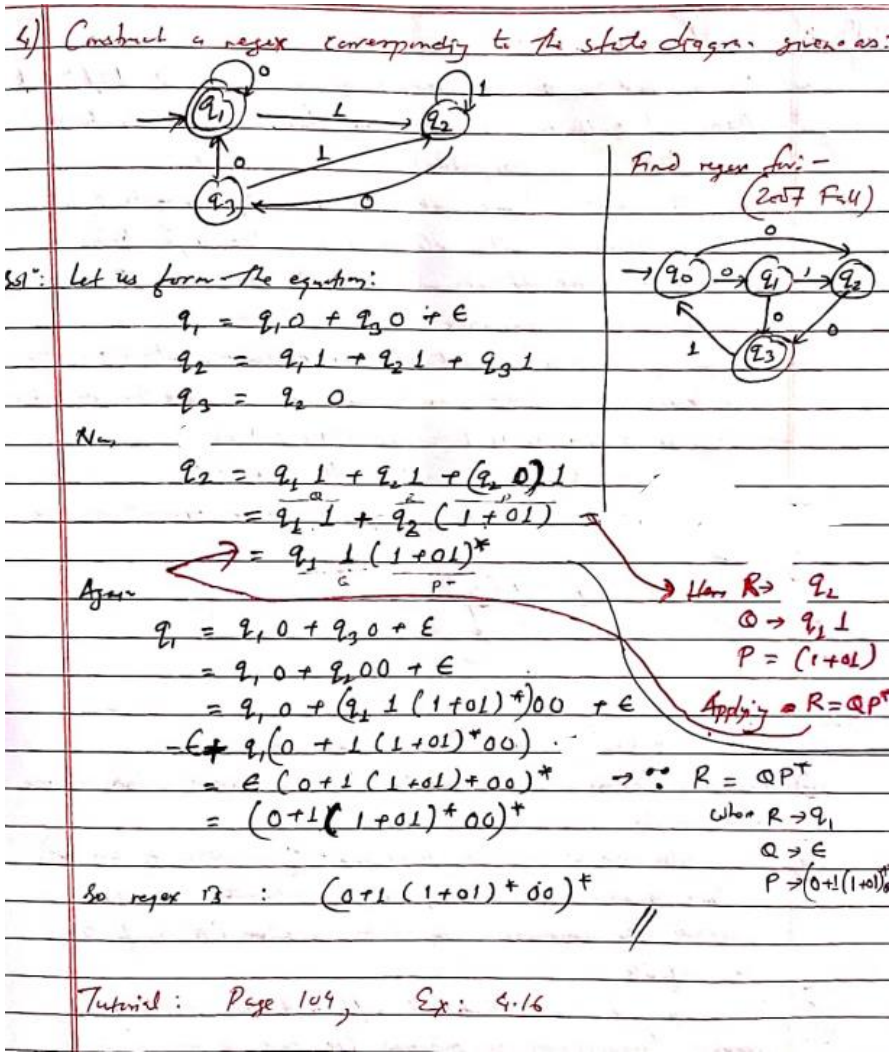
Here $q_1 = \epsilon + q_1 (ab + ba)$ is in the form of $R = Q + RP$

By Arden's theorem, $R = Q + RP \Rightarrow R = QP^*$

$\therefore q_1 = \epsilon (ab + ba)^*$

$R \to q_1$
$Q \to \epsilon$
$R \to q_1$
$P \to (ab + ba)$

So, req^d regular expression is $(ab + ba)^*$

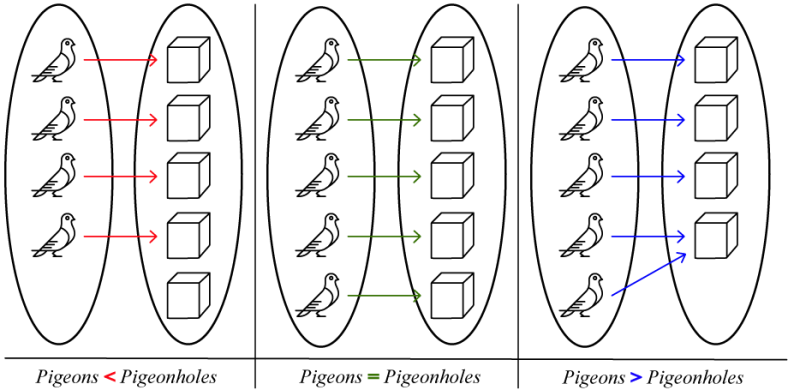


Tutorial:

1. Construct an NFA for (ab/ba)*ab

2.19 The pigeonhole principle

The idea is this.
If there are n number of pigeonholes and (n+1) pigeons, and if all the pigeons come home to roost, then there will be a pigeonhole that contains more than one pigeon.



Pigeonhole Principle Diagram

The pigeonhole principle, also known as the Dirichlet principle, originated with German mathematician Peter Gustave Lejeune Dirichlet in the 1800s, who theorized that given m boxes or drawers and n > m objects, then at least one of the boxes must contain more than one object.
If m is the number of objects (pigeons) and n is the number of boxes (pigeonholes), and let f: m→ n be a function, then

The abstract formulation of the principle: Let X and Y be finite sets

- If m<n, then function is one-to-one but not onto.
- If m=n, then function is both one-to-one but onto (bijection).
- If m>n , then function is onto but not one-to-one

Example:

For instance, suppose there are 35 different time periods during which classes at the local college can be scheduled. If there are 679 different classes, what is the minimum number of rooms that will be needed?

Let $n = 679$ (pigeons) and $m = 35$ (pigeonholes)

Then $\left\lceil \frac{679}{35} \right\rceil = \lceil 19.4 \rceil = 20$

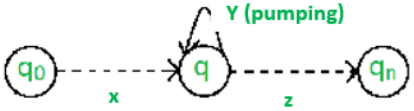
Therefore, the local college will need 20 different rooms to accommodate the different classes and periods.

2.20 Pumping Lemma for Regular Language

Statement

If L is a regular language, then L has a pumping length 'p' such that any string 's' where $|s| \geq p$ may be divided into 3 parts s=xyz such that the following conditions must be true:

1. xyiz ∈L for every $i \geq 0$
2. $|y| > 0$
3. $|xy| \leq p$

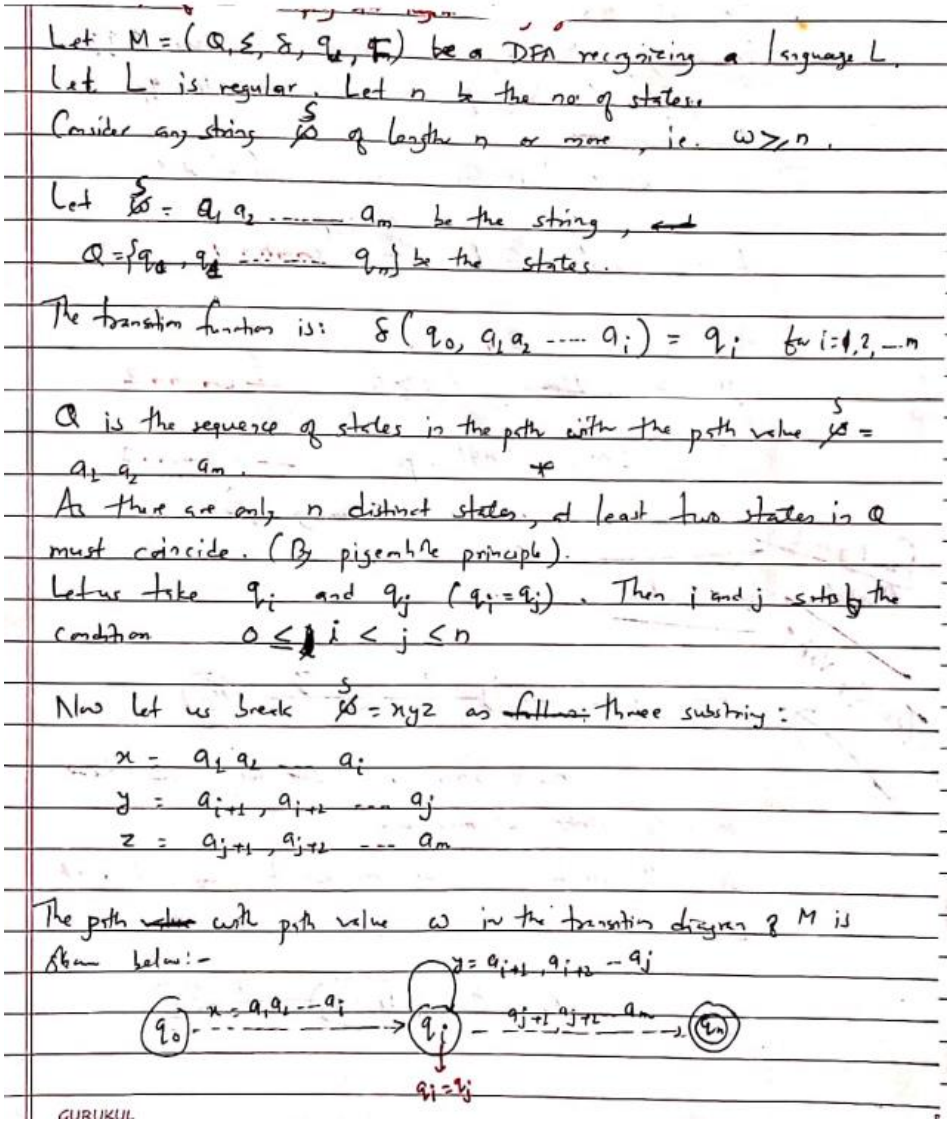


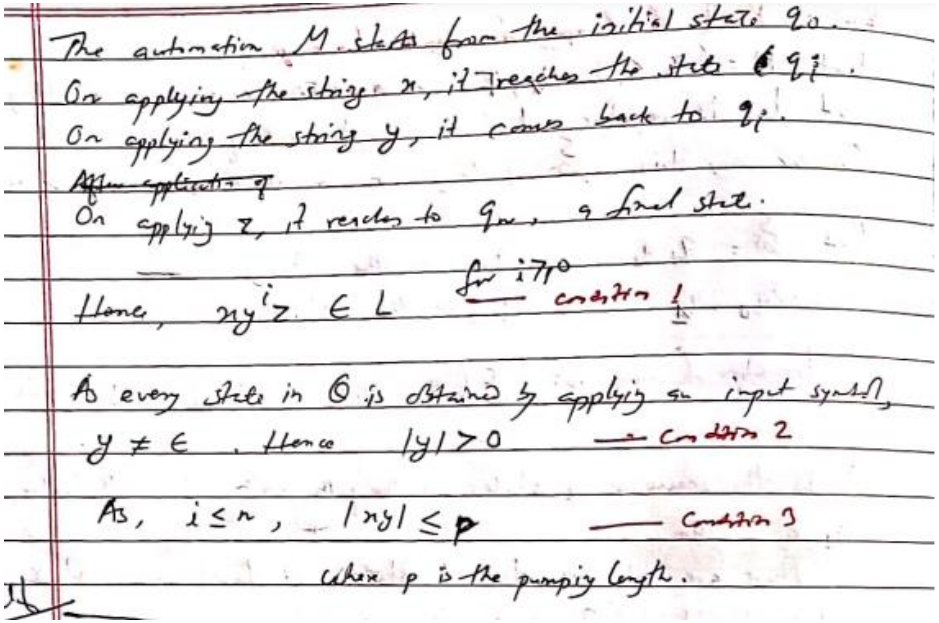
In simple terms, this means that if a string y is 'pumped', i.e., if y is inserted any number of times, the resultant string still remains in L.

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.

Proof




The automaton M starts from the initial state q_0.
On applying the string x, it reaches the state q_i.
On applying the string y, it comes back to q_i.
After application of
On applying z, it reaches to q_f, a final state.

Hence, $xy^iz \in L$ for $i \geq 0$ — condition 1

As every state in Q is obtained by applying an input symbol,
$y \neq \epsilon$. Hence $|y| > 0$ — condition 2

As, $i \leq n$, $|xy| \leq p$ — condition 3
where p is the pumping length.

2.21 Steps for proving given language is not Regular using Pumping Lemma theory

To prove that a language is not regular using Pumping Lemma, follow the below steps.

1. Assume that A is regular
2. It has to have a Pumping length (say P)
3. All strings longer than P can be pumped $|S| \geq P$
4. Now find a string 'S' in A such that $|S| \geq P$
5. Divide S into x,y,z
6. Show that $xy^iz \notin A$ for some i
7. Then consider all ways that S can be divided into xyz
8. Show that none of those can satisfy all the 3 pumping conditions at the same time.
9. S cannot be pumped == Contradiction.

2.22 Numerical

1. Show that $L = \{a^n b^n, n \geq 1\}$ is not regular.

Proof: Assume that L is regular, and the pumping length is P, and S is the string.

Here, $S = a^n b^n$

Let us consider $P = 7$

According to the Pumping Lemma statement, $|S| \geq P$, so we take here,

$S = a^P b^P = a^7 b^7 \Rightarrow aaaaaaa\ bbbbbbb$

This S can be divided in the form of xyz such that $xy^iz \in L$ for every $i \geq 0$, $|y| > 0$ and $|xy| \leq P$

Case 1: Let's divide S in xyz form where y is in the part of a.

Taking $i=2$, $xy^iz = xy^2z$

$= aa\ aaaaa\ bbbbbbb$

$= a^{11}b^7$

Here, $xy^iz \notin A$ NA satisfied

$|y| > 0$ satisfied

$|xy| \leq P$ Satisfied

Case 2: The y is in the part of b

a a a a a a a b b b b b b b
x y z

Taking i=2,

$xy^iz = xy^2z$

= aaaaaaabb bbbbbbbb

$= a^7b^{11}$

Here, $xy^iz \notin A$ NA satisfied

$|y| > 0$ satisfied

$|xy| \leq p$ NA satisfied

Case 3: The y is in the part of both a and b

a a a a a a b b b b b b
x y

Taking i=2,

$xy^iz = xy^2z$

= aaaaa aabbaabb bbbbb

$= a^7b^2a^2b^7$

Here $xy^iz \notin A$ NA satisfied

$|y| > 0$ satisfied

$|xy| \leq p$ Satisfied

From the above three cases, we can observe that none of the cases can satisfy all the three pumping conditions at the same time.

Hence, proved by contradiction, Given Lagyage is not regular.

2. Show that $A = \{yy \mid y \in \{0,1\}^*\}$ is not regular

yy → [first half & second half of the string are same]

Proof: Assume that A is regular, so must have pumping length.

Let pumping length = P and S is any string in the Language. Let P=7

Let us choose, $S = 0^p 1 0^p 1$

$= 0^7 1 0^7 1$ = 00000001 00000001

According to Pumping Lemma statement, the S can be divided into the form of xyz such that $xy^iz \in A$ for every $i \geq 0$, $|y| > 0$ and $|xy| \leq p$.

Case 1: Let's divide S in xyz where y is in the ~~first half~~

S = 0 0000000 1 00000001
x y z

Taking i = 2, $xy^iz = xy^2z$

= 00 00000000 01 00000001

⇒ 1st half ≠ 2nd half

$\therefore xy^iz \notin A$ Not satisfied

$|y| > 0$ satisfied

$|xy| \leq p$ satisfied

Case 2: Dividing S in xyz where y is in 2nd half

S = 00000001 00000001
x y z

~~Here $|xyz| \leq p$ NA satisfied.~~

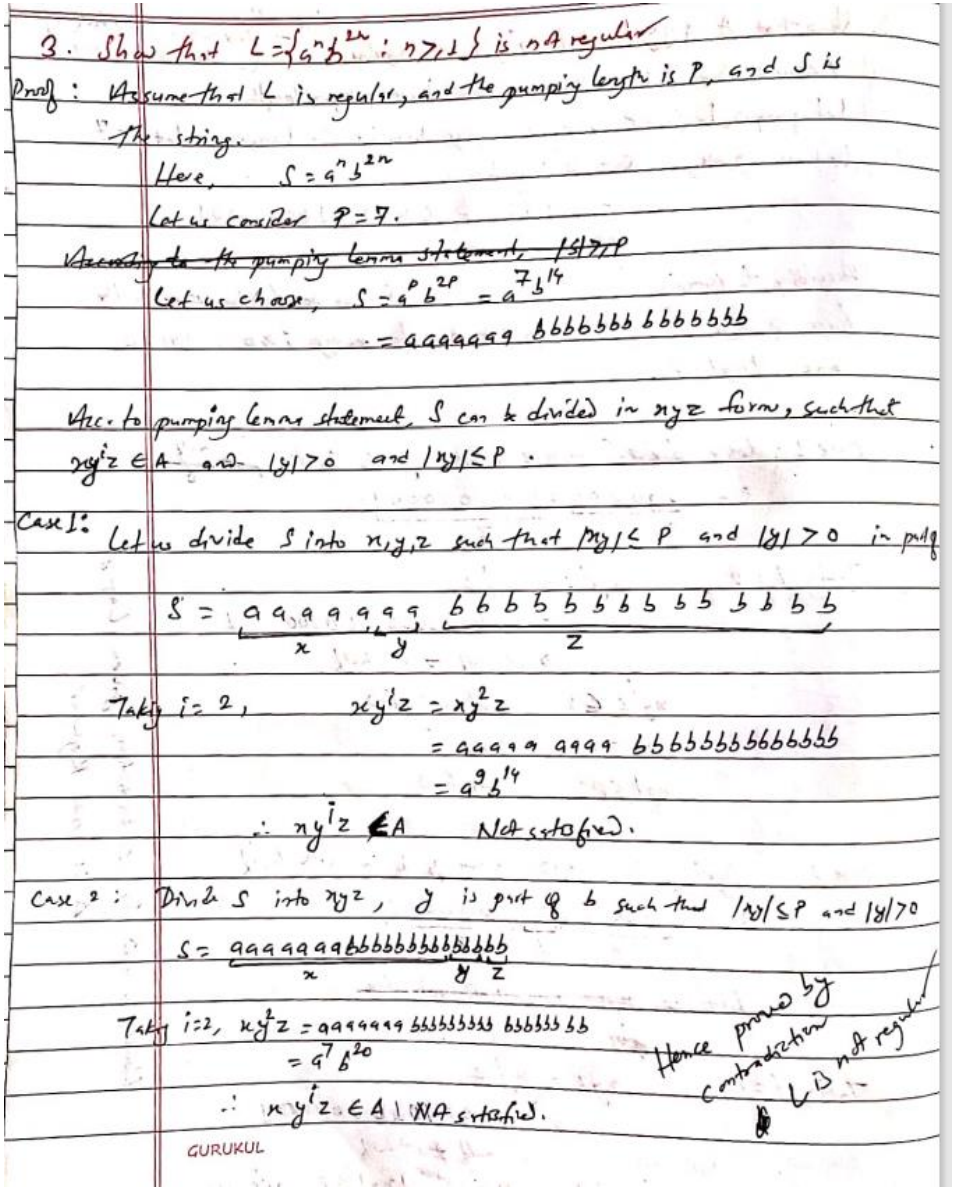
Taking i=2, $xyz = xy^2z$

= 00000001 00000001

GURUKUL NA satisfies ⇒ 1st half ≠ 2nd half

$\therefore xy^iz \notin A$, $|y| > 0$ satisfies $|xy| \leq p$ Satisfied

Hence, proved by contradiction, given Language is not Regular.

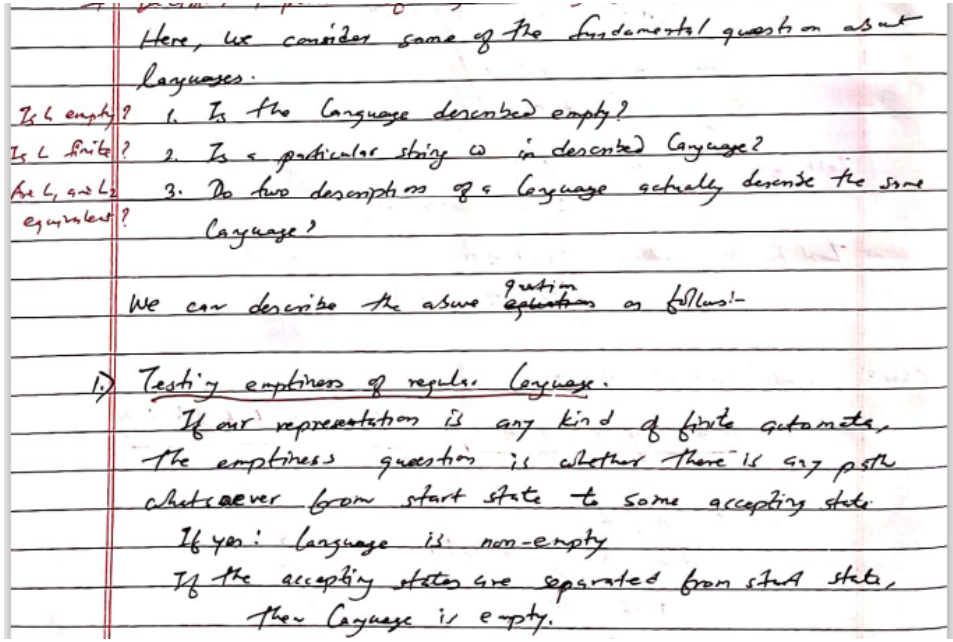


Tutorial:

Prove that below languages are not regular.

1. $L=\{a^n b a^n \text{ for } n=0,1,2,.....\}$
2. $L=\{a^n b a b^{n+1} \text{ for } n=1,2,3,.....\}$
3. $L=\{0^n 1^m 2^n \text{ for } n, m \geq n\}$
4. $L=\{0^n 1^m \mid n \leq m\}$
5. $L=\{0^n 1^{2n} \mid n \geq 1\}$
6. $L=\{0^n \mid n \text{ is a perfect square}\}$
7. $L=\{0^k \mid k \text{ is prime no}\}$

2.23 Decision properties of Regular Language



2) Testing membership in a regular Language

→ Let L be any regular language and w is any string, we have to test whether $w \in L$ or not.

We can represent L by some finite automata. Then simulate the automata processing the string of input symbols w, begining in the start state.

If automata ends in accepting state, the answer is yes otherwise no.

3) Testing for L_1 and L_2 if they are same.

For testing L_1 and L_2 to be the same language, we follow the algorithm:

1. Convert L_1 and L_2 to DFAs
2. Convert L_1 and L_2 to minimal DFAs.
3. Determine if the minimal DFAs are the same.

2.24 Closure Properties of Regular Language

Let L and M be regular languages. Then the following languages are all regular:

1. Union : $L \cup M$
2. Intersection : $L \cap M$
3. Complement : $\overline{M}$
4. Difference : $L \setminus M$
5. Reversal : $L^R = \omega^R : \omega \in L$
6. Closure : L^*
7. Concatenation : $L \cdot M$ → substitution of strings for symbols
8. Homomorphism : $h(L) = \{h(\omega) / \omega \in L, h \text{ is a homomorphism}\}$
9. Inverse homomorphism:

$h^{-1}(L) = \{\omega \in \Sigma / h(\omega) L, h: \Sigma \to \Delta$ is a homomorphism

1. Union:

If L_1 and L_2 are two regular Languages, their union $L_1 \cup L_2$ will also be regular.

Eg. $L_1 = \{a^n / n > 0\}$ and $L_2 = \{b^m / n \geq 0\}$

Then $L_3 = L_1 \cup L_2 = \{a^n \cup b^n / n > 0\}$ is also regular

2. Intersection:

If L_1 and L_2 are two regular Languages, their intersection $L_1 \cap L_2$ will also be regular.

Eg. $L_1 = \{a^m b^n / n > 0$ and $m > 0\}$ and

$L_2 = \{a^m b^n \cup b^n a^n / n > 0$ and $n > 0\}$

Then $L_3 = L_1 \cap L_2 = \{a^m b^n / n > 0$ and $n > 0\}$ is also regular

3. Concatenation:

If L_1 and L_2 are two regular Languages, their concatenation $L_1 \cdot L_2$ will also be regular

Eg. $L_1 = \{a^m / m > 0\}$ and $L_2 = \{b^n / n > 0\}$

Then $L_3 = L_1 \cdot L_2 = \{a^m \cdot b^n / m > 0$ and $n > 0\}$ is also regular

4. Kleene Closure:

If L_1 is a regular Language, its kleene closure L_1^* will also be regular

Eg. $L_1 = (a \cup b)$

$L_1 = (a \cup b)^*$

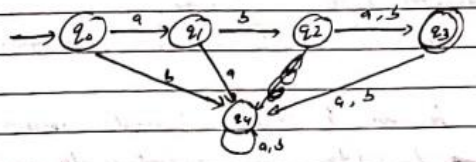
2.25 Numerical

Show that If L is regular then complement of $L(H\bar{L})$ is also regular.

→ If L is a regular language, then $\bar{L}$ is also regular.
Let us consider a DFA that accepts only the strings aba and abb.

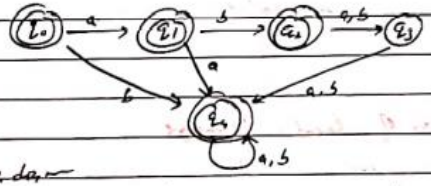
The state diagram is:-

$L = \{ w \in \{a,b\}^+ : w \text{ is aba or abb} \}$



To find out the req^d FA which accepts strings other than aba and abb, we convert every non-final state to final state, and every final state to non-final.

$\bar{L} = \{ w \in \{a,b\}^+ : w \text{ is not aba or abb} \}$



This is the complement of above state diagram, which accepts all other strings except aba and abb.

Thus we can say that if L is regular, then $\bar{L}$ is also regular.

End of Chapter 2

3.1 Introduction to Grammar

- In the literary sense of the term, grammar denote syntactical rules for conversation in natural language
- Linguistics have attempted to define grammars since the inception of natural languages like English, Sanskrit, Mandarin, etc.
- The theory of languages finds its applicability extensively in the fields of Computer Science.
- Noam Chomsky, in 1956, gave a mathematical model of grammar which is effective for writing computer languages.
- A grammar is nothing but a set of rules to define valid sentences in any languages.

A grammar G can be formally written as a 4-tuple (V_n, T, S, P) where:

V_n → Set of variables or non-terminals generally represented by capital letters, A, B, C, D - -

T or Σ → Set of terminals, generally represented by small letters a, b, c, d - - .

S → starting non-terminal, $S \in V_n$

P → Production rules for terminals and non-terminals.

A production rule has the form $\alpha \to \beta$, where α and β are strings on $V_n \cup \Sigma$ and at least one symbol of α belongs to V_n.

Eg: Grammar G1:

$(\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to a, B \to b\})$

Here,

S, A, and B are non-terminal symbols.

a and b are terminal symbols

S is the start symbol, $S \in V_n$

Productions, P: $S \to AB$, $A \to a$, $B \to b$

Eg: Derivation from a grammar

Let us consider the grammar

$G2 = (\{S, A\}, \{a, b\}, S, \{S \to aAb, aA \to aaAb, A \to \epsilon\})$

Some of the strings that can be derived are:

$S \Rightarrow aAb$	using production	$S \to aAb$
$\Rightarrow aaAbb$	"	$aA \to aaAb$
$\Rightarrow aaaAbbb$	"	$aA \to aaAb$
$\Rightarrow aaabbb$	"	$A \to \epsilon$

3.2 Context-free Grammar

- a more powerful tool/method of describing language
- It can describe certain features that have a recursive structure.
- Collections of languages associated with CFG are called CFL.
- Every regular grammar is context-free, so a regular language is also a context-free one.

- It is already proved by pumping lemma that Language $\{a^n b^n \mid n \geq 0\}$ is not regular; but it is possible to design a CFG for those languages.
- So, It is very clear that "the family of regular language is a proper subset of the family of context-free language."

Applications of CFG

1. For defining programming languages
2. For parsing the program by constructing syntax tree
3. For translation of programming languages
4. For describing arithmetic expressions
5. For construction of compilers.

Formal Definition of CFG

A CFG is a 4-tuple such that: $G = (V, \Sigma, R, P)$

where

V or V_n = A finite non-empty sets of variables / non-terminal symbols

Σ or T or V_t = Finite set of terminal symbols

S = Starting non-terminal such that $S \in V$

R or P = Set of production rules of the form $A \to \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$

Consider a grammar $G = (V, \Sigma, S, P)$ where.

$V = \{S\}$

$\Sigma = \{a, b\}$

$S = \{S\}$

$P = \{S \to aSbS,\ S \to bSaS,\ S \to \in\}$

Strings that can be generated are:

$S \Rightarrow aSbS$ ∵ $S \to aSbS$

$\to abSaSbS$ ∵ $S \to bSaS$

$\to ababSbS$ ∵ $S \to \in$

$\to ababbaS$ ∵ $S \to \in$

$\to ababba$

$S \to aSbS$

$\to abSaSbS$

$\to ab\in aSbS$

$\to abaebS$

$\to ababbSaS$

$\to ababba\in aS$

$\to ababba\in\in$

$\to ababba$

Hence, it generates the strings having equal no of a's and b's.

3.3 Some terminologies

Derivation

Let $G = (V, \Sigma, R, S)$ be a context-free grammar

If $\omega_1, \omega_2 \ldots \omega_n$ are strings over variable V such that $\omega_1 \Rightarrow \omega_2 \Rightarrow \ldots \Rightarrow \omega_n$, then we say ω_n is derivable.

The sequence of substitution to obtain a string is called derivation. Simply derivation is a sequence of production rules.

Say ω_i derives ω_n, ie. $\omega_i \overset{*}{\Rightarrow} \omega_n$,

Then, the sequence of steps to obtain ω_n from ω_i is called derivation.

Language of CFG, L(G)

If $G = (V, \Sigma, R, S)$ is a CFG, then the Language of G denoted by $L(G)$ is the set of terminal strings that have derivations from start symbol.

ie. $L(G) = \{ w \in \Sigma^* : S \overset{*}{\Rightarrow} w \}$

Sentential Forms

Derivation from the start symbol produce strings that have a special rule. And this is called sentential form.

If $G = (V, \Sigma, R, S)$ is a CFG, then any string α in $(V \cup \Sigma)^*$ such that $S \overset{*}{\Rightarrow} \alpha$ is a sentential form.

If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then α is a left sentential form

If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$, then α is a right sentential form.

Example:

Let any set of production rules in a CFG is:

$X \rightarrow X+X \mid X*X \mid X \mid a$ over an alphabet $\{a\}$.

Then, the left most derivation is:

$X \rightarrow X+X$ ∵ $X \rightarrow X+X$

$\rightarrow a+X$ $X \rightarrow a$

$\rightarrow a+X*X$ $X \rightarrow X*X$

$\rightarrow a+a*a$ $X \rightarrow a$

Hence, $a+a*a$ is a left sentential form.

The right most derivation is:

$X \rightarrow X*X$	∵ $X \rightarrow X+X$
$\rightarrow X*a$	$X \rightarrow a$
$\rightarrow X+X*a$	$X \rightarrow X+X$
$\rightarrow X+a*a$	$X \rightarrow a$
$\rightarrow a+a*a$	$X \rightarrow a$

Hence $a+a*a$ is a right sentential form.

$X \rightarrow X+X$
$\rightarrow X+a$
$\rightarrow X*X+a$
$\rightarrow X+a+a$
$\rightarrow a*a+a$

Here $a*a+a$ is a right sentential form.

3.4 Numerical

1. Find the CFL for a CFG where, $G = (V, \Sigma, R, S)$ where

$V = \{S\}$

$\Sigma = \{a, b\}$

$R = \{ S \rightarrow aSb,\ S \rightarrow ab \}$

Sol^n: Given: $V = \{S\}$

$\Sigma = \{a, b\}$

$R = \{S \to aSb,\ S \to ab\}$

To find: $L(G) = ?$

Here, S is the only terminal which is the starting symbol for the grammar a and b are terminals.

Now,

$S \Rightarrow \underline{aSb}$ $\quad \because S \to aSb$

$\Rightarrow aabb$ $\quad \because S \to ab$

$\Rightarrow a^2b^2$

Applying first production n-1 times followed by an application of second production, we get:

$S \Rightarrow a\underline{S}b$ $\quad \because S \to aSb$

$\Rightarrow aaSbb$ $\quad \because S \to aSb$

$\Rightarrow \cdots$

$\Rightarrow \cdots$

$\Rightarrow a^{n-1}Sb^{n-1}$

$\Rightarrow a^nb^n$

Hence Language of given grammar is:

$L(G) = \{a^nb^n : n \geq 1\}$

2. Let $G = (V, \Sigma, R, S)$ where, $V = \{S\}$

$\Sigma = \{a, b\}$

$R = \{S \to aaS,\ S \to b\}$

Find $L(G)$

Sol^n: Using given rules recursively:

i) $S \to b$

(ii) $S \to aa\underline{S}$

$\to aab$

(iii) $S \to aa\underline{S}$

$\to aaaa\underline{S}$

$\to aaaab$

$\to (aa)^2b$

(iv) $S \to aa\underline{S}$

$\to aaaa\underline{S}$

$\to (aa)^nb$

Hence, Language of given grammar is:

$L(G) = \{(aa)^nb : n \geq 0\}$

3. Let $G = (V, \Sigma, R, S)$ where $V = \{S, A\}$

$\Sigma = \{a, b\}$

$R = \{S \to aA \mid \in,\ A \to bS\}$; Find $L(G)$

Sol^n: Using given rules recursively

(i) $S \to \in$

(ii) $S \to a\underline{A}$

$\to abS$

$\to ab\in$

$\to ab$

(iii) $S \to a\underline{A}$

$\to abS$

$\to abaA$

$\to ababS$

$\to ababab\in$

$\to ababab$

$\to (ab)^2$

(iv) $S \to a\underline{A}$

$\to ab\underline{S}$

$\to abaA$

$\to ababS$

$\vdots$

$\to (ab)^n$

Hence Language of given grammar is:

$L(G) = \{(ab)^n : n \geq 0\}$

7

4. Write a CFG for the Language given by:
$L = \{ a^m b^n : m > n \}$.

Sol^n: The possible strings are:
$\epsilon, a, b, aab, aaabb, aaab, aaabbb, \ldots$

Design of R as:
$S \rightarrow aS \mid aSb \mid \epsilon$

Let $G = (V, \Sigma, R, S)$ be required CFG
where,
$V = \{S\}$
$\Sigma = \{a, b\}$
$R = \{ S \rightarrow aSb, S \rightarrow aS, S \rightarrow \epsilon \}$
$S = \{S\}$

Now, Check for validity:

For, a
$S \rightarrow aS$
$\rightarrow a\epsilon$
$\rightarrow a$

aab
$S \rightarrow aS$
$\rightarrow aaSb$
$\rightarrow aa\epsilon b$
$\rightarrow aab$

aaabbb
$S \rightarrow aS$
$\rightarrow aaSb$
$\rightarrow aaaSbb$
$\rightarrow aaa\epsilon bb$
$\rightarrow aaabb$

aaab
$S \rightarrow aS$
$\rightarrow aaSb$
$\rightarrow aaaSb$
$\rightarrow aaa\epsilon b$
$\rightarrow aaab$

Rough work:
a
$S \rightarrow aS \mid \epsilon$

aab
$S \rightarrow aSb \mid aS \mid \epsilon$

aaabbb
$S \rightarrow aSb \mid aS \mid \epsilon$

5. Write a CFG for the language given by:
$L = \{ a^m b^n : m > 0, n > 0 \}$

Sol^n: Possible strings are:
$a, b, ab, aabb, aaabbb, aab, aaab, abb, abbb, \ldots$

Design of R as:
$S \rightarrow aSb \mid \epsilon$
$S \rightarrow aS \mid bS$

Check for validity:

ab
$S \rightarrow aSb$
$\rightarrow ab$

aaabbb
$S \rightarrow aSb$
$\rightarrow aaSbb$
$\rightarrow aaaSbbb$
$\rightarrow aaabbb$

abbb
$S \rightarrow aSb$
$\rightarrow aSbb$
$\rightarrow abbSb$
$\rightarrow abbbb$

aab
$S \rightarrow aSb$
$\rightarrow aaSb$
$\rightarrow aab$

Hence, the CFG can be:
$G = (V, \Sigma, R, S)$
where, $V = \{S\}$
$\Sigma = \{a, b\}$
$R = \{ S \rightarrow aSb \mid \epsilon$
$S \rightarrow aS \mid bS \}$
$S = \{S\}$

Rough Work
For ϵ
$S \rightarrow \epsilon$
For a
$S \rightarrow aS \mid \epsilon$
For b
$S \rightarrow bS \mid \epsilon$
For ab
$S \rightarrow aSb \mid \epsilon$
For aabb
$S \rightarrow aSb \mid \epsilon$
For abb
$S \rightarrow aSb \mid \epsilon$

6. Write a CFG for the language given by:-

$L = \{ w \in \{a,b\}^* : \text{there is equal no. of } a \text{ and } b \}$

→ Possible strings are.

ba, ab, aabb, abab, aabbabaabb, ...

Design of R as:

$S \rightarrow aSb \mid bSa \mid \epsilon$

Check for validity.

aabb	abab	aabbabaabb
$S \rightarrow aSb$	$S \rightarrow aSb$	$S \rightarrow aSb$
$\rightarrow aaSbb$	$\rightarrow abSab$	$\rightarrow aaSbb$
$\rightarrow aabb$	$\rightarrow abab$	$\rightarrow aabbSaabb$
		$\rightarrow aabbaSbaabb$
		$\rightarrow aabbabaabb$

Hence, CFG be:

$G = (V, \Sigma, R, S)$

where,

$V = \{S\}$

$\Sigma = \{a, b\}$

$R = \{S \rightarrow aSb \mid bSa \mid \epsilon\}$

$S = \{S\}$

Reference:

youtube.com/watch?v=27_iiKXfP08

Rough work:

For ab

$S \rightarrow aSb \mid \epsilon$

For aabb

$S \rightarrow aSb \mid \epsilon$

For ba

$S \rightarrow bSa \mid \epsilon$

For bbaa

$S \rightarrow bSa \mid \epsilon$

10

7. Write a CFG for the language given by:

$L = \{a^m b^n : m \geq 1, n \geq 1\}$

Sol: Possible strings are:

ab, aabb, aaabbb, aaab, abb, aaabb, abbb, ...

Design of R as:-

$S \rightarrow AB$

$A \rightarrow aA$

$A \rightarrow a$

$B \rightarrow bB$

$B \rightarrow b$

Check for validity:

ab	aab	abb	aaabbb
$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
$\rightarrow aB$	$\rightarrow aAB$	$\rightarrow aB$	$\rightarrow aAB$
$\rightarrow ab$	$\rightarrow aaB$	$\rightarrow abB$	$\rightarrow aaAB$
	$\rightarrow aab$	$\rightarrow abb$	$\rightarrow aaaBB$
			$\rightarrow aaabBB$
			$\rightarrow aaabbB$
			$\rightarrow aaabbb$

Hence the CFG can be:

$G = (V, \Sigma, R, S)$

where, $V = \{S, A, B\}$

$\Sigma = \{a, b\}$

$R = \{ S \rightarrow AB$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b \}$

$S = \{S\}$

Rough

ab

$S \rightarrow aSb \mid \epsilon$

aabb

$S \rightarrow aSb \mid \epsilon$

aab

$S \rightarrow aSb \mid a \mid \epsilon$

?

$S \rightarrow AB$

$A \rightarrow aA \mid a$

$B \rightarrow b$

abb

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow bB \mid b$

aaab

$S \rightarrow AB$

$A \rightarrow aA \mid a$

$B \rightarrow b$

Tutorial: Book page no: 131 to 147

Write a CFG for:-

1. Which generates string of balanced parenthesis. ( ) (( ))
2. which generates palindrome for binary numbers. 0, 1, 010, 101
3. which generates strings having equal no of a and b. ✓ done.
4. for the regular expression $r = (a+b)^+$ Eg: ab aba
5. " " $r = 0^*1(0+1)^+$ Eg: 001 01
6. " " $r = (a+b)^+ aa (a+b)^+$ Eg: baabaa
7. that generates string having any combination of a's and b's except null string.
8. for the language:
   i) $L(G) = \{ww^r : w \in \{0,1\}^+$ Eg: 001100
   ii) $L(G) = \{ab(bbaa)^n bba(ba)^n : n \geq 0\}$ Eg: $ab(bbaa)^2 bba(ba)^1$
   iii) $L = \{a^n b^m : n \neq m\}$
   iv) $L = \{(0^n 1^n / n \geq 0) \cup (1^n 0^n / n \geq 0)\}$
   v) $L = \{a^n b^n c^m d^n / n >= 1, m \geq 1\}$ Eg: aaabbbcd, n=3, m=1
   vi) $L = \{a^n b^m c^m d^n / n \geq 1, m \geq 1\}$ Eg: aabbccdd

3.5 Bacus Naur Form (BNF)

- Bacus-Naur Form or Bacus Normal Form is the notation that involves minor changes in the format and some shorthand.

The above grammar can be rewritten in BNF as:-

$S \rightarrow aSa \mid bSb \mid c$

3.6 Parse Tree/ Derivation Tree

- It is a tree representation of strings of terminals using the productions defined by the grammar
- It pictorially shows how the start symbol of a grammar derives a string in language.
- It is an ordered tree that shows the essentials of a derivation.

- Formally, given CFG, $G = (V, \Sigma, R, S)$, a parse tree has following properties:-
  1. The root is labelled by start symbol.
  2. Each interior node of parse tree are variables.
  3. Each leaf node is labelled by terminal symbol or ϵ.
  4. If an interior node is labelled with non-terminal A and its children are $X_1, X_2, \ldots X_n$ from left to right and then there is a production A as: $A \rightarrow X_1 X_2 \ldots X_n$ for each $X_i \in \Sigma$

- Example:

Consider a grammar where $S \rightarrow aSa \mid a \mid b \mid \epsilon$

Now, for string aabaa,

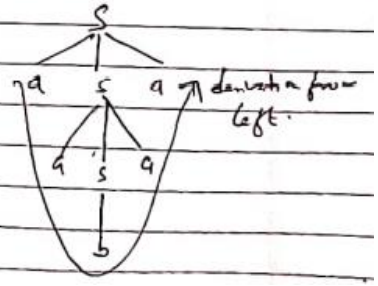
we have

$S \rightarrow aSa$

$\rightarrow aaSaa$

$\rightarrow aabaa$

Now parse tree is:



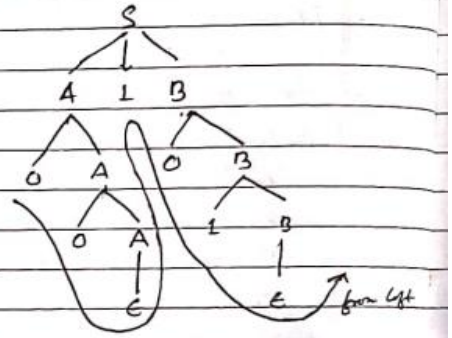
3.7 Numerical

1. Consider the grammar as: $S \to A1B$, $A \to 0A \mid \epsilon$, $B \to 0B \mid 1B \mid \epsilon$

Construct parse tree for: 00101, 1001, 00011

i) For 00101

$S \to A1B$
$\to 0A1B$
$\to 00A1B$
$\to 00\epsilon 1B$
$\to 001 0B$
$\to 0010 1B$
$\to 00101\epsilon$
$\to 00101$



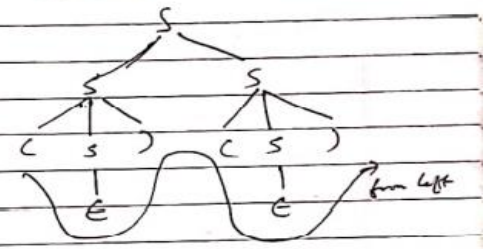
2. Let $G = (V, \Sigma, R, S)$ where $V = \{S\}$, $\Sigma = \{(,)\}$, $R = \{S \to SS, S \to (S), S \to \epsilon\}$

Construct parse tree for ( ) ( )

For ( ) ( )

$S \to SS$
$\to (S)S$
$\to (S)(S)$
$\to (\epsilon)(S)$
$\to ( )(\epsilon)$
$\to ( )( )$

Parse tree



3.8 Left Most and Right-most Derivation

- In LMD, leftmost variable is replaced first
- In RMD, rightmost variable is replaced first

Q: Consider a grammar

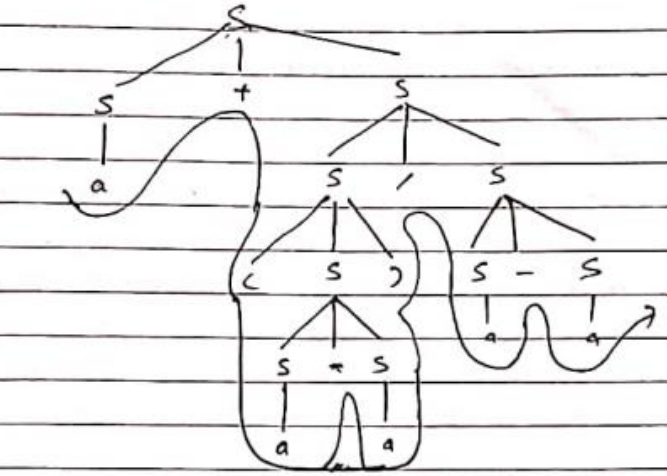
$S \to S+S$
$S \to S \mid S$
$S \to (S)$
$S \to S-S$
$S \to S*S$
$S \to a$

Now derive string $a+(a*a) \mid a-a$ with LMD and RMD also draw parse tree

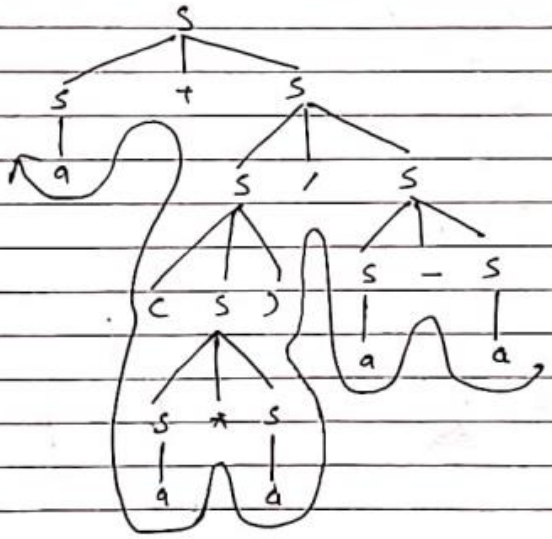
Sol:

LMD	RMD
$S \to S+S$	$S \to S+S$
$\to a+S$	$\to S+S \mid S$
$\to a+S \mid S$	$\to S+S \mid S-S$
$\to a+(S) \mid S$	$\to S+S \mid S-a$
$\to a+(S*S) \mid S$	$\to S+S \mid a-a$
$\to a+(a*S) \mid S$	$\to S+(S) \mid a-a$
$\to a+(a*a) \mid S$	$\to S+(S*S) \mid a-a$
$\to a+(a*a) \mid S-S$	$\to S+(S*a) \mid a-a$
$\to a+(a*a) \mid a-S$	$\to S+(a*a) \mid a-a$
$\to a+(a*a) \mid a-a$	$\to a+(a*a) \mid a-a$

Parse tree by LMD :



Parse tree by RMD



3.9 Ambiguity in Grammar

- A grammar is said to be ambiguous if for a given string generated by the grammar, there exists:
  - more than one leftmost derivation
  - or more than one rightmost derivation
  - or more than one parse tree (or derivation tree)
- If the grammar is not ambiguous, then it is called unambiguous.
- If the grammar has ambiguity, then it is not good for a compiler construction.
- No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing whole grammar without ambiguity.

Eg:

Consider a grammar which has production rules as:-

$S \rightarrow AB \mid aaB$

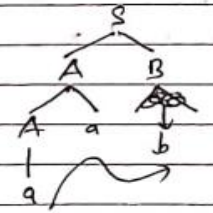
$A \rightarrow a \mid Aa$

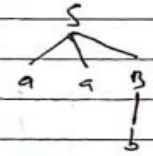
$B \rightarrow b$

Now for string aab, We have two leftmost derivation as:-

(i) $S \rightarrow \underline{AB}$
$\rightarrow \underline{Aa}B$
$\rightarrow aaB$
$\rightarrow aab$

(ii) $S \rightarrow aaB$
$\rightarrow aab$





Hence the grammar is ambiguous.

3.10 Numerical

1. If CFG, G is $S \to SbS \mid a$, show that G is ambiguous.

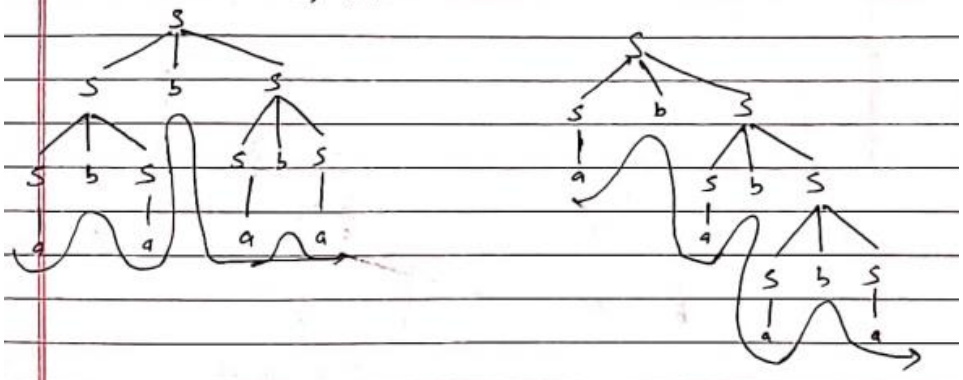
→ To prove that G is ambiguous, we have to find a string, $\omega \in L(G)$ which is ambiguous.

→ Let the $\omega = abababa \in L(G)$.

→ Then we get two derivation tree for ω as below.

$S \to \underline{S}bS$
$\to \underline{S}bS\,bS$
$\to ab\underline{S}\,bS$
$\to ababS$
$\to abab\underline{S}bS$
$\to ababab\underline{S}$
$\to abababa$

$S \to S\underline{bS}$
$\to \underline{S}b\underline{S}bS$
$\to ab\underline{S}\,bS$
$\to abS\underline{bS}$
$\to ab\underline{S}b\underline{S}$
$\to abababa$



Since we get two different derivation tree by LMD, Hence, the given grammar is ambiguous.

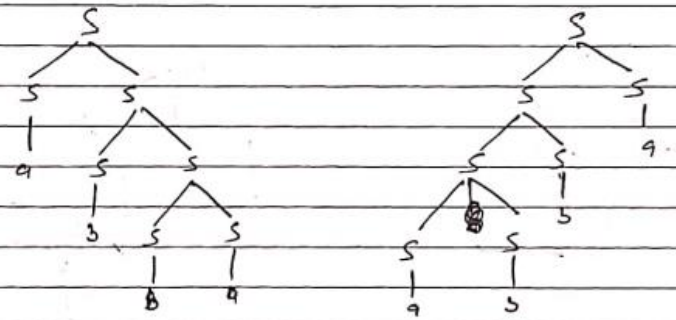
2. Show that the grammar $S \to a \mid abSb \mid aAb$, $A \to bS \mid aAAb$ is ambiguous. Consider $\omega = abab$

3. Show that the grammar $S \to aB \mid ab$, $A \to aAD \mid a$, $B \to ABb \mid b$ is ambiguous. Consider $\omega = ab$

4. Check if given grammar is ambiguous or not:- $S \to SS$, $S \to a \mid b$ Consider $\omega = abba$

Sol": Let us ~~draw the parse tree for the string~~ consider a string ω generated by the given grammar $\omega = abba$

Now let us draw parse tree for this string ω. by LMD



Parse tree-01 Parse tree 02

Since two different parse trees exist for string ω, therefore the given grammar is ambiguous.

Refer: gatevidyalay.com/grammar-ambiguity-ambiguous-grammar/

3.11 Simplification of CFG

- Various languages can efficiently be represented by a CFG.
- However, all the grammar are not always optimized, that means the grammar may consists of some extra symbols (non-terminals).
- Having extra symbols, unnecessarily increase the length of grammar.
- Simplification of grammar means reduction of grammar by removing useless symbols

The properties of reduced grammar are:

1. Each variable (ie. non-terminal) and each terminal of G appears in the derivation of some word in L
2. There should not be any production as $X \to Y$ where X and Y are non-terminals.
3. If ε is not in the language L, then there need not to be the production $X \to \varepsilon$

1. Eliminating useless symbols

- Useless symbols are those variables or terminals that do not appear in any derivation of a terminal string from the start symbol

For eg:

$S \to aaB \mid abA \mid aaS$

$A \to aA$ ⟶ non-generating

$B \to ab \mid b$

$C \to ad$ ⟶ useless

In the above example,

The variable C will never occur in any string, so the production C→ad is useless. So we will eliminate it, and other productions are written in such a way that variable C can never reach from the starting variable S.

Production $A \to aA$ is also useless because there is no way to terminate it.

For a given grammar $G = (V, \Sigma, R, S)$, a symbol x is useful if, There is some derivation of the form: $S \overset{*}{\Rightarrow} \alpha x \beta$

Elimination of useless symbols include identifying whether or not the symbol is "generating" and "reachable".

- A symbol x is generating if $x \overset{*}{\Rightarrow} w$ for some terminal string w.
- A symbol x is reachable if there is derivation $S \overset{*}{\Rightarrow} \alpha x \beta$ for some α and β.

Thus we eliminate non-generating and then non-reachable symbols.

Example:

Consider a grammar: $S \to aB \mid bX$

$A \to Bad \mid bSX \mid a$

$B \to aSB \mid bBX$

$X \to sBd \mid aBX \mid ad$

Here,

- A and X can generate terminals, so A and X are generating symbols as $A \to a$ and $X \to ad$
- Also S is generating symbol as S can generate terminal string such as $S \to bX$ and X generate terminal string
- But B cannot produce any terminal symbol, so it is non-generating

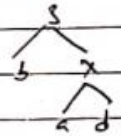
Thus, after eliminating B, we get: $S \to bX$

$A \to bSX \mid a$

$X \to ad$

Again,

A is unreachable as there is no any derivation of form $S \Rightarrow \alpha A \beta$ in grammar.



After eliminating A $S \to bX$

$X \to ad$

∴ This is the simplified grammar

Sample 1:

Consider a CFG, $S \to AB \mid a$, $A \to b$. Identify & eliminate useless symbols

→ Here, B is non-generating symbol since B is not deriving any string w in the V^*. So we eliminate $S \to AB$

So, CFG becomes:

$S \to a$

$A \to b$

Again A is non reachable symbol from the starting symbol S

Thus it is to be eliminated.

Hence CFG becomes: $S \to a$

Sample 2:

Consider below grammar and obtain an equivalent grammar containing no useless grammar symbol. $A \to xyz \mid X_{yzz}$

$X \to Xz \mid xYz$

$Y \to aYy \mid Xz$

$Z \to Zy \mid z$

Here, $A \to xyz$ and $Z \to z$, hence A and Z are directly deriving to the string of terminals. So they are useful symbols.

But X and Y do not lead to a string of terminals. So X and Y are useless symbols.

After eliminating these productions, we get:

$A \to xyz$

$Z \to Zy \mid z$

But Z is not reachable from the starting non-terminal A,

So, we remove Z.

Thus. $A \to xyz$ is the required CFG

2. Removal of Empty production

- A grammar is said to have empty production if there is a production of the form $A \to \epsilon$.
- Here, we have to discover nullable variable.
- A variable 'A' is nullable if $A \overset{+}{\Rightarrow} \epsilon$
- If $A \to \epsilon$ is a production to be eliminated, then we look for all productions whose right side contains A and replace each occurance of A in each of these production to obtain a non-empty production.
- Now these resultant non-empty productions must be added to grammar to keep the language generated to be same.
- Example:

Consider CFG as: $S \to aA$

$A \to b \mid \epsilon$

Sol:

Here $A \to \epsilon$ $\therefore$ A is nullable

Now,

find all productions whose right side contain A.

Here, $S \to aA$

$\to a\epsilon$ after replacing A with ϵ

$\to a$

Finally we get: $S \to aA$

$A \to b$

$S \to a$

Equivalently,

$S \to aA \mid a$

$A \to b$ is required CFG.

Sample 1:

Consider the following Grammar G: $S \to ABAC$

$A \to aA \mid \epsilon$

$B \to bB \mid \epsilon$

remove the ϵ production from the above grammar $C \to c$

Sol: 1) $A \to a\epsilon$ and $B \to b\epsilon$ are nullable productions.

So remove them.

2) Find all productions with A or B on right side.

$S \to ABAC$

$A \to aA$

On replacing each occurance of A by ϵ (one by one), we get four new production to be added to the grammar:

$S \to BAC \mid ABC \mid BC$

$A \to a$

Add these production to the grammar and eliminate $A \to \epsilon$, Obtained grammar is:

$S \to ABAC \mid BAC \mid ABC \mid BC$

$A \to aA \mid a$

$B \to bB \mid \epsilon$

$C \to c$

3) Find all the production with B on right side.

$S \to ABAC \mid BAC \mid ABC \mid BC$

$B \to bB$

on replacing each occurance of B by ϵ (one by one) we obtain

$S \to AAC \mid AC \mid C$

$B \to b$

Add these productions to the grammar and eliminate $B \to \epsilon$, Obtained grammar is:-

$S \to ABAC \mid BAC \mid ABC \mid BC \mid AAC \mid AC \mid C$

$A \to aA \mid a$

$B \to bB \mid b$

$C \to c$

This is the required grammar without ϵ-production.

3. Removal of unit production

- A unit production is a production of the form $A \to B$ where A and B are both variables.
- Unit production increases the cost of derivation in a grammar, hence needs to be removed.
- Algorithm:

while (there exists unit production $A \to B$)

{ i) Select a unit production $A \to B$ such that there exists a production $B \to \alpha$, where α is terminal

ii) for (every non-unit production, $B \to \alpha$)

- add production $A \to \alpha$ to grammar

(iii) eliminate $A \to B$ from grammar

}

Example: $A \to B$

$B \to a$

$B \to c$

Sol: Here, Unit production is $A \to B$

Handling the unit production, we try to derive generate & replace for each non-terminals involved in the unit production as:

$A \to B$ generate $A \to a$

$A \to B$ generate $A \to c$

Then new CFG is:

$A \to a \mid c$

with no unit production

3.12 Numerical:

Identify and remove the unit production from following grammar:

$S \to A \mid bb$

$A \to B \mid b$

$B \to S \mid a$

Note: Capital letter are NT i.e. Non-terminals

Sol: Here unit productions are: $S \to A$

$A \to B$

$B \to S$

Handling one unit production at a time, we try to prove generating and reachable for each non-terminals involved in the unit productions as follows:-

$S \to A$	generate $S \to b$
$S \to A \to B$	generate $S \to a$
$A \to B$	generate $A \to a$
$A \to B \to S$	generate $A \to bb$
$B \to S$	generate $B \to bb$
$B \to S \to A$	generate $B \to b$

Then new CFG becomes:

$S \to bb \mid b \mid a$

$A \to b \mid a \mid bb$

$B \to a \mid bb \mid a$ which has no unit productions

Final CFG $\Rightarrow$:

$S \to bb \mid b \mid a$

Consider the context free grammar G: $S \to AB$

$A \to a$

$B \to C \mid b$

$C \to D$

$D \to E$

$E \to a$

Sol: Unit productions are: $B \to C$

$C \to D$

$D \to E$

To remove unit production $B \to C$, we try to prove generating and ~~non-generating~~ reachable for each non-terminals.

$B \to C \to D \to E$ generate $B \to a$

Similarly for $C \to D$,

$C \to D \to E$ generate $C \to a$

Similarly for $D \to E$,

$D \to E$ generate $E \to a$

Hence, required CFG without unit production is:

$S \to AB$	Final CFG is:	$S \to AB$
$A \to a$		$A \to a$
$B \to a \mid b$		$B \to a \mid b$
$C \to a$	$\Rightarrow$	
$D \to a$		
$E \to a$		

3.13 Normal forms

1. Chomsky Normal Form (CNF)
2. Greibach Normal Form (GNF)

3.14 Chomsky Normal Form (CNF)

A CFG $G = (V, \Sigma, R, S)$ is said to be in Chomsky Normal Form (CNF) if every production in G are in one of the two forms:

$A \rightarrow BC$

$A \rightarrow a$

where, $\{A, B, C\} \in V$ and $a \in \Sigma$

- So a grammar in CNF is one which should not have :-
  - i) ϵ production
  - ii) Unit production
  - iii) Useless symbols.

3.15 Numerical

Q.1. Convert the following CFG into CNF

$G = (V, \Sigma, R, S)$ where $V = \{S, A\}$

$\Sigma = \{a, b\}$

$R = \{ S \rightarrow aAB \mid AaB \mid B$

$A \rightarrow aA \mid \epsilon$

$B \rightarrow ab \mid bA$

$\}$

Sol: To convert CFG into CNF, we have to simplify CFG.

Step 1: Removal of useless symbols

Here, there are no any useless symbols.

Step 2: Removal of empty (ϵ) production

Here, $A \rightarrow \epsilon$ and B is nullable production.

Replacing each occurance of A by ϵ, we get:

$S \rightarrow aAB \mid AaB \mid aB \mid B$

$A \rightarrow aA \mid a$

$B \rightarrow ab \mid bA \mid Ab$

Step 3: Removal of unit production

Here, unit productions are:- $S \rightarrow B$

To remove unit production, $S \rightarrow B$,

$S \rightarrow B$ generate $S \rightarrow ab \mid bA$

So, $S \rightarrow aAB \mid AaB \mid aB \mid ab \mid bA \mid b$

$A \rightarrow aA \mid a$

$B \rightarrow ab \mid bA \mid b$

Step4: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals.

$S \to aAB \mid ACB \mid CB \mid CD \mid DA \mid b$

$C \to a$

$D \to b$

$A \to CA \mid a$

$B \to CD \mid DA \mid b$

consider $C \to a$, $D \to b$

single exist ~~so~~ change

Like $S \to b$

Step 5: ~~Here again~~ Eliminate RHS with more than two terminals

Here again $S \to CAB$ and $S \to ACB$ are not in required form

So, we can perform as below:

$S \to EB \mid FB \mid CB \mid CD \mid DA \mid b$

$E \to CA$

$F \to AC$

$C \to a$

$D \to b$

$A \to CA \mid a$

$B \to CD \mid DA \mid b$

Tutorial:

Convert below CFG into CNF

$S \to AAC$

$A \to aAb \mid \epsilon$

$C \to aC \mid a$

This is the required CNF.

Ans: $S \to AC \mid DC \mid a$

$A \to EO \mid DB$

$C \to DC \mid a$

$D \to a$

$B \to b$

$E \to DA$

3.16 Greibach Normal Form

- A grammar $G = (V, \Sigma, R, S)$ is said to be in GNF if all the production of grammar are in the form:

$A \to a\alpha$

$A \to a$

where,

$a \in \Sigma$ and α is a string of non-terminals (or variables)

- Eg:

$S \to aAB \mid bBB \mid bB$

$A \to aA \mid bB \mid b$

$B \to b$ is in GNF

- GNF are used to construct a push-down automata (PDA)

Steps for converting CFG into GNF:

1. Convert the grammar into CNF
2. If the grammar exists left recursion, eliminate it
3. In the grammar, convert the given production rule into GNF form.

Note:

Left Recursion

- A production is said to have left recursion if the leftmost variable of its RHS is same as the variable in its LHS.

Eg: $S \to Sa \mid \epsilon$ is left recursion grammar

$S \to \epsilon C$

$C \to aC \mid \epsilon$

Eliminate Left Recursion

$A \to A\alpha \mid \beta$

$\Downarrow$

$A \to \beta C$

$C \to \alpha C \mid \epsilon$

3.17 Numerical

1. Convert the grammar $S \to AB \mid BC$, $A \to aB \mid bA \mid a$, $B \to bB \mid cC \mid b$, $C \to c$ into GNF

Sol^n:

The production $S \to AB \mid BC$ is not in GNF

On applying the substitution rule we immediately get required GNF

$S \to aBB \mid bAB \mid aB \mid bBC \mid cCC \mid bC$

$A \to aB \mid bA \mid a$

$B \to bB \mid cC \mid b$

$C \to c$

Here the grammar is not in CNF

$A \to aB \mid bA \mid a$

$B \to AB \mid CC \mid b$

But it is GNF.

So, we simply try to transform $S \to AB \mid BC$ into CNF

If we convert it into CNF, we would get:-

$S \to AB \mid BC$

$A \to DB \mid EA \mid a$

$B \to EB \mid CC \mid b$

$C \to c$, $D \to a$, $E \to b$

2. Convert the grammar $S \to aba \mid Sa \mid ab$ into GNF

Sol^n: If we introduce new variables A and B and productions as

$A \to a$

$B \to b$ and substitute into given grammar, we obtain

$S \to aBASA \mid aBA$

$A \to a$

$B \to b$

This is in GNF

3. Convert the grammar $S \to XB \mid AA$, $A \to a \mid SA$, $B \to b$, $X \to a$ into GNF

Sol^n: The given grammar G is already in CNF.

i) The production rule $A \to SA$ is not in GNF.

Substitute $S \to XB \mid AA$ in production rule A. $A \to SA$

$S \to XB \mid AA$

$A \to a \mid XBA \mid AAA$

$B \to b$ is in GNF

$X \to a$ ✓ is in GNF

ii) The production rule $S \to XB$ and $A \to XBA$ is not in GNF

Substitute $X \to a$ in production rule $S \to XB$ and $B \to XBA$

$S \to aB \mid AA$ ∵ $X \to a$

$A \to a \mid aBA \mid AAA$ ∵ $X \to a$

Now in GNF

$B \to b$

$X \to a$

iii) Remove left recursion $(A \to AAA)$, we get.

$S \to aB \mid AA$

$A \to aC \mid aBAC$ $A \to aE \mid aBAC$

$C \to AAC \mid \epsilon$ $\to a \mid aBA$

$B \to b$

$X \to a$

$A \to AAA \mid a$

$A \to aC$

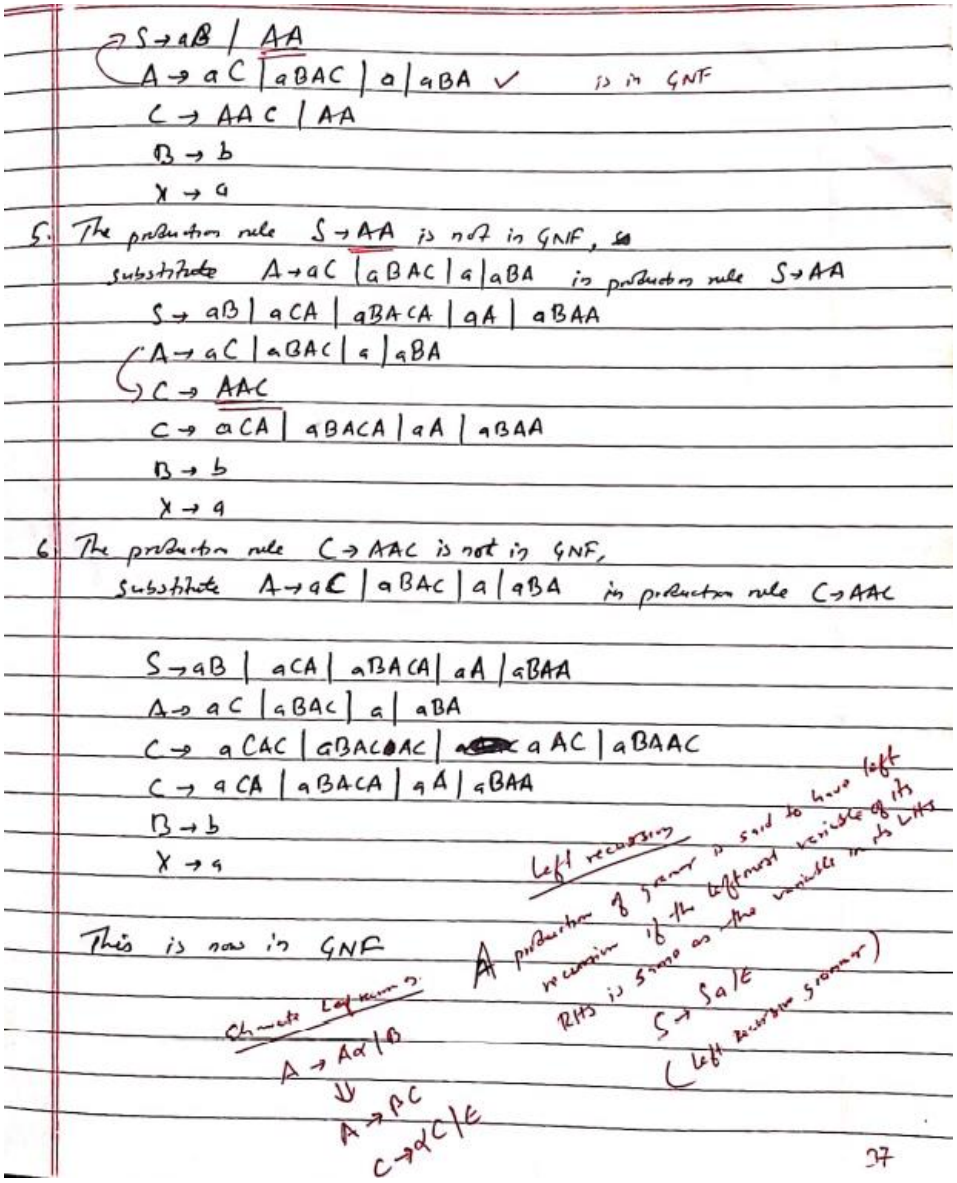
$C \to AAC \mid \epsilon$

$A \to AAA \mid aBA$

$A \to aBAC$

$C \to AAC \mid \epsilon$

iv) Remove null production. $C \to \epsilon$



End of Chapter 3

4.1 Pushdown Automata

- Pushdown automata is a way to implement a CFG in the same way we design DFA for a regular grammar.
- A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.
- Pushdown automata is simply an NFA augmented with an "external stack memory". The addition of stack is used to provide a last-in-first-out memory management capability to Pushdown automata. Pushdown automata can store an unbounded amount of information on the stack. It can access a limited amount of information on the stack. A PDA can push an element onto the top of the stack and pop off an element from the top of the stack. To read an element into the stack, the top elements must be popped off and are lost.
- A PDA is more powerful than FA. Any language which can be acceptable by FA can also be acceptable by PDA. PDA also accepts a class of language which even cannot be accepted by FA. Thus, PDA is much more superior to FA.

PDA Components:

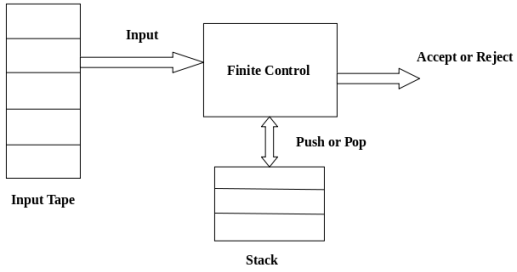


Fig: Pushdown Automata

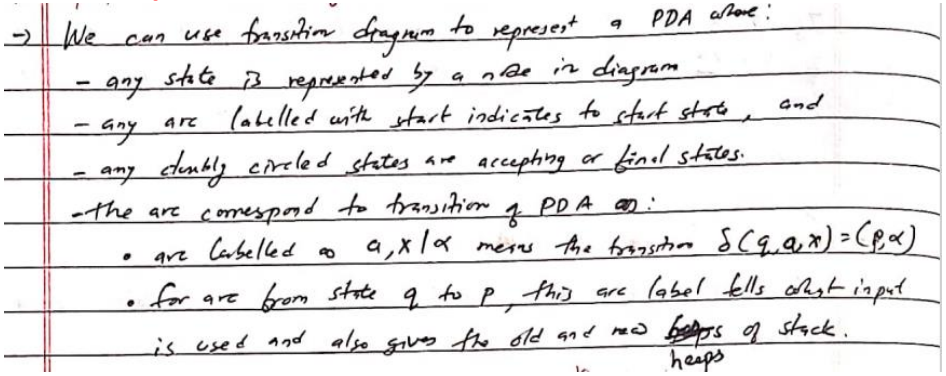
- **Input tape:** The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time.
- **Finite control:** The finite control has some pointer which points the current symbol which is to be read.
- **Stack:** The stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, the stack is used to store the items temporarily.

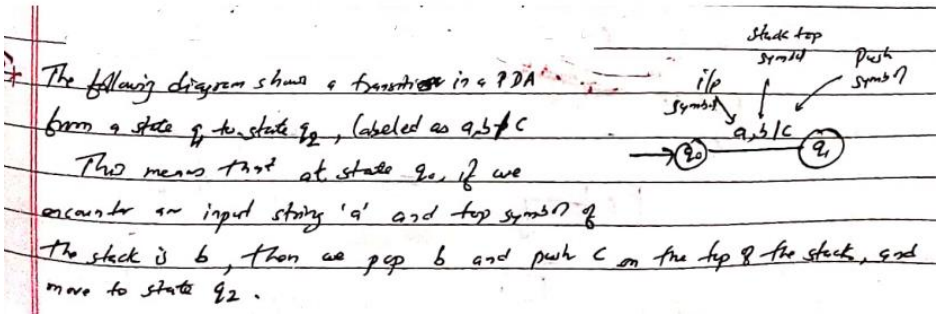
4.2 Formal definition of PDA

The PDA can be defined as a collection of 7 components:

Q : the finite set of states
∑ : the input set
Γ : a stack symbol which can be pushed and popped from the stack
q0 : the initial state
Z : a start symbol which is in Γ.
F : a set of final states
Δ : mapping function which is used for moving from current state to next state. (Q x ∑* x Γ *) → (Q x Γ *)

4.3 Graphical notation of PDA





The following diagram shows a transition in a PDA from a state q_0 to state q_2, labeled as a,b/c

This means that at state q_0, if we encounter an input string 'a' and top symbol of the stack is b, then we pop b and push c on the top of the stack, and move to state q_2.

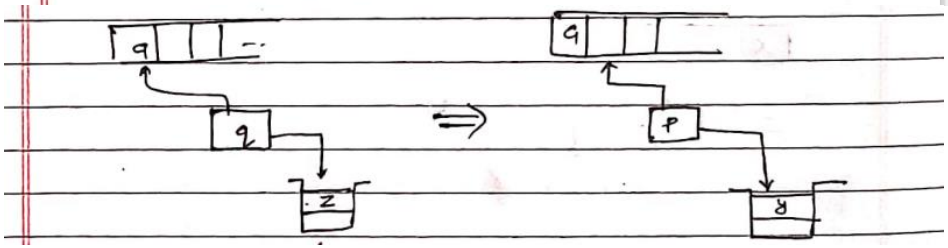
4.4 Moves of PDA

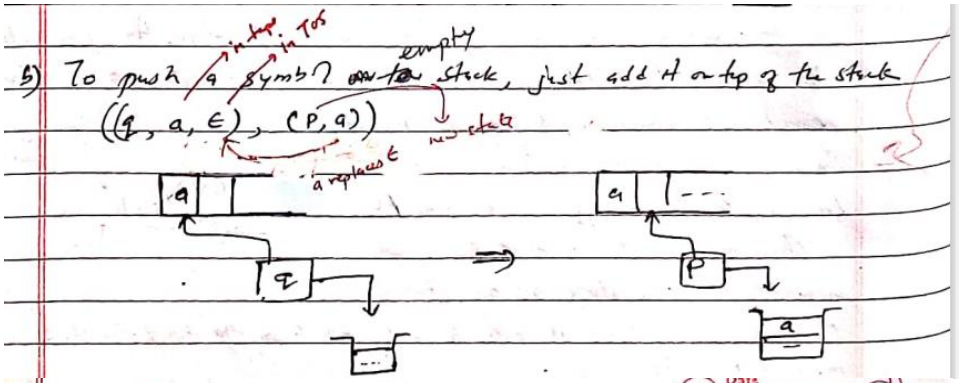
Let us see some moves of PDA in the form of mapping, that

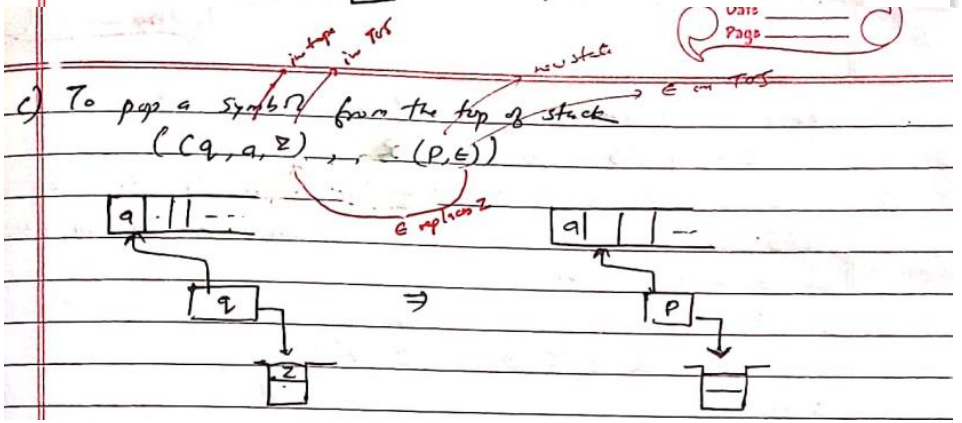
$$(Q \times \Sigma^*, \Gamma^*) \rightarrow (Q \times \Gamma^*)$$

a) Interpretation of $((q, a, z), (p, y)) \in \delta$

- PDA is in q state
- If z is on top of stack, and input is a, replace z by y on the top of the stack and enter state P







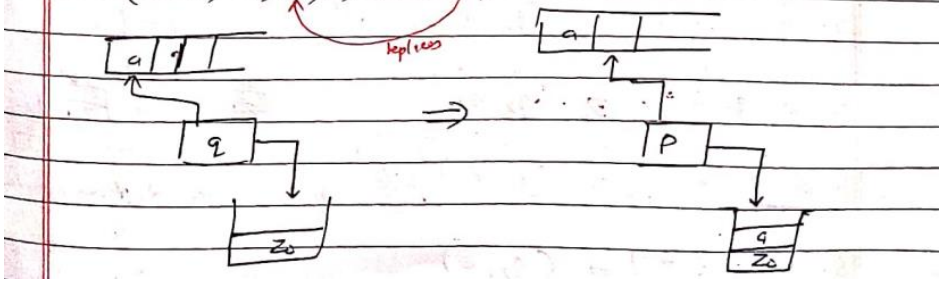
d) PDA also behaves as do nothing machine, just read the input from the input and don't make any change in the state and symbol at the stack

$((p, a, \epsilon), (p, \epsilon))$ a, ε | ε

At this case, PDA just reads input a and without making any change, the stack or at stack top, waits for another input from input tape.

e) To push a symbol on the top of stack (which is not empty stack) just add it on top of the stack.

$(q, a, Z_0) ; (p, aZ_0)$



4.5 Numerical

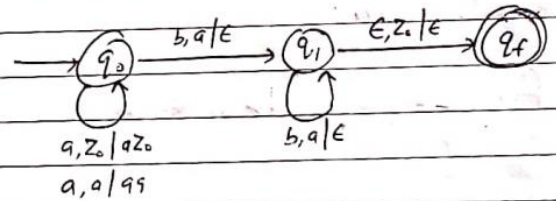
1. Design a PDA that accepts $L = \{a^n b^n \mid n>1\}$

By analysis it is clear that:

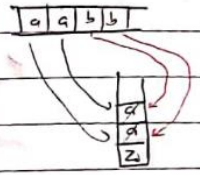
PDA accepts ab, aabb, aaabbb, - - - ..

PDA rejects ϵ, a, b, aab, aaabb, bbaa, ...

Let us design the PDA with the state diagram.



Try to design for aabb



Hence, the PDA is:-

$P = \{Q, \Sigma, \tau, \delta, q_0, Z_0, F\}$

Where,

$Q = \{q_0, q_1, q_f\}$

$\Sigma = \{a, b\}$

$\tau = \{a, b, Z_0\}$

$q_0 = q_0$

$Z_0 = Z_0$

$F = \{q_f\}$

Transition δ is given by:-

1. $\delta(q_0, a, Z_0) \rightarrow (q_0, Z_0)$
2. $\delta(q_0, a, a) \rightarrow (q_0, aa)$
3. $\delta(q_0, b, a) \rightarrow (q_1, \epsilon)$
4. $\delta(q_1, b, a) \rightarrow (q_1, \epsilon)$
5. $\delta(q_1, \epsilon, Z_0) \rightarrow (q_f, \epsilon)$

Now take the string aaabbb

S.NO	State	String	Stack	Transition used
1.	q_0	aaabbb	Z_0	-
2.	q_0	aabbb	aZ_0	1
3.	q_0	abbb	aaZ_0	2
4.	q_0	bbb	$aaaZ_0$	2
5.	q_1	bb	aaZ_0	3
6.	q_1	b	aZ_0	4
7.	q_1	ϵ	Z_0	4
8.	q_f	ϵ	ϵ	5

∴ Given string is accepted.

Take another string abb that should be rejected.

S.No.	state	string	Stack	Transition used
1	q_0	abb	Z_0	—
2	q_0	bb	aZ_0	1
3	q_1	b	Z_0	4
4.	Cannot move forward.			

Hence the string is rejected.

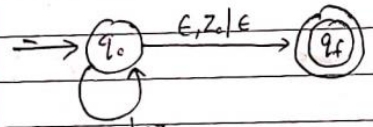
2. Design a PDA for that accepts:
$L = \{ w \mid w \in \{a,b\}^* \}$ and a's and b's are equal.

Soln: By analysis it is clear that the PDA accepts any combination of a and b where the no of a's and b's are equal.

PDA accepts : aabb, ab, abab, abbaba, ...

PDA rejects : a, b, abb, bba, aba, bab, abbab, ...

Now,
Let us design the PDA as:-



a, Z0 | aZ0
b, Z0 | bZ0
a, a | aa
b, b | bb
a, b | ε
b, a | ε

Try to design for

a	a	b	b

a	b	a	b

b	a	b	a

Hence, the PDA is:

$P = \{ Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \}$

Where,

$Q = \{q_0, q_f\}$

$\Sigma = \{a, b\}$

$\Gamma = \{a, b, Z_0\}$

$q_0 = \{q_0\}$

$Z_0 = Z_0$

$F = \{q_f\}$

and transition (δ) is given by:-

1) $\delta(q_0, a, Z_0) \rightarrow (q_0, aZ_0)$
2) $\delta(q_0, b, Z_0) \rightarrow (q_0, bZ_0)$
3) $\delta(q_0, a, a) \rightarrow (q_0, aa)$
4) $\delta(q_0, b, b) \rightarrow (q_0, bb)$
5) $\delta(q_0, a, b) \rightarrow (q_0, \epsilon)$
6) $\delta(q_0, b, a) \rightarrow (q_0, \epsilon)$
7) $\delta(q_0, \epsilon, Z_0) \rightarrow (q_f, \epsilon)$

Let us check for the string abbaab

Step	state	string	stack	Transition used
1.	q_0	abbaab	Z_0	—
2.	q_0	bbaab	aZ_0	1
3.	q_0	baab	Z_0	6
4.	q_0	aab	bZ_0	2
5.	q_0	ab	Z_0	5
6.	q_0	b	aZ_0	1
7.	q_0	ϵ	Z_0	6
8.	q_f	ϵ	ϵ	7

Here, the PDA enters q_f state and stack is empty. Hence, the string is accepted by PDA

Also let us check another string abaab that should be rejected.

step	state	string	stack	Transition used
1.	q_0	abaabb	Z_0	—
2.	q_0	baabb	aZ_0	1
3.	q_0	aabb	Z_0	6
4.	q_0	abb	aZ_0	1
5.	q_0	bb	Z_0	6
6.	q_0	b	bZ_0	2
7	cannot move forward.			

Here, PDA does not enter q_f state and also stack is not empty.

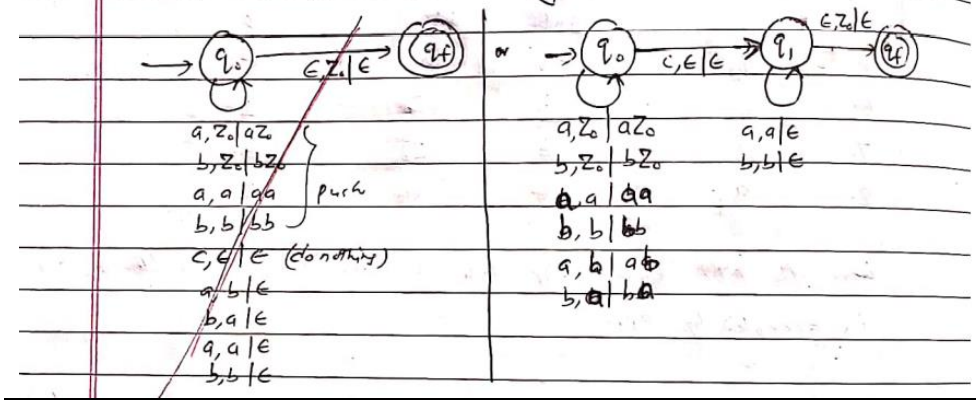
Hence, the string abaabb is not accepted by our PDA.

3. Design for $L = \{wcw^r \mid w \in \{a,b\}^*\}$

Sol^n: Accepted string: abcba, ababcbabа, acа, ...

Rejected string: abcab, abca, cbc, ...

Design of state diagram:



Hence, the req PDA is: $P = \{Q, \Sigma, \tau, \delta, q_0, Z_0, F\}$

Where,

$Q = \{q_0, q_1, q_f\}$

$\Sigma = \{a, b\}$

$\tau = \{a, b, Z_0\}$

$q_0 = \{q_0\}$

$Z_0 = \{Z_0\}$

$F = \{q_f\}$

Transition rules (δ) is given by:-

1) $(q_0, a, Z_0) \rightarrow (q_0, aZ_0)$
2) $(q_0, b, Z_0) \rightarrow (q_0, bZ_0)$
3) $(q_0, a, a) \rightarrow (q_0, aa)$
4) $(q_0, b, b) \rightarrow (q_0, bb)$
5) $(q_0, a, b) \rightarrow (q_0, ab)$
6) $(q_0, b, a) \rightarrow (q_0, ba)$
7) $(q_0, c, \epsilon) \rightarrow (q_1, \epsilon)$
8) $(q_1, a, a) \rightarrow (q_0, \epsilon)$
9) $(q_1, b, b) \rightarrow (q_0, \epsilon)$
10) $(q_1, \epsilon, Z_0) \rightarrow (q_f, \epsilon)$

Check for abbcbba

S.No	state	string	stack	Transition used
1	q_0	abbcbba	Z_0	—
2.	q_0	bbcbba	aZ_0	1
3.	q_0	bcbba	baZ_0	6
4.	q_0	cbba	$bbaZ_0$	4
5.	q_1	bba	$bbaZ_0$	7
6.	q_1	ba	baZ_0	9
7.	q_1	a	aZ_0	8
8.	q_1	ϵ	Z_0	8
9.	q_f	ϵ	ϵ	10

Since, we reached q_f (final state) and the stack is empty, the PDA D accepted.

Check for abca

S.No.	State	String	Stack	Transition
1.	q_0	abca	Z_0	—
2.	q_0	bca	aZ_0	1
3.	q_0	ca	baZ_0	6
4.	q_0	a	baZ_0	7
5.	No further movement.			

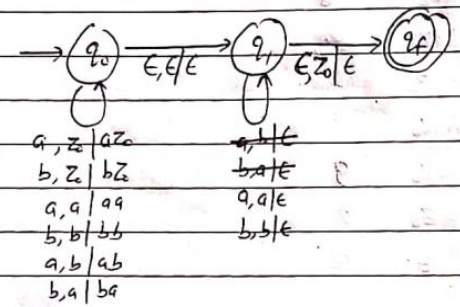
So, PDA rejects this string.

4. Design PDA for $L = \{ ww^R : w \in \{a,b\}^+ \}$

Accepted string: aεa, bεb, abbεbba, babεbab, aababεbaaba, ...

Rejected string: abεab, bεba, baεbab, abεbab, ...

PDA design with state diagram:



Here req^d PDA is:- $P = \{Q, \Sigma, \tau, \delta, Z_0, F, q_0\}$

Where,

$Q = \{q_0, q_1, q_f\}$

$\Sigma = \{a, b\}$

$\tau = \{a, b, Z_0\}$

$q_0 = \{q_0\}$

$F = \{q_f\}$

$Z_0 = \{Z_0\}$

Transition rule (δ) is given by:-

1) $(q_0, a, Z_0) \to (q_0, aZ_0)$
2) $(q_0, b, Z_0) \to (q_0, bZ_0)$
3) $(q_0, a, a) \to (q_0, aa)$
4) $(q_0, b, b) \to (q_0, bb)$
5) $(q_0, a, b) \to (q_0, ab)$
6) $(q_0, b, a) \to (q_0, ba)$
7) $(q_0, \epsilon, \epsilon) \to (q_1, \epsilon)$
8) $(q_1, a, a) \to (q_1, \epsilon)$
9) $(q_1, b, b) \to (q_1, \epsilon)$
10) $(q_1, \epsilon, Z_0) \to (q_f, \epsilon)$

Test for string aabεbaa (which should be accepted)

Step	State	String	Stack	Transition used
1.	q_0	aabεbaa	Z_0	—
2.	q_0	abεbaa	aZ_0	1
3.	q_0	bεbaa	aaZ_0	3
4.	q_0	εbaa	$baaZ_0$	6
5.	q_1	baa	$baaZ_0$	7
6.	q_1	aa	aaZ_0	~~9~~
7.	q_1	a	aZ_0	~~9~~
8.	q_1	ε	Z_0	~~9~~
9.	q_f	ε	ε	~~10~~

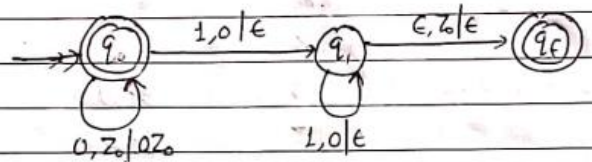
Hence, PDA accepts aabεbaa

5) Construct a PDA that accepts $L = \{0^n 1^n \mid n \geq 0\}$

PDA accepts: ϵ, 01, 0011, 000111, ...

PDA rejects: 0, 1, 00, 001, 110, 0101, 1010, ...

PDA design with state diagram:



Hence, PDA is:-

$P = \{Q, \Sigma, \tau, Z_0, q_0, F, \delta\}$

Where,

$Q = \{q_0, q_1, q_f\}$

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

$\tau = \{0, 1, Z_0\}$

$q_0 = \{q_0\}$

$Z_0 = \{Z_0\}$

$F = \{q_0, q_f\}$

Transition rule(δ) given by:-

1. $(q_0, 0, Z_0) \rightarrow (q_0, 0Z_0)$
2. $(q_0, 0, 0) \rightarrow (q_0, 00)$
3. $(q_0, \epsilon, Z_0) \rightarrow (q_0, \epsilon)$
4. $(q_0, 1, 0) \rightarrow (q_1, \epsilon)$
5. $(q_1, 1, 0) \rightarrow (q_1, \epsilon)$
6. $(q_1, \epsilon, Z_0) \rightarrow (q_f, \epsilon)$

Test for ϵ

Step	State	String	Stack	Transition
1.	q_0	ϵ	Z_0	—
2.	q_0	ϵ	ϵ	3

∴ PDA accepts ϵ

Test for 0011

Step	State	String	Stack	Transition
1.	q_0	0011	Z_0	—
2.	q_0	011	$0Z_0$	1
3	q_0	11	$00Z_0$	2
4	q_1	1	$0Z_0$	4
5	q_1	ϵ	Z_0	5
6.	q_f	ϵ	ϵ	6

∴ PDA accepts 0011

Test for 00111

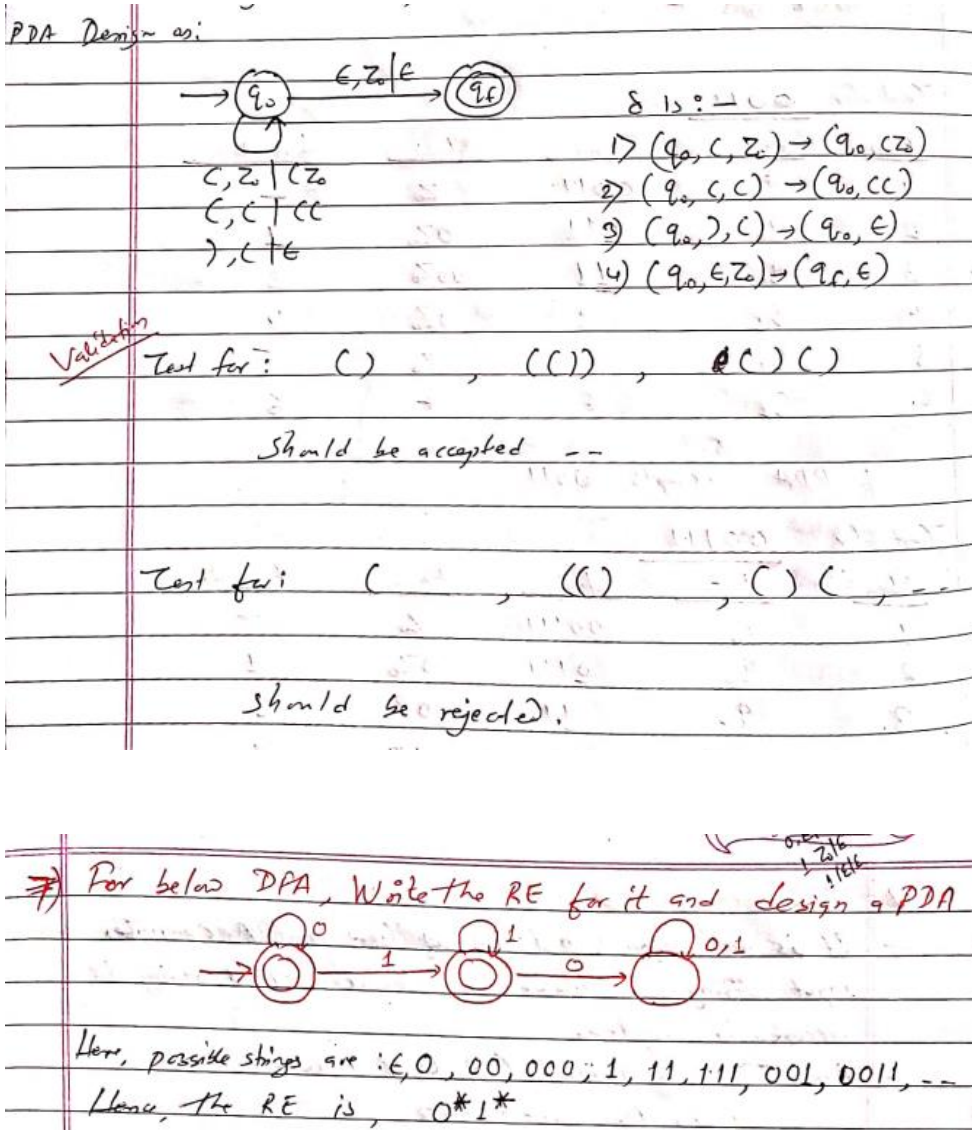
Step	State	String	Stack	Transition
1	q_0	00111	Z_0	—
2.	q_0	0111	$0Z_0$	1
3.	q_0	111	$00Z_0$	2
4.	q_1	11	$0Z_0$	4
5.	q_1	1	Z_0	5
6.	No movement			

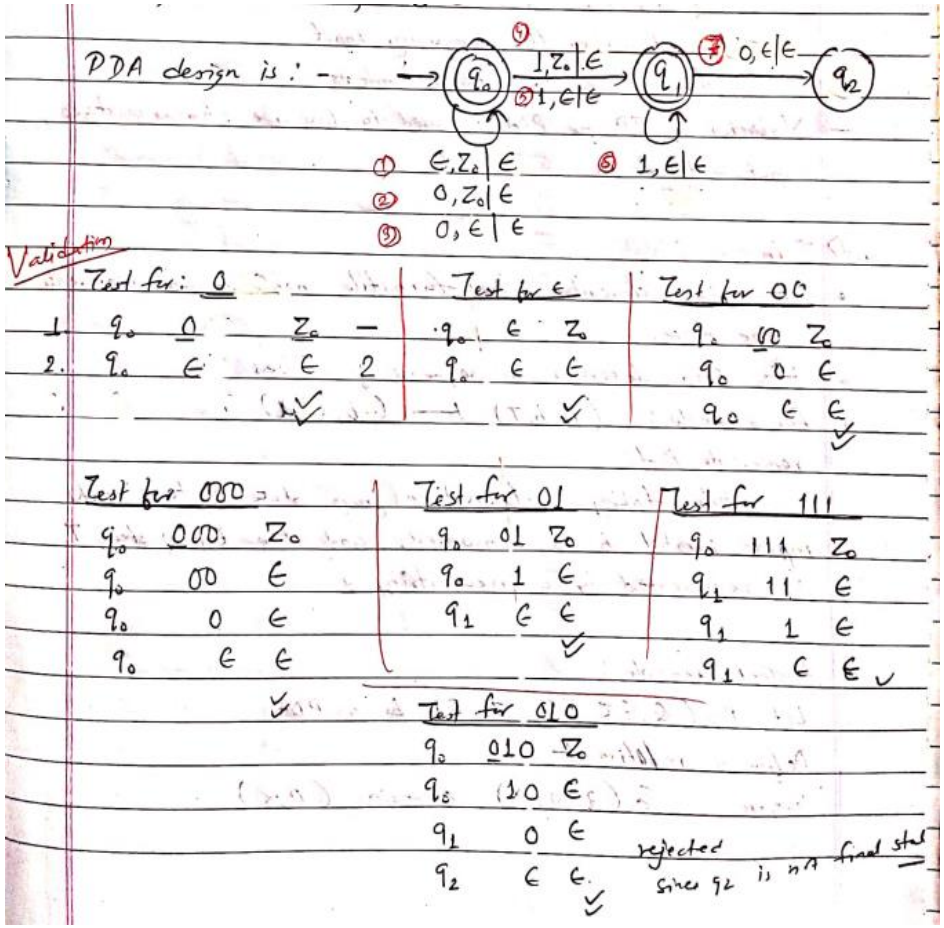
∴ PDA rejects 00111

6. Design a PDA that accepts proper parenthesis:
ie. (), ()(), (()), ()()(), ((())), ...

PDA accepts: (), ()(), (()), ((())), ...

rejects:)(, (,), (()), ()),)(, ((), ...





4.6 Instantaneous Description (ID) of PDA

- ID is an informal notation of how a PDA computes an input string and make a decision that string is accepted or rejected.
- Notation of ID for PDA is used to describe changes in state, input and stack.

- An instantaneous description is a triple (q, w, α) where:

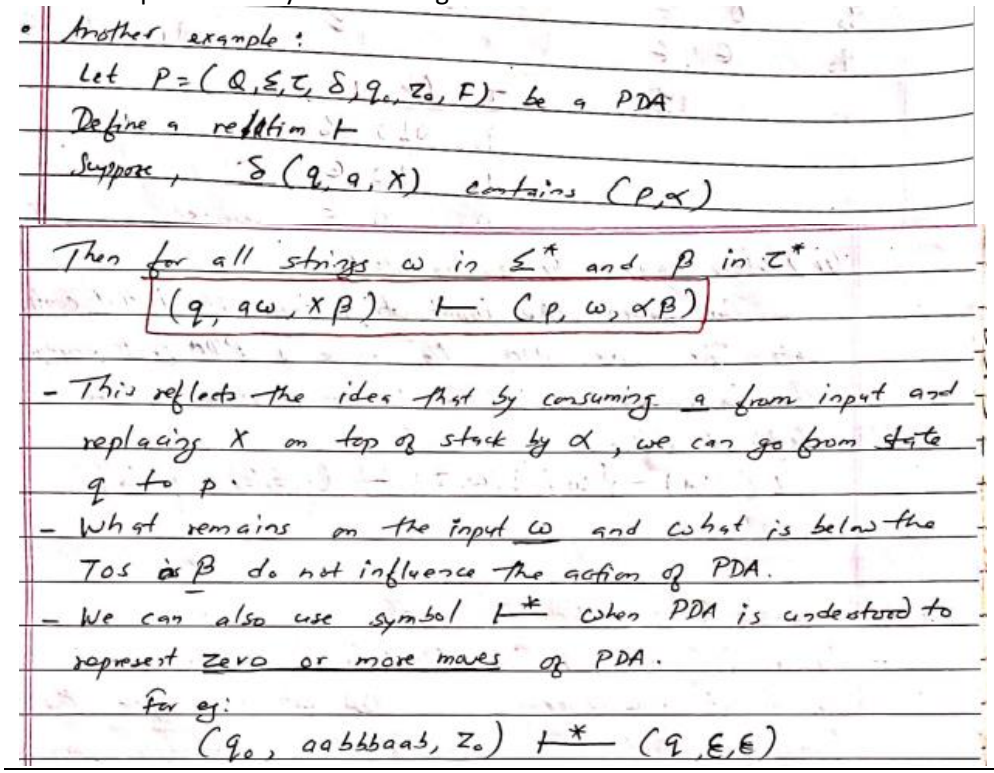
- q describes the current state.
- w describes the remaining input.
- α describes the stack contents, top at the left.

4.7 Turnstile Notation

- ⊢ sign describes the turnstile notation and represents one move.
- ⊢* sign describes a sequence of moves.
- For example,

  (p, b, T) ⊢ (q, w, α)

  In the above example, while taking a transition from state p to q, the input symbol 'b' is consumed, and the top of the stack 'T' is represented by a new string α.



4.8 PDA Acceptance

A language can be accepted by Pushdown automata using two approaches:

1. Acceptance by Final State:

   The PDA is said to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

   Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by the final state can be defined as:

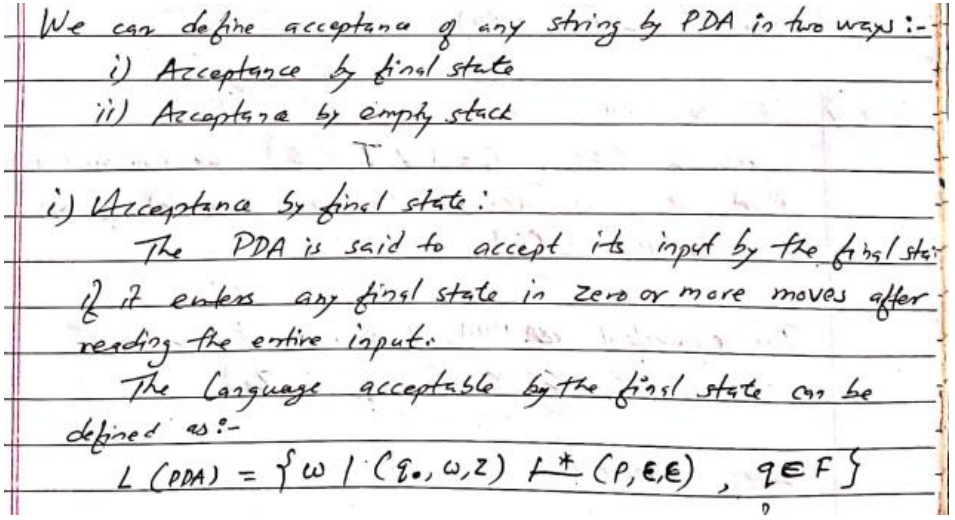
   L(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ F}

2. Acceptance by Empty Stack:

   On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

   Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by empty stack can be defined as:

   N(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ Q}



(ii) Acceptance by empty stack

On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty. The language acceptable by empty stack can be defined as:-

$$L(PDA) = \{\omega \mid (q, \omega, z) \vdash^* (p, \varepsilon, \varepsilon),\ q \in Q\}$$

4.9 Equivalence of PDA and CFG

- If a grammar G is context free, we can build an equivalent non-deterministic PDA which accepts the language that is produced by the context-free grammar G.
- If P is a PDA, an equivalent CFG can be constructed where; $L(G) = L(P)$

- If L = N(P1) for some PDA P1, then there is a PDA P2 such that L = L(P2). That means the language accepted by empty stack PDA will also be accepted by final state PDA.
- If there is a language L = L (P1) for some PDA P1 then there is a PDA P2 such that L = N(P2). That means language accepted by final state PDA is also acceptable by empty stack PDA.

4.10 CFG to PDA Conversion

Given a CFG, $G = (V, T, P, S)$, we construct a PDA, P that accepts (language generated) by G
ie. $L(P) = L(G)$.

The equivalent PDA, P can be defined as:-

$$P = (Q, \Sigma, \tau, \delta, q_0, F, Z_0)$$

(where,

$Q = \{q\}$

$\Sigma = T$

$\tau = V \cup T$

$Z_0 = S$

$F = q$

$q_0 = q$

and

δ can be defined as:-

i) $\delta(q, \varepsilon, A) = \{(q, \alpha)\}$ for each production rule $A \to \alpha$ in CFG

ii) $\delta(q, a, a) = \{(q, \varepsilon)\}$ for each $a \in T$ in CFG.

4.11 Numerical

1. Convert the following grammar to a PDA that accepts the same language. Given $G = (V, T, P, S)$ where P is defined as:-

$S \to aSa$

$S \to bSb$

$S \to c$

Sol^n: Let PDA be: $P = \{Q, \Sigma, \tau, \delta, z_0, q_0, F\}$

where $Q = \{q\}$

$\Sigma = \{a, b, c\}$

$\tau = \{S, a, b, c\}$

$q_0 = \{q\}$

$F = \{q\}$

$z_0 = \{S\}$

Here, $V = \{S\}$

$T = \{a, b, c\}$

$S = \{S\}$

and δ is defined as:-

1) $\delta(q, \varepsilon, S) \to (q, aSa)$ — For P rule
2) $\delta(q, \varepsilon, S) \to (q, bSb)$ — For P rule
3) $\delta(q, \varepsilon, S) \to (q, c)$ — For P rule
4) $\delta(q, a, a) \to (q, \varepsilon)$ — For terminals
5) $\delta(q, b, b) \to (q, \varepsilon)$ — For terminals
6) $\delta(q, c, c) \to (q, \varepsilon)$ — For terminals

Now, Let us test for abbcbba which is accepted by CFG also should be accepted by our PDA.

(state, input, stack)

$(q, abbcbba, S) \vdash (q, abbcbba, aSa)$ by rule 1
$\vdash (q, bbcbba, Sa)$ by rule 4
$\vdash (q, bbcbba, bSba)$ by rule 2
$\vdash (q, bcbba, Sba)$ by rule 5
$\vdash (q, bcbba, bSbba)$ by rule 2
$\vdash (q, cbba, Sbba)$ by rule 5
$\vdash (q, cbba, cbba)$ by rule 3
$\vdash (q, bba, bba)$ by rule 6
$\vdash (q, ba, ba)$ by rule 5
$\vdash (q, a, a)$ by rule 5
$\vdash (q, \varepsilon, \varepsilon)$ by rule 4

Hence PDA accepted by empty stack //

2. Design a PDA for the grammar $G = (V, T, P, S)$ where P is:

$E \to T \mid E+T$

$T \to F \mid T*F$

$F \to a \mid (E)$

Sol^n: Here, $G = \{V, T, P, S\}$ where

$V = \{T, E, F\}$

$T = \{+, *, (,), a\}$

$S = \{E\}$

$P = \{ E \to T \mid E+T$

$F \to T \mid T*F$

$F \to a \mid (E) \}$

Let the equivalent PDA be:

$P = \{Q, \Sigma, \tau, q_0, Z_0, F, \delta\}$

where,

$Q = \{q\}$

$\Sigma = \{+, *, (,), a\}$

$\tau = \{T, E, F, +, *, (,), a\}$

$q_0 = \{q\}$

$Z_0 = \{E\}$

$F = \{q\}$

and δ is given by:-

1) $\delta(q, \varepsilon, E) \to \{(q, T), (q, E+T)\}$ — For Non terminals
2) $\delta(q, \varepsilon, T) \to \{(q, F), (q, T*F)\}$ — For Non terminals
3) $\delta(q, \varepsilon, F) \to \{(q, a), (q, (E))\}$ — For Non terminals
4) $\delta(q, a, a) \to (q, \varepsilon)$ — For terminals
5) $\delta(q, +, +) \to (q, \varepsilon)$ — For terminals

6) $\delta(q, *, *) \rightarrow (q, \epsilon)$

7) $\delta(q, c, c) \rightarrow (q, \epsilon)$

8) $\delta(q,),)) \rightarrow (q, \epsilon)$

Now trace for a + (a*a)

Using CFG

$E \rightarrow E+T$
$\rightarrow T+T$
$\rightarrow F+T$
$\rightarrow a+T$
$\rightarrow a+E$
$\rightarrow a+(E)$
$\rightarrow a+(T)$
$\rightarrow a+(T*F)$
$\rightarrow a+(F*F)$
$\rightarrow a+(a*F)$
$\rightarrow a+(a*a)$

Using PDA (stack initialization → $q, a+(a*a), E$ ← input string)

$\vdash (q, a+(a*a), E+T)$ by rule 1
$\vdash (q, a+(a*a), T+T)$ by rule 1
$\vdash (q, a+(a*a), F+T)$ by rule 2
$\vdash (q, a+(a*a), a+T)$ by rule 3
$\vdash (q, +(a*a), +T)$ by rule 4
$\vdash (q, (a*a), T)$ by rule 5
$\vdash (q, (a*a), F)$ by rule 2
$\vdash (q, (a*a), (E))$ by rule 3
$\vdash (q, a*a), E))$ by rule 7
$\vdash (q, a*a), T))$ by rule 1
$\vdash (q, a*a), T*F))$ by rule 2
$\vdash (q, a*a), F*F))$ by rule 2
$\vdash (q, a*a), a+F))$ by rule 3
$\vdash (q, *a), *F))$ by rule 4
$\vdash (q, a), F))$ by rule 6
$\vdash (q, a), a))$ by rule 3
$\vdash (q,),))$ by rule 4
$\vdash (q, \epsilon, \epsilon)$ by rule 8

Thus PDA accepted by empty stack

3. Design a PDA for the grammar G, $S \rightarrow 0S1S$, $S \rightarrow 1S0S$, $S \rightarrow \epsilon$

Sol": Here, CFG is $G(G) = \{V, T, P, S\}$

where, $V = \{S\}$
$T = \{0, 1, \}$
$S = \{S\}$
$P = \{S \rightarrow 0S1S / 1S0S / \epsilon\}$

Let PDA is $P = \{Q, \Sigma, \tau, \delta, Z_0, S, F\}$

where, $Q = \{q\}$
$\Sigma = \{0, 1\}$
$\tau = \{0, 1, S\}$
$S = \{q\}$
$F = \{q\}$
$Z_0 = \{S\}$

δ is defined as:-

1) $\delta(q, \epsilon, S) \rightarrow (q, 0S1S)$
2) $\delta(q, \epsilon, S) \rightarrow (q, 1S0S)$
3) $\delta(q, \epsilon, S) \rightarrow (q, \epsilon)$
4) $\delta(q, 0, 0) \rightarrow (q, \epsilon)$
5) $\delta(q, 1, 1) \rightarrow (q, \epsilon)$
6) $\delta(q, \epsilon, \epsilon) \rightarrow (q, \epsilon)$

Let us test string for: 011001 which is accepted by CFG also should be accepted by our PDA.

$(q, 011001, S)$
$\vdash (q, 011001, 0S1S)$ by rule 1
$\vdash (q, 11001, S1S)$ by rule 4
$\vdash (q, 11001, 1S0S1S)$ by rule 2
$\vdash (q, 1001, S0S1S)$ by rule 5
$\vdash (q, 1001, 1S0S0S1S)$ by rule 2
$\vdash (q, 001, S0S0S1S)$ by rule 5
$\vdash (q, 001, 0S0S1S)$ by rule 3
$\vdash (q, 01, S0S1S)$ by rule 4
$\vdash (q, 01, 0S1S)$ by rule 3
$\vdash (q, 1, S1S)$ by rule 4
$\vdash (q, 1, 1S)$ by rule 3
$\vdash (q, \epsilon, S)$ by rule 5
$\vdash (q, \epsilon, \epsilon)$ by rule 3

Hence PDA accepted by empty stack

8.) Design a PDA for $L=\{a^n b^{2n} : n>0\}$

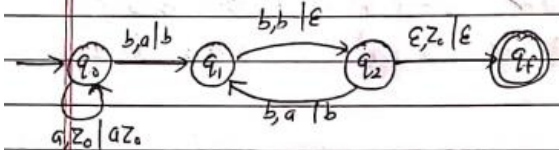
PDA accepts: abb, aabbbb, aaabbbbbb, ---

PDA rejects, a, b, ab, ba, aab, abbb, ---

PDA design D :-

Try for.

abb, aabbbb



Tutorial:

1. Design a PDA for $L=\{a^n b^{2n} : n>0\}$
2. Construct a PDA for the RE $r = 0^* 1^+$
3. Construct a PDA for $L=\{a^n b^{n+1} \mid n>0\}$
4. Design a PDA for $L=\{a^n b^m \mid n>m>0\}$
5. Construct a PDA for $L=\{a^3 b^n c^n : n>0\}$

4.12 Properties of CFL

Context free Languages (CFL) are accepted by PDA but not by FA.

1. Union:
   If L1 and L2 are two CFLs, then their union L1 ∪ L2 is also CFL.
2. Concatenation
   If L1 and L2 are two CFLs, then L1.L2 will also be CFL
3. Kleene * closure
   If L1 is CFL, then $L1^+$ will also be CFL
4. Intersection
   If L_1 and L_2 are two CFLs, then their intersection $L_1 \cap L_2$ is not CFL
5. Complement.
   If L_1 is CFL, then $\bar{L}_1$ is not CFL

not (closed under)

4.13 Closure properties of CFL

CFL are closed under : a) Union
b) Concatenation
c) Kleene Star Operation

4.14 Theorem Proofs

1. Theorem 1:

The family of CFL is closed under Union, Concatenation and Kleene-star closure.

Proof: Let L_1 and L_2 be two CFLs generated by two context-free grammar respectively as:

$G_1 = (V_1, \Sigma_1, P_1, S_1)$

$G_2 = (V_2, \Sigma_2, P_2, S_2)$

a) Union

Consider the Language $L(G)$, generated by the following grammar:

$G = \{V, \Sigma, P, S\}$

where,

$V = \{V_1 \cup V_2 \cup \{S\}\}$

$\Sigma = \{\Sigma_1 \cup \Sigma_2\}$

$S = \{S\}$

$P = \{P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}\}$

Let us choose a string $W \in (V_1 \cup V_2)^*$

If $S_1 \xrightarrow[G_1]{*} W$ or $S_2 \xrightarrow[G_2]{*} W$, and in our grammar $S \to S_1 \mid S_2$.

Hence, S will also lead to W

Here, G is clearly a CFG because it can generate the same string as by G_1 and G_2.

So, we can claim, $L(G) = L(G_1) \cup L(G_2)$

∴ CFL are closed under union.

Verification:

Suppose, G_1 and G_2 are two CFL where:

For G_1 we have:	For G_2 we have:
$S_1 \to aS_1b$	$S_2 \to cS_2d$
$S_1 \to \epsilon$	$S_2 \to \epsilon$

i.e. $L(G_1) = \{a^n b^n : n \geq 0\}$ $L(G_2) = \{c^n d^n : n \geq 0\}$

Let, $L(G) = L(G_1) \cup L(G_2)$

$= \{a^n b^n : n \geq 0\} \cup \{c^n d^n : n \geq 0\}$

Let us choose a string aabb

We have: $S \to S_1 \mid S_2$,

So, $S \to S_1$ $S \to S_2$

$\to aS_1b$ $\to cS_2d$

$\to aaS_1bb$ $\to c\epsilon d$

$\to aa\epsilon bb$ $\to cd$

$\to aabb$

For example:

$L_1 = \{a^n b^n \mid n>0\}$

$L_2 = \{c^n d^n \mid n>0\}$

$L = L_1 \cup L_2 = \{a^n b^n \cup c^n d^n \mid n>0\}$ is also CFL.

Their union says either of two conditions to be true.

b) Concatenation

Let us consider a language $L(G)$ generated by the CFG

$G = (V, \Sigma, P, S)$

where,

$V = V_1 \cup V_2 \cup \{S\}$

$\Sigma = \Sigma_1 \cup \Sigma_2$

$S = \{S\}$

$P = \{P_1 \cup P_2 \cup \{S \to S_1 S_2\}\}$

Let us choose a string, $W_1 \in L_1$ and $W_2 \in L_2$.

If $S_1 \xrightarrow[G_1]{*} W_1$ and $S_2 \xrightarrow[G_2]{*} W_2$

We can claim

$S \xrightarrow[G]{*} W_1 W_2$ $\because S \to S_1 S_2$

$\therefore L(G) = L(G_1)\, L(G_2)$

Hence,

CFL are closed under concatenation.

Eg:

$S_1 \to aS_1b \mid \epsilon$ $\quad$ $S_2 \to cS_2d \mid \epsilon$

$S \to S_1 S_2$

$\to a S_1 b S_2$

$\to a \epsilon b S_2$

$\to ab c S_2 d$

$\to abc \epsilon d$

$\to abcd$

c) Kleene Star

Let $L_1(G_1)$ be a CFL generated by a grammar

$G_1 = (V_1, \Sigma_1, P_1, S_1)$

Let kleene star of language L_1 (i.e. L_1^*) be generated by a grammar $G = (V, \Sigma, P, S)$

where,

$V = V_1 \cup \{S\}$

$\Sigma = \Sigma_1$

$P = P \cup \{S \to S_1 \mid SS_1 \mid \in\}$

Let us chose a string $W_1 \in L_1$.

If $S_1 \xrightarrow[G_1]{*} W_1$, then we can claim that

$S \xrightarrow{*} (W_1)^*$ $\because S \to S_1 \mid SS_1 \mid \epsilon$

$\therefore \quad L(G) = L(G_1)^*$

$\therefore$ CFL are closed under Kleene Star

Verification $ab \rightarrow (ab)^*$

Eg: $S_1 \rightarrow ab S_1 \mid \epsilon$

$S \rightarrow S_1$	$S \rightarrow S_1$
$\rightarrow \epsilon$	$\rightarrow ab S_1$
	$\Rightarrow ab$

$S \rightarrow S S_1$
$\rightarrow S_1 S_1$
$\rightarrow ab S_1 S_1$
$\rightarrow ab \epsilon S_1$
$\rightarrow ab ab S_1$
$\rightarrow ab ab$

So, $ab \rightarrow (ab)^*$

Theorem 2:

The family of Context Free Language is not closed under intersection and Complementation

a) Intersection:

Let us define two CFLs L_1 and L_2 as:-

$L_1 = \{a^n b^n c^m \mid n \geq 1, m \geq 1\}$

$L_2 = \{a^n b^n c^n \mid n \geq 1, m \geq 1\}$

Here,

$L = L_1 \cap L_2 = \{a^n b^n c^m \cap a^n b^n c^n\}$

$= \{a^n b^n c^n : n \geq 1\}$

But by $L = \{a^n b^n c^n : n \geq 1\}$ is not a CFL. This can be proved by using pumping lemma by contradiction.

Hence, CFLs are not closed under intersection.

b) Complementation.

We know CFL are closed under union.

Let L_1 and L_2 be two CFLs.

Assume, $\overline{L_1}$ and $\overline{L_2}$ are also CFLs. So, we can say that $\overline{L_1} \cup \overline{L_2}$ are CFL.

By demorgan's Law,

$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$

But $L_1 \cap L_2$ is not CFL. This contradicts our assumption.

Hence CFL is not closed under complementation

4.15 Pumping lemma for CFL

- The pumping Lemma for CFL states that for any Context Free Language L, it is possible to find two substrings that can be pumped any number of times, and still be in the same language.
- For any Language L, we can break its string into five parts and pump the second and the fourth ~~substring~~ substring.
- Pumping Lemma, here also, is used as a tool to prove that a language is not CFL.
- Because, if any one string does not satisfy its conditions, then the language is not CFL.

4.16 Pumping Lemma statement

If L is a CFL and ω be any string in L, there exists an integer n with $|\omega| \geq n$, then we can write $\omega = uvxyz$ such that:

i) $|vxy| \leq n$

ii) $|vy| > 0$

iii) $uv^ixy^iz \in L$ for all $i \geq 0$.

It means two strings v and y can be pumped any number of times, and the resulting string will still be in L.

If L is a context free language then L has a pumping length 'n' such that any string $|S| \geq n$ may be divided into 5 pieces i.e. $S = uvxyz$, such that the following conditions must be true:

1) $uv^ixy^iz \in L$ for every $i \geq 0$
2) $|vy| > 0$
3) $|vxy| \leq n$

4.17 Proof of Pumping Lemma for CFL

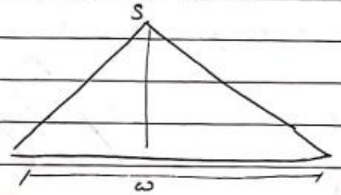
→ Consider a CNF grammar $G = (V, \Sigma, P, S)$ that has m non-terminals (or variables).

→ Let n be the pumping length, and lets choose $n = 2^m$.

→ Consider a string ω ~~that is produced~~ of length 2^m (i.e. n) derived from the grammar G.

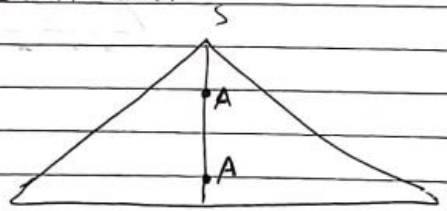
i.e. $|z| = 2^m$ or more.

The derivation tree is as shown below:-



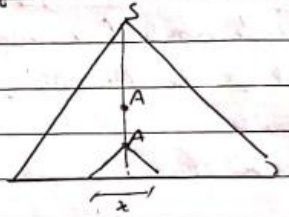
→ The derivation tree for ω must have a path of length m+1 or more. Clearly there are m+1 non-terminals.

→ But there are only m variables in grammar. So at least two of the last m+1 variables on path are same (ie. by pigeonhole principle). Let they be A and A as shown in the derivation tree.



→ The sub-tree subtended at the lower A is as shown below. It generates a string of w.

i.e. $A \overset{*}{\Rightarrow} x$

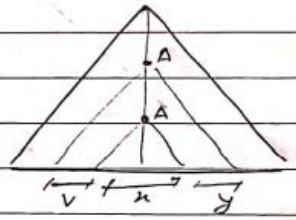


→ The sub-tree subtended at the upper A is as shown below. It derives the string vxy

i.e. $A \overset{+}{\Rightarrow} vxy$

Also,

$A \overset{+}{\Rightarrow} vAy$

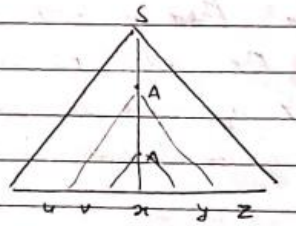


→ The S derives the entire string as uvxyz

$S \overset{*}{\Rightarrow} uvxyz$

Also

$S \overset{*}{\Rightarrow} uAz$



Now,

$S \overset{*}{\Rightarrow} uAz$

$\Rightarrow uvAyz$

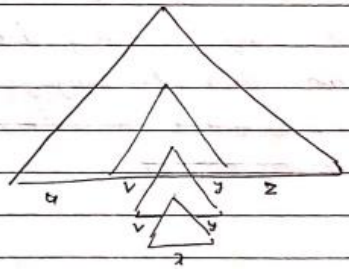
$\Rightarrow uvvAyyz$

$\Rightarrow uvv\ldots A\ldots yyz$

$\Rightarrow uvv\ldots x\ldots yyz$

$\Rightarrow uv^ixy^iz$ for $i>0$

→ The resulting parse tree is shown as below:-



Hence, we can find any yield of form $uv^ixy^iz \in L$ for any $i>0$.

4.18 Numerical

1. Show that $L = \{a^n b^n c^n : n \geq 1\}$ is not CFL

Sol^n: Assume that L is context free, and ω be any string in L, ie $\omega \in L$.

Let n be pumping length with $|\omega| \geq n$;

Then, we can decompose $\omega = uvxyz$ such that $|vy| \leq n$ and $|vy| > 0$.

So, it must satisfy the condition $uv^i xy^i z \in L$ for all $i \geq 0$.

Here, for given language L, $\omega = a^n b^n c^n$

Let us take $n = 4$, then $L = a^4 b^4 c^4$

So,

$z = aaaa\ bbbb\ cccc$

Now

Decompose z into $z = uvxyz$ such that $|vxy| \leq n$ and $|vy| > 0$

$\underbrace{a\ a\ a\ a}_{u}\ \underbrace{b\ b\ b\ b}_{v}\ \underbrace{c\ c\ c\ c}_{z}$ (x, y)

Lets take $i = 2$.

After pumping vy, we get :-

$aaa\ (ab)^2\ b\ (bb)^2\ bcccc$

$= aaa\ abab\ b\ bbbb\ bcccc$

$= a^4 b^4 c^4$ which is not in language L.

Thus,

given language L is not CFL. Hence proved by contradiction.

2. Check if $L = \{a^{2n} b^n \mid n > 0\}$ is context free.

Sol^n: Assume L is context free, and ω be any string in L

Let 'n' be the pumping length with $|\omega| \geq n$.

Then, we can decompose $\omega = uvxyz$ such that $|vxy| \leq n$ and $|vy| > 0$.

So it must satisfy the condition $uv^i xy^i z \in L$ for all $i \geq 0$

Here,

for given language L, $\omega = a^{2n} b^n$

Let us take $n = 4$, then $L = a^8 b^4$

So,

$z = aaaaaaaa\ bbbb$

Now,

decompose z into $z = uvxyz$ such that $|vxy| \leq n$ and $|vy| > 0$.

$\underbrace{a\ a\ a\ a\ a\ a}_{u}\ \underbrace{a\ a}_{v}\ \underbrace{b}_{x}\ \underbrace{b}_{y}\ \underbrace{b\ b}_{z}$

Let us take $i = 2$

After applying pumping vy, we get

$aaaa\ aaaaaa\ b\ bb\ bb$

$= a^{11} b^5$

which is not in the given grammar

Hence, given language L is not CFL.

Thus, proved by contradiction.

3. Prove that the language $L = \{ww \mid w \in \{a,b\}^*\}$ is not context free.

Sol^n: Assume L is context free, and ω be any string in L.
Let n be the pumping length with $|\omega| \geq n$.
Then, we can decompose $\omega = uvxyz$ such that $|vy| \leq n$ and $|vy| > 0$.
So it must satisfy the condition $uv^ixy^iz \in L$ for all $i \geq 0$.
Here, lets take ω for the given language L as:-

$$\omega = a^n b^n a^n b^n$$

Taking $n = 4$, then $L = a^4 b^4 a^4 b^4$

So,

$$z = aaaa\ bbbb\ aaaa\ bbbb$$

Now,
decompose z into $z = uvxyz$ such that $|vxy| \leq n$ and $|vy| > 0$

aaaa bbbb aaaa bbbb
u v x y z

Let us take $i = 2$,
After pumping vy, we get:

aaaa bbbb baba aaa abbbb
$= a^4 b^4 a\ b\ a^5 b^4$

which is not in the grammar.
Hence, given language L is not CFL.

Thus, proved by contradiction.

8. Check if $L = \{a^{2n} b^n \mid n > 0\}$ is context free

→ Assume L is context free, and ω be any string in L.
Let n be the pumping length with $|\omega| \geq n$.
Then we can decompose $\omega = uvxyz$ such that $|vy| \leq n$ and $|vy| > 0$.
So it must satisfy the condition $uv^ixy^iz \in L$ for all $i \geq 0$

Here for given language L,

$$\omega = a^{2n} b^n$$
$$= aaaabbbb$$

Let us take $n = 4$, then, $L = a^8 b^4$

So,

$$z = aaaaaaaa\ bbbb$$

Now, decompose z into $z = uvxyz$ such that $|vxy| \leq n < |vy|$

aaaaaaaa bbbb
u v x y z

Let us take $i = 2$,
After pumping vy, we get

$$aaaaaaaaaa\ bbbb$$
$$= a^{10} b^4$$

which is not in the given grammar

Hence given language L is not CFL
proved by contradiction.

4.19 Decision Algorithms for CFL

Basically there are algorithms
① Algorithm for deciding Emptiness of CFL
② Algorithm for deciding finiteness of CFL
③ Membership - The CYK algorithm

1) Emptiness Algorithm
- If we cannot derive any string of terminals from the given grammar, then its language is called as an Empty language. i.e. $L(G) = \phi$

Algorithm:
1. Remove all the useless symbols from the grammar.
   A useless symbol is one that does not derive any string of terminals.
2. If the start symbol is found to be useless, then language is empty otherwise not.

Eg: Consider the grammar:
$S \to XY$
$X \to AX$
$X \to AA$
$A \to a$
$Y \to BY$
$Y \to BB$
$B \to b$

Here, $S \to XY$
$\to AAY$
$\to aAY$
$\to aaY$
$\to aaBB$
$\to aabB$
$\to aabb$

So, $L(G) \neq \phi$

2) Algorithm for Finiteness
Algorithm
1. Reduce the grammar by eliminating $\in$, unit and useless productions.
2. Draw a directed graph whose nodes are variables of given grammar.
   There exists an edge from A to B if there exists a production of the form $A \to \alpha B \beta$.
3. If cycle in directed graph → infinite CFL
   else (no cycles) → finite CFL

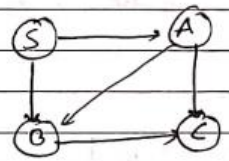
Eg: Check whether language of the following grammar is finite or not.
$S \to AB/a$
$A \to BC/b$
$B \to CC/c$

Eg: Check whether language of the following grammar is finite or not.
$S \to AB/a$
$A \to BC/b$
$B \to CC/c$

Sol":

1) Grammar is already completely reduced.

2) Draw directed graph for S, A, B, C



$S \to AB \Rightarrow S \to A, S \to B$

$A \to BC \Rightarrow A \to B, A \to C$

$B \to CC \Rightarrow B \to C$

3) Since no cycle, the CFG is finite.

3) The membership - CYK algorithm

- It is used to decide whether a given string belongs to the language of grammar or not.
- named after Cocke - Younger - Kasami
- operates on CFG given in CNF

Example:

For the given grammar, check the acceptance of string $\omega = baaba$ using CYK algorithm.

$S \to AB \mid BC$

$A \to BA \mid a$

$B \to CC \mid b$

$C \to AB \mid a$

Here, $\omega = baaba$

Length of word $|\omega| = 5$

No. of sub-strings possible $= n(n+1)/2 = 5 \times 6/2 = 15$

Let us construct a triangular table and fill the cells.

V_{15} {S,A,C}				
V_{14} ϕ	V_{24} {S,C,A}			
V_{13} ϕ	V_{23} {B}	V_{33} {B}		
V_{12} {A,S}	V_{22} {B}	V_{32} {S,C}	V_{42} {A,S}	
V_{11} {B}	V_{21} {A,C}	V_{31} {A,C}	V_{41} {B}	V_{51} {A,C}

Working notes:

For 1st row

b	a	a	b	a
{B}	{A,C}	{A,C}	{B}	{A,C}

For 2nd row:

V_{12}	V_{22}	V_{32}	V_{42}	
V_{11}	V_{21}	V_{31}	V_{41}	V_{51}

$V_{12} = V_{11} \cdot V_{21} = \{B\}\{A,C\} = \{BA, BC\} = \{A,S\}$

$V_{22} = V_{21} \cdot V_{31} = \{A,C\}\{A,C\} = \{AA, AC, CA, CC\} = \{CC\} = \{B\}$

$V_{32} = V_{31} \cdot V_{41} = \{A,C\}\{B\} = \{AB, CB\} = \{S,C,\phi\} = \{S,C\}$

$V_{42} = V_{41} \, V_{51} = \{B\}\{A,C\} = \{BA, BC\} = \{A,S\}$

For 3rd row:

V_{13}	V_{23}	V_{33}		
V_{12}	V_{22}	V_{32}	V_{42}	
V_{11}	V_{21}	V_{31}	V_{41}	V_{51}

$V_{13} = V_{11} V_{22} \cup V_{12} \cdot V_{31}$

$= \{B\}\{B\} \cup \{A,S\}\{A,C\}$

$= \{BB\} \cup \{AA, AC, SA, SC\}$

$= \phi \cup \phi = \phi$

$V_{23} = V_{21} \cdot V_{32} \cup V_{22} \cdot V_{41}$

$= \{A,C\}\{S,C\} \cup \{B\}\{B\}$

$= \{AS, AC, CS, CC\} \cup \{BB\}$

$= \{B\} \cup \phi = \{B\}$

$V_{33} = V_{31} \cdot V_{42} \cup V_{32} V_{51}$

$= \{A,C\}\{A,S\} \cup \{S,C\}\{A,C\}$

$= \{AA, AS, CA, CS\} \cup \{SA, SC, CA, CC\}$

$= \emptyset \cup \{B\}$

$= \{B\}$

For 4th row:

$V_{14} = V_{11} \cdot V_{23} \cup V_{12} \cdot V_{32} \cup V_{13} \cdot V_{41}$

$= \{B\}\{B\} \cup \{A,S\}\{S,C\} \cup \{\emptyset\}\{B\}$

$= \{BB\} \cup \{AS, AC, SS, SC\} \cup \{B\}$

$= \emptyset \cup \emptyset \cup \emptyset$

$= \emptyset$

$V_{24} = V_{21} \cdot V_{33} \cup V_{22} V_{42} \cup V_{23} V_{51}$

$= \{A,C\}\{B\} \cup \{B\}\{A,S\} \cup \{B\}\{A,C\}$

$= \{AB, CB\} \cup \{BA, BS\} \cup \{BA, BC\}$

$= \{S,C\} \cup \{A\} \cup \{A,S\}$

$= \{S,C,A\}$

For 5th row:

$V_{15} = V_{11} V_{24} \cup V_{12} V_{33} \cup V_{13} V_{42} \cup V_{14} V_{51}$

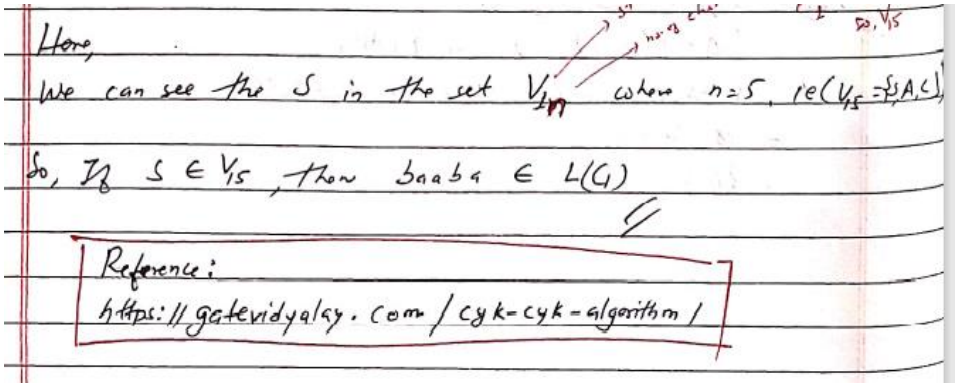
$= \{B\}\{S,C,A\} \cup \{A,S\}\{B\} \cup \{\emptyset\}\{A,S\} \cup \{\emptyset\}\{A,C\}$

$= \{BS, BC, BA\} \cup \{AB, SB\} \cup \{A,S\} \cup \{A,C\}$

$= \{S,A\} \cup \{S,C\} \cup \{A,S\} \cup \{A,C\}$

$= \{S,A,C\}$

baaba



End of Chapter 4

5.1 Turing Machine Introduction

Turing machine was invented in 1936 by Alan Turing. It is an accepting device which accepts Recursive Enumerable Language generated by type 0 grammar. There are various features of the Turing machine:

- It has an external memory which remembers arbitrary long sequence of input.
- It has unlimited memory capability.
- The model has a facility by which the input at left or right on the tape can be read easily.
- The machine can produce a certain output based on its input. Sometimes it may be required that the same input has to be used to generate the output. So in this machine, the distinction between input and output has been removed. Thus a common set of alphabets can be used for the Turing machine.

A Turing machine is a hypothetical machine thought of by the mathematician Alan Turing in 1936.
- Despite its simplicity, the machine can simulate ANY computer program, no matter how complicated it is!
- It is the theoretical foundation of modern computers.
- It can do everything that a real computer can do.
- At any time, the machine ~~can~~ has a head which is positioned over one of the squares on the tape. With this head, the machine can perform three very basic operations:-
  1. Read the symbol on the square under the head.
  2. Edit the symbol by writing a new symbol or erasing it.
  3. Move the tape left or right by one square so that the machine can read and edit the symbol on a neighbouring (adjacent) square.

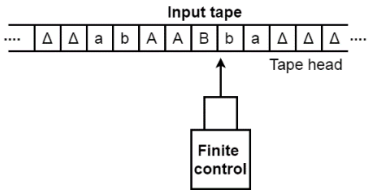
Basic Model of Turing machine

The turning machine can be modelled with the help of following representation.

1. The input tape is having an infinite number of cells, each cell containing one input symbol and thus the input string can be placed on tape. The empty tape is filled by blank characters.

Input tape

2. The finite control and the tape head which is responsible for reading the current input symbol. The tape head can move to left to right.
3. A finite set of states through which machine has to undergo.
4. Finite set of symbols called external symbols which are used in building the logic of Turing machine.



5.2 TM as physical computing device

We can visualize a TM as a physical computing device that can be represented as a diagram as shown below:-

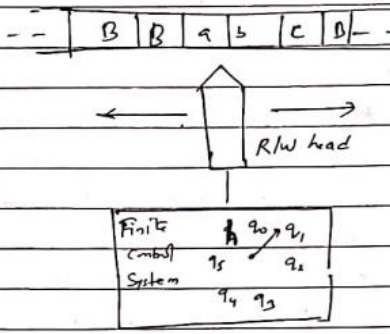


Fig: Turing machine

A TM consists of:

1. A Tape, ~~divided~~ (infinite on left and)
   - with an ~~end~~ on the left but infinite on the right side.
   - tape is divided into squares (or cells), with each cell capable of holding one of the tape symbols including the blank symbol #.

2. A Finite control
   - It can have any one of the finite number of states.
   - The states in TM can be divided into three categories:-
   i) Initial state
      - denoted by q_0
      - it the start of its operation of TM
   ii) Halt state
      It is the state in which TM stops all further operations
   iii) Other intermediate states

3. Read/Write Head.
   - always stationed at one of the tape cells.
   - provides interaction between tape and finite control system.
   - The head can move left or to the right.
   - The control may decide not to move the head.
   - It can also read the cell and replace the symbol in the cell by another symbol.
   ⇒ * After reading an input symbol, it is replaced with another symbol, the internal state of TM is changed, and it moves from one cell to the right or left.
   - If the TM reaches the final state, the input string is accepted, otherwise rejected.

5.3 Formal definition of TM

A TM can be formally described as a 7-tuple as:-

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

where,

$Q \rightarrow$ set of states

$\Sigma \rightarrow$ a finite set of input alphabets

$\tau \rightarrow$ a finite set of tape alphabets

$q_0 \rightarrow$ start state $(q_0 \in Q)$

$F \rightarrow$ set of final states $(F \subseteq Q)$

$B \rightarrow$ Blank symbols $(B \in \tau)$

$\delta \rightarrow$ transition rules

$$Q \times \Sigma \rightarrow (Q \times \tau \times \{L, R, N\})$$

where:

L → denotes the tape Head moves to the left adjacent cell

R → " " " " " a right " "

N → denotes " " " does not move.

A Turing machine can be defined as a collection of 7 components:

- Q : the finite set of states
- ∑ : the finite set of input symbols
- T : the tape symbol
- q0 : the initial state
- F : a set of final states
- B : a blank symbol used as a end marker for input
- δ : a transition or mapping function.

The mapping function shows the mapping from states of finite automata and input symbol on the tape to the next states, external symbols and the direction for moving the tape head. This is known as a triple or a program for Turing machine.

(q0, a) → (q1, A, R)

That means in q0 state, if we read symbol 'a' then it will go to state q1, replaced a by X and move ahead right (R stands for right).

5.4 Moves of TM

The moves of TM, $M = (Q, \Sigma, \tau, \delta, q_0, B, f)$ is described by the notation $\vdash$ for single move and $\vdash^*$ for zero or more moves.

Let us discuss some moves of the TM;

1) If $q \in Q$, $c \in \Sigma$. and $\delta(q, c) = (q_1, b, R)$, then:

TM ~~will~~ when in state q and currently scanning symbol c will enter in state q_1 and move towards right adjacent cell after replacing c with b

2) $\delta(q_1, a) = (q_2, b, L)$ means:-

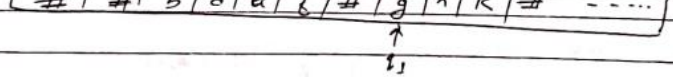
At present, TM is in state q_1 and scanning the symbol a, will enter in state q_2 and write b in place of a and move towards left adjacent cell.

3) $\delta(q_1, x) = (q_2, y, N)$ means:-

At present, TM is in state q_1 and scanning the symbol x will enter state q_2 and write y in the place of x. But TM will stick with current position, ie. it does not move towards left or right.

5.5 Instantaneous Description for TM

- A string $x_1 x_2 x_3 \ldots x_{i-1}\, q\, x_i\, x_{i+1} \ldots x_n$ represents instantaneous description of TM where q is state of TM.
- The tape head is scanning the i^{th} symbol from left.
- $x_1 \ldots x_n$ is the portion of tape between left-most and right-most non-blank.
- For eg: TM is at state q_1 scanning the symbol g with the symbols on the tape as follows:-



We can show the situation as follows:-

$(q_1, \#\#bdab\#ghk\#)$

Definition: ID of a TM is a snapshot of TM to describe the current situation of the TM.

5.6 Transition diagram for TM

It consists of :-

- set of nodes representing states of TM.
- an arc from any state q to p is labelled by the items of the form X/YD where ~~X and Y are~~ ~~where~~ X and Y are tape symbols
  D is a direction either Left or Right

ie. $\delta(q, x) = (p, Y, D)$

Q: Let TM = {Q, Σ, τ, δ, q_0, F, B} where,

Q = {q_0, q_1, q_2, h}

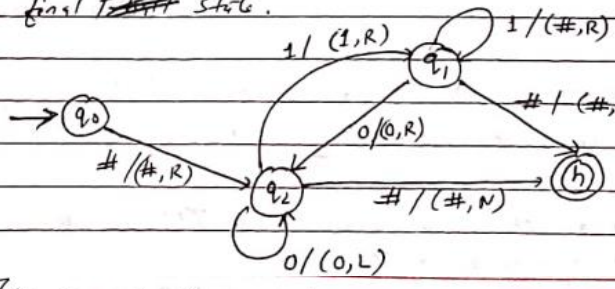
Σ = {0, 1}

τ = {0, 1, #} and

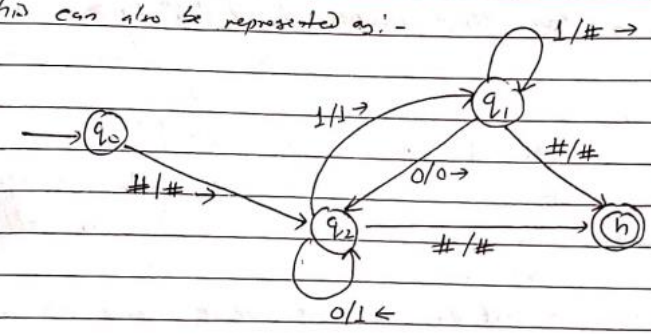
δ is given by the following transition table:-

	0	1	#
q_0	–	–	(q_1, #, R)
q_1	(q_1, 0, R)	(q_1, #, R)	(h, #, N)
q_2	(q_2, 0, L)	(q_1, 1, R)	(h, #, N)
h	–	–	–

The transition diagram for above TM will be as shown below, where we assume that q_0 is the initial state and h is a final state.



This can also be represented as:-



5.7 TM as language acceptor

- A TM works as language acceptor for a language L if it is able to tell whether a string 'w' belongs to the language L or not. i.e. $w \in L(M)$.
- If the TM halts in a final state, then it accepts string w.
- If the TM halts in a non-final state or never halts, then the TM doesn't accepts the string w i.e. $w \notin L(M)$.

Q: A TM to accept the set of all strings of 0's and 1's containing at least one 1.

Let TM = {Q, Σ, τ, δ, q_0, h, B}

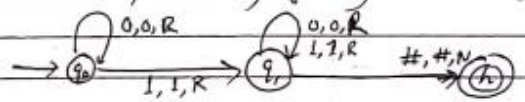
Q = {q_0, q_1, h}

τ = {0, 1, #}

Σ = {0, 1}

q_0 = {q_0}

δ is represented by following transition diagram:



Here, halt state is reached only when at least one 1 is encountered and h works as accepted state

Test for #00110#

q_0 #00110#

q_0 #00110#

q_0 #00110#

q_1 #00110#

q_1 #00110#

q_1 #00110#

h #00110#

5.8 Numerical: TM design for Language Acceptor

1. Design a TM that accepts $L = \{a^n b^n : n \geq 1\}$

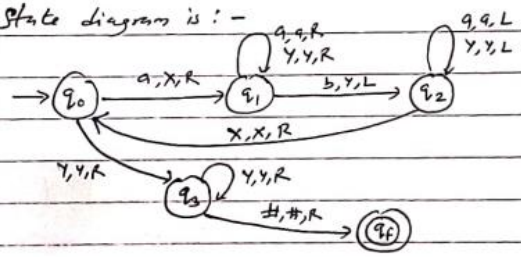
Solⁿ: Let us assume that the input string is terminated by a blank symbol # at each end of the string.

#	#	a	a	a	b	b	b	#	#

The TM can be constructed by the following moves:

- TM will change a to X and b to Y until all a's and b's are matched.
- Starting at left end of input, it repeatedly changes a to X and moves over whatever a's and Y's

State diagram is :-



Hence, TM is :-

$M = \{Q, \Sigma, \tau, \delta, q_0, F, B\}$

where,

$Q = \{q_0, q_1, q_2, q_3, q_f\}$

$\Sigma = \{a, b\}$

$\tau = \{X, Y, \#\}$

$q_0 = \{q_0\}$

$F = \{q_f\}$

$B = \{\#\}$

δ is given by transition table as:-

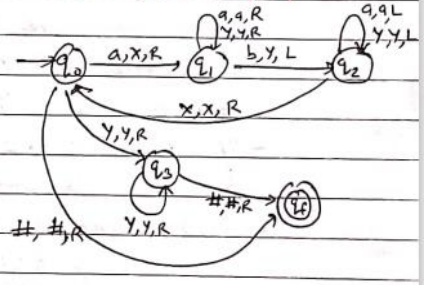
	a	b	X	Y	#
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	(q_1, a, R)	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	(q_2, a, L)	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	$(q_f, \#, R)$
q_f	—	—	—	—	—

Verification: Test for aabb

q_0 a a b b #
$\vdash q_1$ X a b b #
$\vdash q_1$ X a b b #
$\vdash q_2$ X a Y b #
$\vdash q_2$ X a Y b #
$\vdash q_0$ X a Y b #
$\vdash q_1$ X X Y b #
$\vdash q_1$ X X Y b #
$\vdash q_2$ X X Y Y #
$\vdash q_2$ X X Y Y #
$\vdash q_0$ X X Y Y #
$\vdash q_3$ X X Y Y #
$\vdash q_3$ X X Y Y #
$\vdash q_f$ X X Y Y #

Hence accepted

Design a TM that accepts $L = \{a^n b^n : n \geq 0\}$



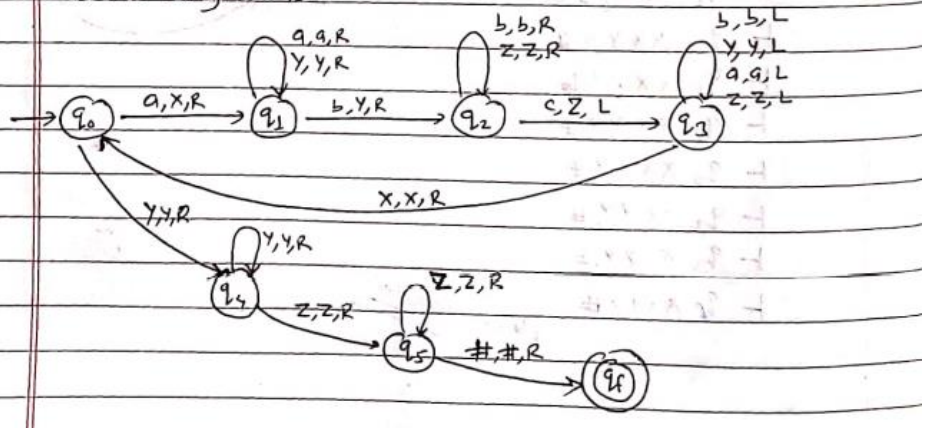
2. Design a TM that accepts $L = \{a^n b^n c^n : n \geq 1\}$

Sol^n: Let us assume that the input string is terminated by a blank symbol #, at each end of the string.

#	a	a	b	b	c	c	#

The TM can be constructed by the following moves:-

- TM will change a to X, b to Y and c to Z until all a's, b's and c's are matched.
- Starting at left end of input, it repeatedly changes a to X and moves over whatever a's and Y's it sees until it reaches to state q_1.
- It changes b to Y and moves over whatever b's and Z's it sees until it reaches to state q_2.
- It changes c to Z and moves left over Z's, Y's, b's and a's until it finds an X.
- State diagram is:-



Hence, TM is: $M = \{Q, \Sigma, \tau, \delta, q_0, F, B\}$

where, $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_f\}$ $\quad q_0 = \{q_0\}$

$\Sigma = \{a, b, c\}$ $\quad F = \{q_f\}$

$\tau = \{X, Y, Z, \#\}$ $\quad B = \{\#\}$

δ is given by:-

	a	b	c	X	Y	Z	#
q_0	(q_1, X, R)	–	–	–	(q_4, Y, R)	–	–
q_1	(q_1, a, R)	(q_2, Y, R)	–	–	(q_1, Y, R)	–	–
q_2	–	(q_2, b, R)	(q_3, Z, L)	–	–	(q_2, Z, R)	–
q_3	(q_3, a, L)	(q_3, b, L)	–	(q_0, X, R)	(q_3, Y, L)	(q_3, Z, L)	–
q_4	–	–	–	–	(q_4, Y, R)	(q_5, Z, R)	–
q_5	–	–	–	–	–	(q_5, Z, R)	$(q_f, \#, R)$
q_f	–	–	–	–	–	–	–

Verification: for aabbcc

q_0 aabbcc# $\vdash$ q_2 XXYYZc#

$\vdash q_1$ XabbCc# $\vdash$ q_2 XXYYZc#

$\vdash q_1$ XabbCc# $\vdash$ q_3 XXYYZZ#

$\vdash q_2$ XaYbCc# $\vdash$ q_3 XXYYZZ#

$\vdash q_2$ XaYbCc# $\vdash$ q_3 XXYYZZ#

$\vdash q_3$ XaYbZc# $\vdash$ q_3 XXYYZZ#

$\vdash q_3$ XaYbZc# $\vdash$ q_0 XXYYZZ#

$\vdash q_3$ XaYbZc# $\vdash$ q_4 XXYYZZ#

$\vdash q_3$ XaYbZc# $\vdash$ q_4 XXYYZZ#

$\vdash q_0$ XaYbZc# $\vdash$ q_5 XXYYZZ#

$\vdash q_1$ XXYbZc# $\vdash$ q_5 XXYYZZ#

$\vdash q_1$ XXYbZc# $\vdash$ q_f XXYYZZ# accepted

GURUKUL

3. Design a TM that accepts $L = \{w \in \{a,b\}^* \}$ and no of a's and no of b's are equal.

Sol^n: Clearly the Language accepts strings like ab, ba, aabb, abab, abba, aabbba, abbbaa, ---

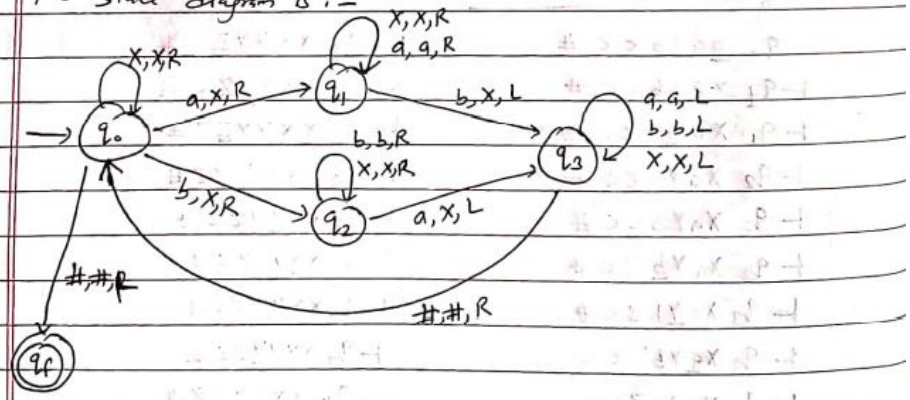
Let us assume that the input string is terminated by a blank symbol #, at each end of the string.

#	a	a	b	b	b	a	#

TM is constructed by the following moves:-

- TM will change a to X, and finds b and change it to X.
- Repeatedly move left and right, each pair of a and b are replaced with X.

The state diagram is:-



Here, TM is: $M = \{Q, \Sigma, \tau, \delta, q_0, FB\}$

where $Q = \{q_0, q_1, q_2, q_3, q_f\}$, $q_0 = \{q_0\}$

$\Sigma = \{a,b\}$, $F = \{q_f\}$

$\tau = \{X, \#\}$, $B = \{\#\}$

δ is given by:-

	a	b	X	#
q_0	(q_1, X, R)	(q_2, X, R)	(q_0, X, R)	$(q_f, \#, L)$
q_1	(q_1, a, R)	(q_3, X, L)	(q_1, X, R)	—
q_2	(q_3, X, L)	(q_2, b, R)	(q_2, X, R)	—
q_3	(q_3, a, L)	(q_3, b, L)	(q_3, X, L)	$(q_0, \#, R)$
q_f	—	—	—	—

Verification: For aabbba:

q_0 #a a b b b a #
$\vdash$ q_1 #X a b b b a #
$\vdash$ q_1 #X a b b b a #
$\vdash$ q_3 #X a X b b a #
$\vdash$ q_3 #X a X b b a #
$\vdash$ q_3 # X a X b b a #
$\vdash$ q_0 # X a X b b a #
$\vdash$ q_0 # X a X b b a #
$\vdash$ q_1 # X X X b b a #
$\vdash$ q_1 # X X X b b a #
$\vdash$ q_3 # X X X X b a #
$\vdash$ q_3 # X X X X b a #
$\vdash$ q_3 # X X X X b a #
$\vdash$ q_3 # X X X X b a #

$\vdash$ q_0 # X X X X b a #
$\vdash$ q_1 # X X X X b a #
$\vdash$ q_1 # X X X X b a #
$\vdash$ q_3 # X X X X X a #
$\vdash$ q_3 # X X X X X a #
$\vdash$ q_2 # X X X X X a #
$\vdash$ q_3 # X X X X X X #
$\vdash^*$ q_3 # X X X X X X #
$\vdash$ q_0 # X X X X X X #
$\vdash^*$ q_0 # X X X X X X #
$\vdash$ q_0 # X X X X X X #
$\vdash$ q_f # X X X X X X #

Hence accepted

4. Design a TM for $L = \{ w c w \mid w \in \{a,b\}^* \}$

Sol^n: Clearly the string generated by the language L is like:
abcab, abbcabb, ababcabab, ...
Let us assume that the input string has blank symbol # at both the extreme ends.

#	a	b	a	a	b	c	a	b	a	a	b	#

We design the TM as in below fashion:-
- TM scans a in the first half and changes it into X, and finds the corresponding a in the second half and changes it to X.
- TM scans b in first half and changes to Y, and correspondingly changes b in second half to Y.
- Thus all a are changed into X and b into Y.

The ~~state diagram~~ TM is:

$M = \{ Q, \Sigma, \tau, \delta, q_0, F, B \}$

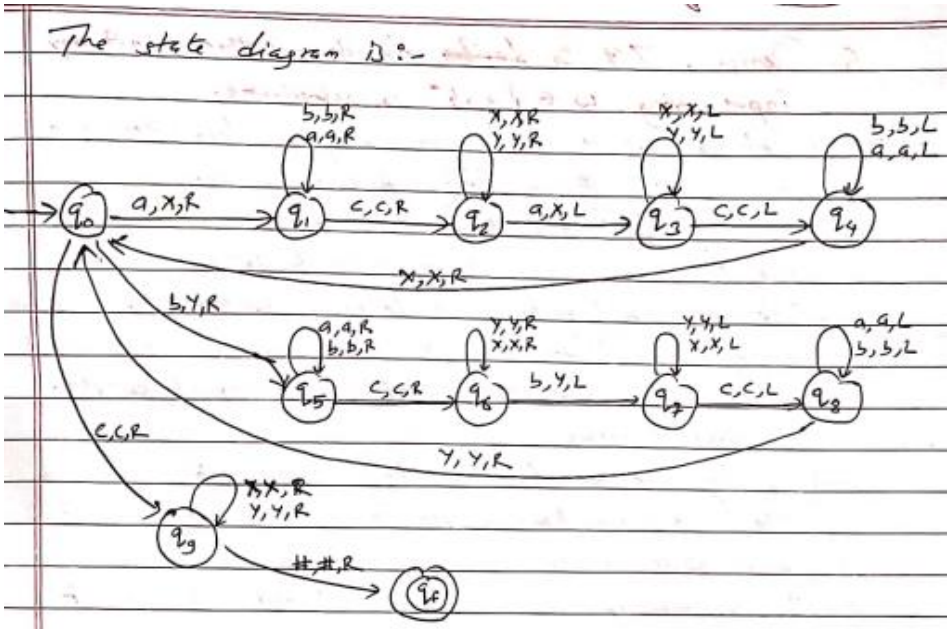
where, $Q = \{ q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_f \}$
$\Sigma = \{a, b\}$
$\tau = \{X, \#\}$
$q_0 = \{q_0\}$
$F = \{q_f\}$
$B = \{\#\}$

δ is given by:

	a	b	c	#
q_0				
q_1				
q_f				

GURUKUL

The state diagram is:-



Verification: for string abbcabb

q_0 #abbcabb#
$\vdash$ q_1 #Xbbcabb#
$\vdash$ q_1 #Xbbcabb#
$\vdash$ q_1 #Xbbcabb#
$\vdash$ q_2 #Xbbcabb#
$\vdash$ q_3 #XbbcXbb#
...
$\vdash^*$ q_f #XYYcXYY# Hence accepted

5. Design a TM to ~~decide~~ decide whether or not any input string $w \in \{a,b\}^*$ is palindrome.

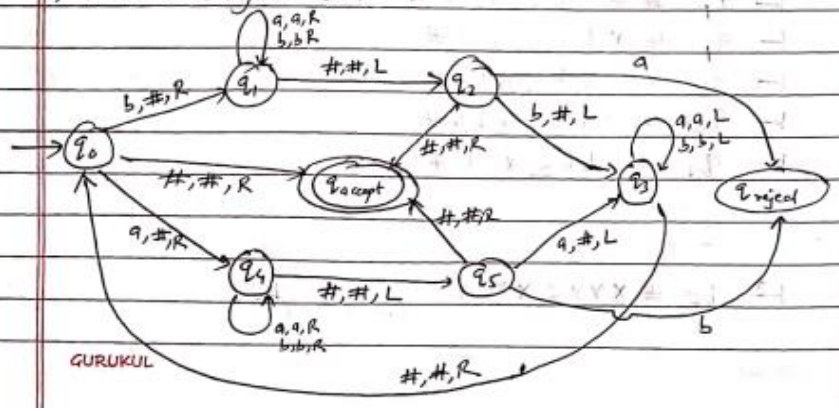
Sol^n: Let us assume that input string is terminated by a blank symbol #, at ~~right~~ end of the string.

|a|b|b|a|#|

The TM can be constructed by the following moves:-

- The tape head ~~now~~ reads leftmost symbol of w, replaces it by #. Then the tape head moves to the right most symbol and test whether it is equal to the leftmost symbol.
- If they are equal, then Right most symbol is deleted, then the tape head moves to the new left-most symbol and above process is repeated.
- If corresponding left most and right-most cells are not equal, TM enters the reject state and stops.
- TM enters accept state as soon as string currently stored in tape is empty.

The state diagram is:-



Here TM is :-

$M = \{Q, \Sigma, \tau, \delta, q_0, F, B\}$

where,

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$

$\Sigma = \{a,b\}$

$\tau = \{a, b, \#\}$ δ is given by:

$q_0 = \{q_0\}$

$F = \{q_{accept}\}$

$B = \{\#\}$

	a	b	#
q_0	$(q_4, \#, R)$	$(q_4, \#, R)$	q_{accept}
q_1	(q_1, a, R)	(q_1, b, R)	$(q_2, \#, L)$
q_2	-	$(q_3, \#, L)$	q_{accept}
q_3	(q_3, a, L)	(q_3, b, L)	$(q_0, \#, R)$
q_4	(q_4, a, R)	(q_4, b, R)	$(q_5, \#, L)$
q_5	$(q_3, \#, L)$		q_{accept}
q_{accept}	-	-	-

Verification: test for abba

q_0 : $\underline{a}bba\#$
$\vdash q_4$. $\#\underline{b}ba\#$
$\vdash q_4$ $\#b\underline{b}a\#$
$\vdash q_4$ $\#bb\underline{a}\#$
$\vdash q_4$ $\#bba\underline{\#}$
$\vdash q_5$ $\#bb\underline{a}\#$
$\vdash q_3$ $\#b\underline{b}\#\#$
$\vdash q_3$ $\#\underline{b}b\#\#$
$\vdash q_3$ $\#\underline{\#}bb\#\#$
$\vdash q_0$ $\#\underline{b}b\#\#$
$\vdash q_1$ $\#\#\underline{b}\#\#$

$\vdash q_1$ $\#\#\underline{b}\#\#$
$\vdash q_2$ $\#\#\underline{b}\#\#$
$\vdash q_3$ $\#\#\underline{\#}\#\#$
$\vdash q_0$ $\#\#\underline{\#}\#\#$
$\vdash q_{accept}$ $\#\#\#\underline{\#}\#$

Finally, the TM replaces all symbol by # and enters q_{accept} state. The input string is palindrome.

Also check for ab#

q_0 $\underline{a}b\#$
q_4 $\#\underline{b}\#$
q_4 $\#b\underline{\#}$
q_5 $\#\underline{b}\#$
q_{rej} $\#b\#$

Since TM enters q_{reject} state, the input string is not palindrome.

5. Design a TM that works as a simple eraser, which changes every non-blank symbol to blank with alphabets $\Sigma = \{0, 1, \#\}$. Hence test your design for #01o1# to ######

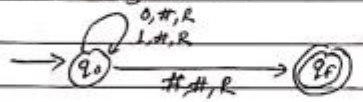
Sol: Let us assume that input string is terminated by a blank symbol # at each end of the string.

#	0	1	0	1	#

The TM can be constructed as:

- The TM reads 0 and replaces 0 or 1 whatsoever with a blank # and moves right.
- If the TM reaches the extreme right it stops at final state.

The state diagram is:-



Here, the TM is as:

$$M = \{Q, \Sigma, \delta, q_0, F, \tau, B\}$$

where,

$Q = \{q_0, q_f\}$

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

$\tau = \{\#\}$

$q_0 = \{q_0\}$

$F = \{q_f\}$

$B = \{\#\}$

δ is given as:-

	0	1	#
q_0	$(q_0, \#, R)$	$(q_0, \#, R)$	$(q_f, \#, R)$
q_f	–	–	–

Verification: for #01o1#

q_0 #01o1#
⊢ #$q_0$1o1#
⊢ ##q_0o1#
⊢ ###$q_0$1#
⊢ ####q_0#
⊢ #####q_f

Verified

Tutorial:

Tutorial:

1. Design a TM that recognizes the language of all strings of even length over alphabet $\{a, b\}$.
2. Design a TM that accepts the language of all strings which contain aba as a substring.
3. Design a TM that recognizes the set of all strings of 0's and 1's containing at least one 1.
4. Design a TM that replace every 0 and 1 with every 1 and 0 in a binary string.
5. Design a TM which works as eraser.
6. Design a TM for the RE $r = aa^*$
7. Design a TM for the language $L = \{(ab)^n \mid n \geq 0\}$
8. Design a TM which accepts the language $L = \{w \in (a,b)^* \mid w$ has equal number of a's and b's.
9. Design a TM that accepts the language $L = 1^n 2^n 3^n \mid n \geq 0$ [2015 Fall]
10. Design a TM that accepts the language $L = \{w \in \{a,b\}^* \mid w$ has equal number of a's and b's$\}$.
11. Design a TM that accepts $L = \{p^n q^m r^n : m, n \geq 0\}$ [2014 Fall]

5.9 TM for computing functions

- A TM can be used to compute function.
- The input string W is presented in the form of #W#.
- The head of TM is placed positioned at the blank symbol which immediately follows the string W.
- We use an underscore to show the current position of machine head in the tape.
- A TM is said to halt on input W if we can reach to a halting state 'h' after performing some operations.
- A TM, say $T_m = (Q, \Sigma, \tau, \delta, q_0, h, B)$, is said to halt on input W if and only if $(q_0, \#W\#)$ yields to $(h, \#\underline{u}\#)$.

5.10 Definition of TM for Computing function

A function $f(x) = y$ is said to be computable by a TM M if $(q_0, \#W\#) \xrightarrow{*}_M (h, \#y\#)$.

5.11 Numerical: TM for Computing Function

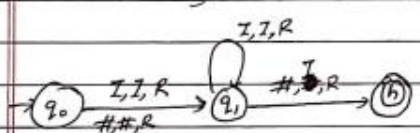
1. Design a TM which computes the function $f(n) = n+1$ for each $n \in N$. [2013 fall, 2018 Spring]

Sol^n: Given function is $f(n) = n+1$

Let us represent input n on the tape by a number of I on the tape.

If $n=1$, input will be #I#, and output will be #II#
If $n=2$, input will be #II#, and " " " #III#
If $n=3$, input will be #III#, " " " #IIII#

The state diagram is :-



Hence, TM is :-

$M = \{Q, \Sigma, \tau, \delta, q_0, F, B\}$

where,

$Q = \{q_0, q_1, h\}$

$\Sigma = \{I, \#\}$

$\tau = \{I, \#\}$

$q_0 = \{q_0\}$

$F = \{h\}$

δ is given by :-

δ	I	#
q_0	(q_1, I, R)	$(q_1, \#, R)$
q_1	(q_1, I, R)	(h, I, R)
h		

Verification :

q_0 #III#
$\vdash$ #q_1III#
$\vdash$ #Iq_1II#
$\vdash$ #IIq_1I#
$\vdash$ #IIIq_1#
$\vdash$ #IIIq_1I
$\vdash$ #IIIhI

Hence verified

2. Prove that following function is computable: $f(n) = n+2$

Sol^n: If any function is computable, then there exist a TM for it. So, it will be sufficient to construct a TM to prove any function computable.

Given function is $f(n) = n+2$

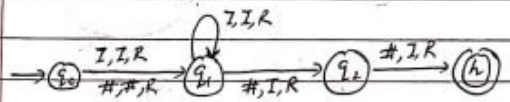
Let us represent n on the tape by number of I on the tape.

If $n = 1$, input will be #1#, then output will be #III#

If $n = 2$ " " #II#, " " " " #IIII#

If $n = 3$ " " #III#, " " " " #IIIII#

The state diagram would be:-



Here, TM is:- $M = \{Q, \Sigma, \tau, \delta, q_0, F, B\}$

Where,

$Q = \{q_0, q_1, q_2, h\}$

$\Sigma = \{I, \#\}$

$\tau = \{I, \#\}$

$q_0 = \{q_0\}$

$F = \{h\}$

$B = \{\#\}$

δ is given by:-

δ	I	#
q_0	(q_1, I, R)	$(q_1, \#, R)$
q_1	(q_1, I, R)	$(q_2, \#, R)$
q_2	–	(h, I, R)
h	–	–

Verification:

q_0 #III####

$\vdash$ q_1 #III####

$\vdash$ q_1 #III####

$\vdash$ q_1 #III####

$\vdash$ q_1 #III####

$\vdash$ q_2 #IIII#

$\vdash$ h #IIIII#

Hence, given function is computable.

3. Design a TM for computing a function $f(w) = w\#w$

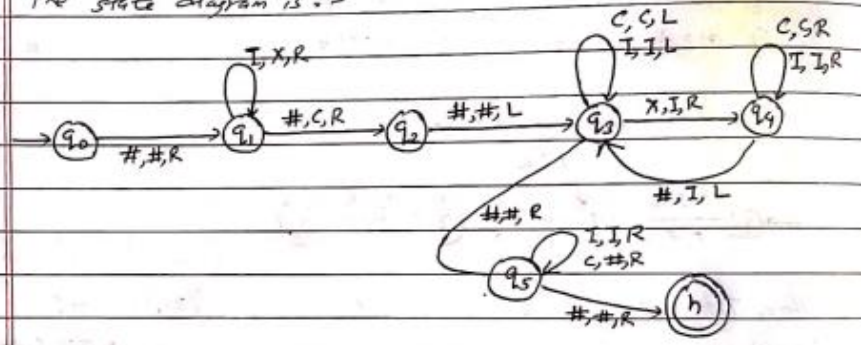
Sol^n: Here,

We have to design a TM that takes w and gives output w#w.

Let us assume, w as #III#,

then, the output should be #III#III#

The state diagram is:-



Hence, the TM is:-

$M = \{Q, \Sigma, \tau, \delta, q_0, F, B\}$

Where,

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, h\}$

$\Sigma = \{I, \#\}$

$\tau = \{X, I, C, \#\}$

$B = \{\#\}$

$q_0 = \{q_0\}$

$F = \{h\}$

δ is given by:-

	I	#	X	C
q_0	–	$(q_1, \#, R)$	–	–
q_1	(q_1, X, R)	(q_2, C, R)	–	–
q_2	–	$(q_3, \#, L)$	–	–
q_3	(q_3, I, L)	$(q_5, \#, R)$	(q_4, I, R)	(q_2, C, L)
q_4	(q_4, I, R)	(q_3, I, L)	–	(q_4, C, R)
q_5	(q_5, I, R)	$(h, \#, R)$	–	$(q_5, \#, R)$
h	–	–	–	–

Verification: for #II#

1. $\vdash q_0 \#II\#\#\#$
2. $\vdash q_1 \#I\underline{I}\#\#\#$
3. $\vdash q_1 \#XI\#\#\#$
4. $\vdash q_1 \#XX\underline{\#}\#\#$
5. $\vdash q_3 \#X\underline{X}C\#\#$
6. $\vdash q_4 \#X\underline{I}C\#\#$
7. $\vdash q_4 \#XIC\underline{\#}\#$
8. $\vdash q_3 \#X\underline{I}CI\#\#$
9. $\vdash q_3 \#X\underline{I}CI\#\#$
10. $\vdash q_3 \#\underline{X}ICI\#\#$
11. $\vdash q_4 \#I\underline{I}CI\#\#$
12. $\vdash q_4 \#II\underline{C}I\#\#$
13. $\vdash q_4 \#IIC\underline{I}\#\#$
14. $\vdash q_4 \#IICI\underline{\#}\#$
15. $\vdash q_3 \#IIC\underline{I}I\#$
16. $\vdash q_3 \#I\underline{I}CII\#$
17. $\vdash q_3 \#I\underline{I}CII\#$
18. $\vdash q_3 \#\underline{I}ICII\#$
19. $\vdash q_3 \underline{\#}IICII\#$
20. $\vdash q_5 \#\underline{I}ICII\#$
21. $\vdash q_5 \#I\underline{I}CII\#$
22. $\vdash q_5 \#II\underline{C}II\#$
23. $\vdash q_5 \#II\#\underline{I}I\#$
24. $\vdash q_5 \#II\#I\underline{I}\#$
25. $\vdash q_5 \#II\#II\underline{\#}$
26. $\vdash h \#II\#II\#$

Hence accepted

Note: Take #I# for verification to minimize time consumption

Tutorial

1. Design a TM that accepts every 0 and 1 with even 1 and any 0 in a binary string.
2. Prove that the function $f(\omega) = \overline{\omega}$ is computable where $\omega \in \{0,1\}^*$ and $\overline{\omega}$ the one's complement of ω.
3. Construct a TM that computes the following function $f(n,m) = n+m$
4. Design a TM which works as eraser.
5. Prove that following function is turing computable:

a) $$f(n) = \begin{cases} n-2 & \text{if } n > 2 \\ 1 & \text{if } n \leq 2 \end{cases}$$

b) $$f(n) = \begin{cases} n-1, & n > 0 \\ 0, & n = 0 \end{cases}$$

6. Design a TM for the following function: $f(x,y) = y$
7. Design a TM for the following function $f(x,y) = x$
8. Design a TM for the regular expression $r = aa^*$
9. Design a TM for the language $L = \{(ab)^n \mid n \geq 0\}$
10. Design a TM which accepts the language $L = \{\omega \in (a,b)^+ \mid \omega$ has equal no of a's and b's$\}$.
11. Design a TM which accepts the language $L = \{\omega \in (a,b,c)^+ \mid \omega$ has equal no of a's, b's and c's$\}$.
12. Design a TM for the following language $L = \{ab^n \mid n > 0\}$
13. Design a TM which computes following function. $f(\omega) = \omega\omega^R$, where ω^R is the reverse of string ω, $(\omega \in (a,b)^+)$
14. Design a TM which works as copying machine(C), for $\omega \in (a,b)^+$.

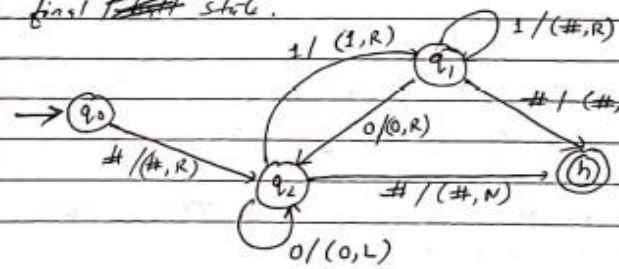
5.12 TM as Transducer

It consists of :-
- set of nodes representing states of TM.
- an arc from any state q to p is labelled by the items of the form X / Y D where X and Y are tape symbols
  D is a direction either Left or Right
  ie. $\delta(q, x) = (p, Y, D)$

Eg: Let TM = $\{Q, \Sigma, \tau, \delta, q_0, F, B\}$ where,
$Q = \{q_0, q_1, q_2, h\}$
$\Sigma = \{0, 1\}$
$\tau = \{0, 1, \#\}$ and
δ is given by the following transition table :-

	0	1	#
q_0	–	–	$(q_1, \#, R)$
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	$(h, \#, N)$
q_2	$(q_2, 0, L)$	$(q_1, 1, R)$	$(h, \#, N)$
h	–	–	–

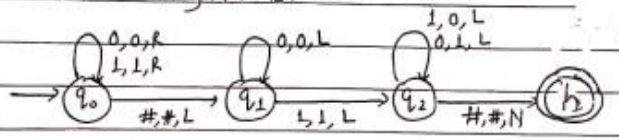
The transition diagram for above TM will be as shown below, where we assume that q_0 is the initial state and h is a final state.



5.13 Numerical: TM as Transducer

Design a TM that converts a string of 0 and 1 to its equivalent 2's complement.

The state diagram is:-



0110
1010

Here, TM is as:-
$M = \{Q, \Sigma, \tau, \delta, q_0, F, B\}$
where,
$Q = \{q_0, q_1, q_2, h\}$
$\Sigma = \{0, 1\}$
$\tau = \{0, 1, \#\}$
$q_0 = \{q_0\}$
$F = \{h\}$
$B = \{\#\}$
δ is given as:-

	0	1	#
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	$(q_1, \#, L)$
q_1	$(q_1, 0, L)$	$(q_2, 1, L)$	–
q_2	$(q_2, 1, L)$	$(q_2, 0, L)$	$(h, \#, N)$
h	–	–	–

Verification for : 0110

$q_0\#0110\#$
$\vdash q_0\#0110\#$
$\vdash q_0\#0110\#$
$\vdash q_0\#0110\#$
$\vdash q_0\#0110\#$
$\vdash q_1\#0110\#$
$\vdash q_2\#0110\#$
$\vdash q_2\#0010\#$
$\vdash q_2\#1010\#$
$\vdash h\#1010\#$

Hence transduced and accepted.

5.14 Extensions of TM

- TMs can perform fairly powerful computations.
- There are a lots of extensions we can make to our basic TM model. They may make it easier to write TM programs, but none of them increase the power of TM because we can show that every extended TM has an equivalent basic machine.
- The extensions of TM are:-

i) Multiple tapes TM:
The may be several tapes instead of only one, each tape have its own independent head.

ii) Two Way infinite tape.
The tape may be allowed to be infinite in both the directions.

(iii) Multiple head TM:
There may be more than one head scanning various cells of the tape.

Eg. ## b c #de f # # ---
(H2 under b, H1 under c)

δ (state, symbol under h_2, symbol under H_2)
= (new state, (S_1, M_1), (S_2, M_2))

(iv) K-dimensional Turing Machine.
The tape may be k-dimensional, $k \geq 2$, instead of only one dimensional.

v) Non-deterministic TM
For a given pair of current state and symbol under the head instead of at most one possible move, there may be any finite number of next moves.

vi) Random Access TM:
- A Random Access TM has fixed no. of registers and a one-way infinite tape
- Tape acts as Random Access memory chip.
- RAndom Access TM has a finite length program, composed of instructions with operators such as Read, Write, Load, Store, Add, Sub, Jump.
- The machine acts on its tape squares and its registers as ~~data~~ dictated by a fixed program.

5.14.1 Multi-tape Turing Machine

- It consists of several tapes.
- Each tape is connected to the finite control by means of a R/W head (one on each tape).

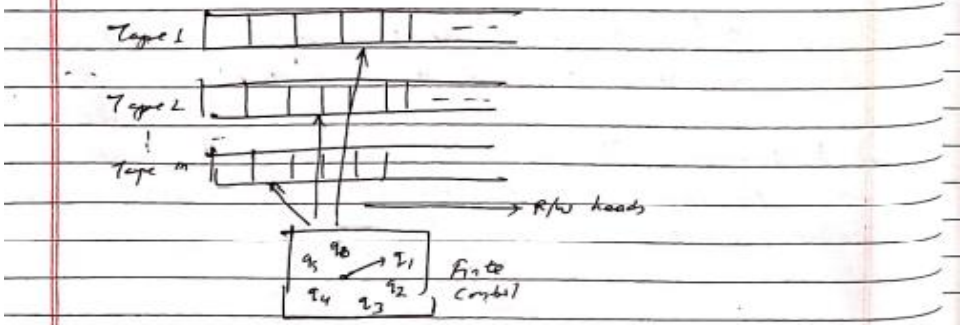


Fig: Multitape TM

- The machine can, in one step, read the symbols scanned by all its heads, ~~and then~~
- Depending on these symbols in current state, it can rewrite some of those scanned squares and move some of the heads to the left or right, ~~in addition to change~~ and also change the states.

Formal definition of m-tape TM:

Let $m \geq 1$ be an integer.

An m-tape TM is a six tuple machine as follows:-

$$T_m = (Q, \Sigma, \tau, \delta, q_0, h)$$

where,

Q : finite set of states of the finite control

Σ : finite set of input symbols

τ : tape symbol where $\tau = \Sigma \cup \#$

h : halt state $h \in Q$

q_0 : initial state

δ : transition function that maps:

$$(Q \times \Sigma^m) \rightarrow Q \times (\Sigma \cup \{L, R, N\}^m).$$

5.14.2 Two-Way Infinite Tape

Refer book: Page 248

5.14.3 Multi-Head TM

Refer book: Page 249

5.14.4 K-dimensional TM

Refer book: Page 250

5.14.5 Random Access TM.

Refer: cs.uwaterloo.edu/~nclimo/ear/automata/CS341-Fall-2004-Packet11-LectureNotes/23-24-TuringMachines-Handout.pdf

Theory of Computation and Formal Language – Ashesh Kr. Pandey

5.15 Non-Deterministic TM

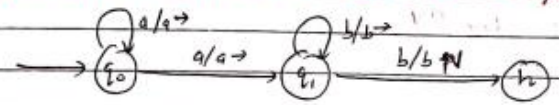
- It is powerful than multitape and Random Access TM due to its non-deterministic feature.
- It may contain certain combination of states and scanned symbol, and more than one possible choice of behavior.
- Non-deterministic machine can produce two different outputs or final states from the same input.
- The figure below shows a NDTM.

- It is one which at any point in a computation may proceed according to several possibilities.
- ~~NDTM can choose~~ The transition function δ are subsets rather than single element of set $Q \times \tau \times \{R, L, S\}$.
- Here, transition function δ is such that for each state q and tape symbol X, $\delta(q, X)$ is set of triplets $\{(q_1, Y_1, D_1), (q_2, Y_2, D_2) \cdots (q_k, Y_k, D_k)\}$ where k is finite integer.

- The NDTM can choose, at each step, any of the triplets to be the next move.
- But it can't however pick a state from one, a tape symbol from another, and direction from yet another.
- The computation of a NDTM is a tree whose branches correspond to different possibilities for the machine.
- If some branch of the computation leads to the accept state, the machine accepts its input.
- If M_N is a NDTM, then there is a deterministic TM M_D such that $L(M_N) = L(M_D)$.

Construct a NDTM which accepts the language $\{a^n b^m : n \geq 1, m \geq 1\}$ that is Language of all strings over $\{a,b\}$ in which there is at least one 'a' and one 'b' and all a's precede all b's.

Sol:



Here TM = $\{Q, \Sigma, \tau, \delta, q_0, h\}$

where $Q = \{q_0, q_1, h\}$, $\Sigma = \{a,b\}$, $\tau = \{a, b, \#\}$

$q_0 = \{q_0\}$, $h = \{h\}$

δ is defined as:-

$\delta(q_0, a) = \{(q_0, a, R), (q_1, a, R)\}$

$\delta(q_1, b) = \{(q_1, b, R), (h, b, N)\}$

or.

	a	b
q_0	$\{(q_0,a,R), (q_1,a,R)\}$	—
q_1	—	$\{(q_1, b, R), (h,b,N)\}$
h	—	—

Here, ab, aabb are accepted by the NDTM.

End of Chapter 5

6.1 Introduction to undecidability

In the theory of computation, we often come across such problems that are answered either 'yes' or 'no'. The class of problems which can be answered as 'yes' are called solvable or decidable. Otherwise, the class of problems is said to be unsolvable or undecidable.

6.2 Decidable and Undecidable problems

- A problem is decidable if we can construct a TM which will halt in finite amount of time for every input and give answer as 'Yes' or 'No'. A decidable problem has an algorithm to determine the answer for a given input.

  Some of the decidable problems are mentioned below. There is an algorithm and TM to decide for the below problems:

  1. Equivalence of two regular Language
  2. Finiteness of regular Language
  3. Emptiness of CFL

A problem is undecidable if there is no TM which will always halt in finite amount of time to give answer 'yes' or 'No'. An undecidable problem has no algorithm to determine the answer for a given input.

Some of the undecidable problems are mentioned below. There is no TM which halts in finite amount of time and decide for below problems:-

1. Ambiguity of CFL
2. Equivalence of two CFL

Two popular undecidable problems are: The halting problem and the PCP (Post Correspondence Problems).

6.3 The Halting problem

- Halting means that the program on certain input will accept it and halt, or reject it and halt, and would never go into an infinite loop.
- Basically, halting means terminating.

- Halting problem is undecidability.
- It asks the question - "Is it possible to tell whether a given machine will halt for some given input"
- The answer is No. We can't design a generalized algorithm which can appropriately say that given a program, the machine will ever halt or not.

Example:

Input: A Turing Machine (TM) and input string w

Problem: Does the TM finish computing of the string w in a finite no. of steps?

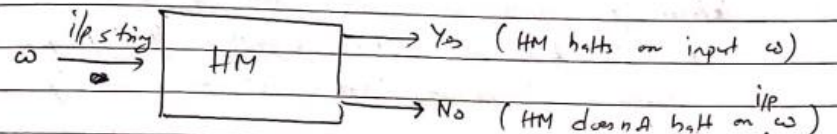
Proof:

Assume TM exists to solve this problem and then we will show it is contradicting itself.

Let us call this TM as a Halting machine that produce a 'Yes' or 'No' in a finite amount of time.

If the Halting machine (HM) finishes in a finite amount of time, the o/p comes as 'Yes' otherwise as 'No'.

The block diagram of HM B:-



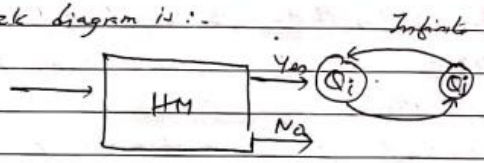
This allows us to design another machine.

Now let us design an inverted halting machine as:-

If it returns 'Yes', then loop forever

If it return 'No', then halt.

The block diagram is:-



Here, If HM halts on given input, then it loops forever.

But if HM doesn't halt, It returns 'No' and hence halts.

This can be depicted by an algorithm.

```
C(X)
    if (H(X,ω) == Halt)
        Loop forever;
    else
        Return;
```

If we run C on itself:

C(C)

H(C,C) == Halt → No Halt

H(C,C) == Not Halt → Halt

Hence, By contradiction, the Halting problem is undecidable.

6.4 Church's Thesis

- It is believed that there are no functions that can be defined by humans, whose calculation can be described by an well-defined mathematical algorithm that people can be taught to perform, that cannot be computed by TM.
- Church's Thesis consider the TM to be the ultimate calculating mechanism.
- This church Thesis was submitted by Alonzo Church (1936).
- The Church Thesis states that- "No computational procedure will be considered an algorithm unless it can be represented by a TM".
- Church actually said that any machine that can do certain list of operations, will be able to perform all conceivable algorithms.
- He tied together the idea of recursive functions and computable functions.
- However, Church Thesis cannot be a theorem because ideas such as "can ever be defined by humans" and "algorithm that people can be taught to perform" are not part of any known mathematics.

6.5 Universal TM

- A Universal TM (UTM) is a TM that can simulate the behavior of an arbitrary TM over any set of input symbols.
- The UTM achieves this by reading both the description of machine to be simulated as well as the i/p from its own tape.

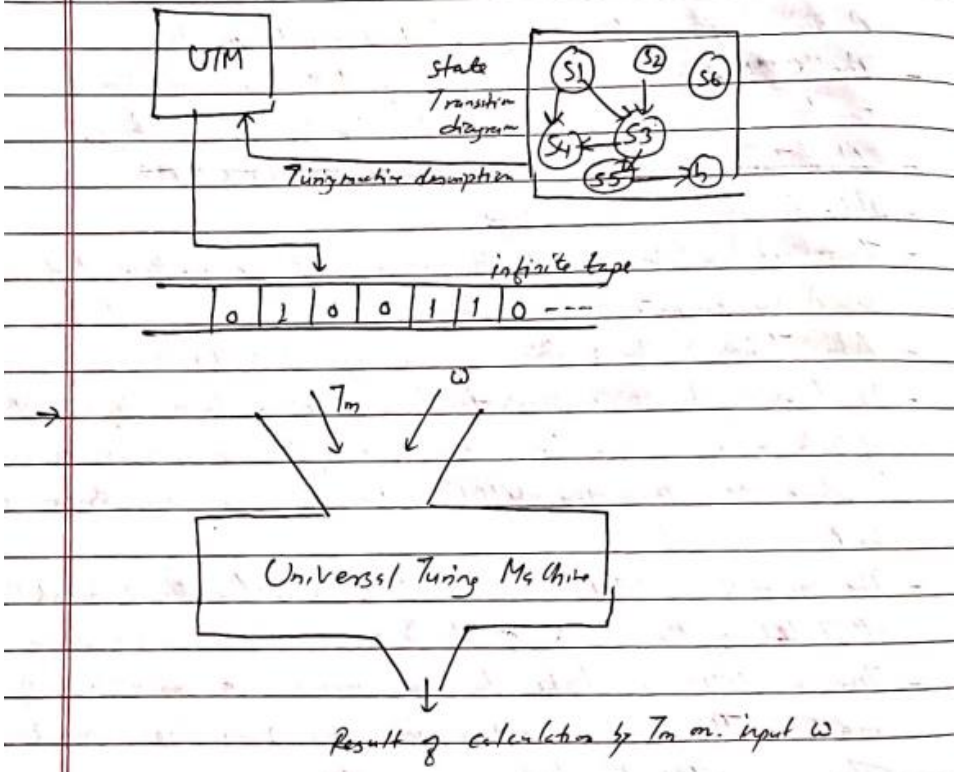
- Thus, it can be possible to create a single machine that can be used to compute any computable sequence.
- Alan Turing introduced this machine in 1936-1937.
- If this machine is supposed to be supplied with the tape on the beginning of which is written the input string of quintuple of some computing machine M, then the UTM will compute the same strings as those by M.
- The model of a UTM is considered to be a theoretical breakthrough that led to the concept of stored programmed computing device.
- Thus a UTM 'U' takes two arguments, a description of a machine TM, "Tm" and a description of an input string ω, "ω". Then the UTM has following property:-

  U halts on input "Tm" "ω" iff Tm halts on input ω.

  ie $U("m" "\omega") = "m(\omega)"$

It is the functional notation of UTM.

- The UTM simulates an other machine as shown below:-



- We can modify our basic TM and design a UTM by:-
  - increasing the no. of R/W heads
  - Increasing the no. of dimensions of input tape
  - Adding a special purpose memory.

6.6 Undecidable problems about TM

- The problems for which no algorithms exist are called Undecidable or Unsolvable. One famous undecidable problem is Halting Problem.
- The following problems about TM are undecidable:

a) Given a TM, M and an input string w, does M halt on input w?
b) Given a TM, M, does M halt on the empty tape?
c) Given a TM, M, is there any string at all on which M halts?
d) Given a TM M, does M halts on every input string?
e) Given two TMs, M_1 and M_2, do they halt on the same input strings?
f) Given a TM M, is the language that M semidecides regular? Is it context free? Is it recursive?

6.7 Encoding of TM

- It is a process of formulating a notation system where we can encode both an arbitrary TM, TL and an input string x over an arbitrary alphabet as strings e(TL) and e(x) over some fixed alphabet.
- This encoding must not destroy any original information. ie. we must be able to reconstruct TM TL and string x.
- For encoding of TM, we use alphabets {0,1}, although TM may have much larger alphabet

Steps for Encoding:

1. Start by assigning positive integer to each state, each tape symbol and each of three directions in TM that we want to encode.
2. Represent a state or a symbol by a string of 0's of appropriate length. Here 1's are used as separator.
3. For transition rule, use encoding function(S).

   q_i: $\delta(q_i, a_j) = (q_k, a_l, D_m)$

   encoded as:

   $S(q_i) 1 S(a_j) 1 S(q_k) 1 S(a_l) 1 S(D_m)$ - say n_1
4. Separate the entire transition rules by pair of 1's.

   ie. $n_1 11 n_2 11 n_3 11 \cdots n_n$
5. Now code for TM and i/p string x will be formed by separating them by three consecutive 1's.

   ie. e(TM) 111 e(x)

Example:

Encode a TM, $T = (Q, \Sigma, \tau, \delta, q_1, F, B)$ where
$Q = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $\tau = \{a, b, B\}$, $q_1 = \{q_1\}$,
$F = \{q_3\}$, $B = \{B\}$, and δ is:-

$\delta(q_1, b) = (q_2, a, R)$
$\delta(q_2, a) = (q_1, b, R)$
$\delta(q_2, b) = (q_2, a, R)$
$\delta(q_3, B) = (q_3, b, L)$ and input string $x = ab$

Sol^n: Here,

Step 1:

Here, $\Sigma = \{a, b\}$
let $q_1 = a$
$q_2 = b$

Step 2:

Let us represent states and symbols by a string of 0's of appropriate length.

$S(B) = 0$
$S(q_i) = 0^{i+1}$ for each $q_i \in \Sigma$
$S(q_i) = 0^{i+2}$ for each $q_i \in Q$
$S(N) = 0$
$S(L) = 00$
$S(R) = 000$

Using encoding function S as defined above:

$S(q_1) = 000$	$S(a) = S(q_1) = 00$	$S(N) = 0$
$S(q_2) = 0000$	$S(q_2) = S(b) = 000$	$S(L) = 00$
$S(q_3) = 00000$	$S(B) = 0$	$S(R) = 000$

Step 3: Using encoding function for transition rules

$e(m_1) = S(q_1)\ 1\ S(b)\ 1\ S(q_3)\ 1\ S(a)\ 1\ S(R)$
$= 000\ 1\ 000\ 1\ 00000\ 1\ 00\ 1\ 000$

$e(m_2) = S(q_2)\ 1\ S(a)\ 1\ S(q_1)\ 1\ S(b)\ 1\ S(R)$
$= 0000\ 1\ 00\ 1\ 000\ 1\ 000\ 1\ 000$

$e(m_3) = S(q_3)\ 1\ S(b)\ 1\ S(q_2)\ 1\ S(a)\ 1\ S(R)$
$= 00000\ 1\ 000\ 1\ 0000\ 1\ 00\ 1\ 000$

$e(m_4) = S(q_3)\ 1\ S(B)\ 1\ S(q_3)\ 1\ S(b)\ 1\ S(L)$
$= 00000\ 1\ 0\ 1\ 00000\ 1\ 000\ 1\ 00$

Step 4: Code for TM, T is:

$e(T) = e(m_1)\ 11\ e(m_2)\ 11\ e(m_3)\ 11\ e(m_4)$
$= 000\ 1\ 000\ 1\ 00000\ 1\ 00\ 1\ 000\ 11$
$000\ 1\ 00\ 1\ 000\ 1\ 000\ 1\ 000\ 11$
$0000\ 1\ 00\ 1\ 000\ 1\ 000\ 1\ 000\ 11$
$00000\ 1\ 0\ 1\ 00000\ 1\ 000\ 1\ 00$

Step 5: Now for T and and any input string x where $x = ab$, code will be: $e(T)\ 111\ e(x)$

Here, $e(x) = S(a)\ 1\ S(b)$
$= 00\ 1\ 000$

Hence,

$e(T)\ 111\ e(x) = 000\ 1\ 000\ 1 \ldots\ 1\ 000\ 111\ 00\ 1\ 000$

6.8 Recursive and Recursively Enumerable Language

When a TM executes an input, there are four possible outcomes of execution. Then TM

i) Halts and accept the input
ii) Halts and rejects the input
iii) Never halts (fall into loop), or
iv) Crash.

Recursive Enumerable Language (RE):

- also called type-0 Language, or TM recognizable language.
- RE languages - are those languages that can be accepted by TM
- Strings that are not in the language may be rejected or may cause the TM to go into an infinite loop.
- A language is Recursively Enumerable if there exists a TM that accepts every string of the language, and does not accept strings that are not in the language.
- RE Languages: are the superset of Recursive Language.
- Every Recursive Language is RE language but not vice-versa.

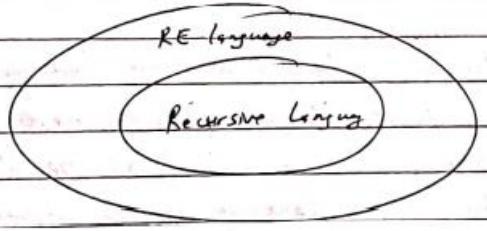


Fig: Relationship between RE and REC language

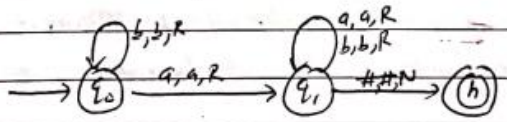
Recursive Language (REC Language):

- A language is recursive if there exists a TM that accepts every strings of the language and rejects every strings that are not in the language.
- A REC language is the subset of RE language.
- A REC language is decidable by TM, which means it will enter into final state for acceptable strings and into rejecting state for non-acceptable strings.
- So, the TM will always halt in this case.
- For eg: $L = \{ a^n b^n c^n \mid n >= 1 \}$ is recursive language because we can construct a TM which will move to final state if the string is of the form $a^n b^n c^n$ else move to non-final state.

Example:

Let $L = \{ w \in \{a,b\}^* : w$ contains at least one 'a'.

Then we can design a TM for L as:-



This machine scans to the right to find one 'a'.
If no 'a' is found, it goes forever, never halting.
It halts only if there is at least one 'a'.
Therefore, given Language is Recursively Enumerable.

6.9 Turing Recognizable language

- can run forever without deciding

A Language L is Turing recognizable if there exists a TM M such that for all strings ω:
- If $\omega \in L$, eventually M enter q_{accept}
- If $\omega \notin L$, either M enters q_{reject} or M never terminates

6.10 Turing decidable language

- Always terminates.

A Language L is turing decidable if there exists a TM M such that for all strings ω:
- If $\omega \in L$, M enters q_{accept}
- If $\omega \notin L$, M enters q_{reject}

6.11 Properties of Recursive and Recursively Enumerable Language

1. The complement of a Recursive language is recursive
2. The union of two recursive language is recursive
3. The union of two RE language is recursively enumerable
4. If a language L and its complement $\bar{L}$ are both recursively enumerable, then L and $\bar{L}$ here $\bar{L}$ is recursive.
5. If L is recursive language then $\Sigma^* - L$ is recursive.

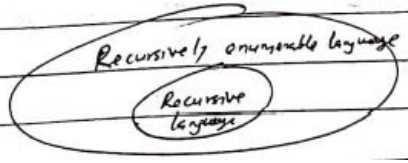
6.12 Theorem Proof

1. If a language is recursive then it is recursively enumerable.

A Language L is recursive if a TM accepts every strings of the Language ~~and reject~~ then enters into final state, and rejects every strings that are not in the Language then enters into rejecting state.

A language L is recursively enumerable if a TM accepts every strings of the language then enters into final state, and rejects every string that are not in the Language then may enter into rejecting state or may loop forever.

Hence, we can say that ~~every~~ recursive language is also a recursively enumerable. This is shown below with a relationship diagram.



13

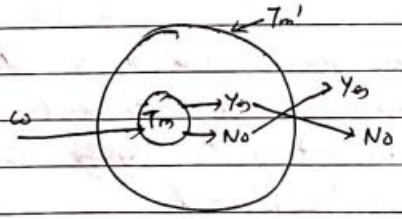
2. The complement of a Recursive Language is recursive.

Proof:- Let L → Recursive language

Tm → Turing machine that halts on all inputs and accepts L

Let us construct a TM Tm' from Tm so that if Tm enters a final state on input ω, then Tm' halts without accepting.

If Tm halts without accepting, Tm' enters a final state



Since, one of these two events occurs, Tm' is an algorithm. So clearly, T(Tm') is the complement of L and thus the complement of L is recursive language.

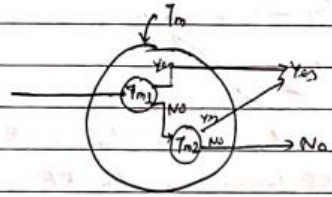
3. The union of two recursive language is recursive.

Proof: Let, L_1 and L_2 be two recursive language accepted by TM T_{M1} and T_{M2}

Let us construct a TM T_m that first simulates T_{M1}.

If T_{M1} accepts, then T_m accepts

If T_{M1} rejects, then T_m simulates T_{M2} and accepts if and only if T_{M2} accepts.



Since both T_{M1} and T_{M2} are algorithm, so T_m is guranteed to halt.

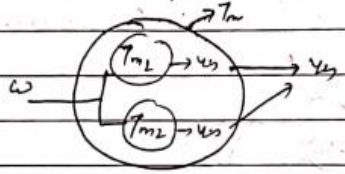
Clearly T_m accepts $L_1 \cup L_2$

Hence The union of two recursive language is also recursive.

4. The union of two recursively enumerable language is recursively enumerable.

Proof: Let L_1 and L_2 are recursively enumerable language and their enumerative TM are T_{M1} and T_{M2} respectively.

Let us construct a TM T_m which can simulate T_{M1} and T_{M2} simultaneously on separate tape.



Here, If either T_{M1} or T_{M2} accepts, T_m also accepts.

Thus union of two RE language is also RE.

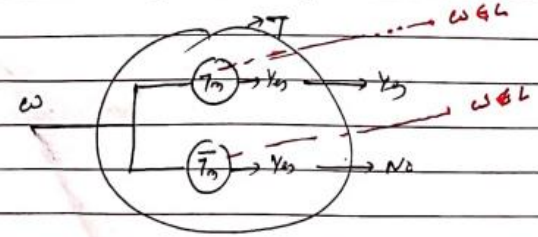
5. If a Language L and its complement L' are both recursively enumerable, then L (and hence L') is recursive.

Proof: Let, T_M and $\overline{T_M}$ accept L and $\overline{L}$ respectively.

Let us construct a TM T which simulate T_m and $\overline{T_m}$ simultaneously.

T accepts w if T_m accepts w, and

T rejects w if $\overline{T_m}$ accepts w

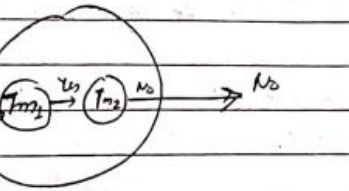


Thus T will always say either 'Yes' or 'No' but never says

both. Since T_M is algorithm that accepts L, it follows that L is recursive.

6. If L is recursive then Σ^*-L is recursive.

Proof: The required T_M-complement can be represented by as:-



The machine T_M-complement functions as follows:-

- When a string $W \in \Sigma^*$ is given as input to T_M-complement, its control passes the string to T_{M_1} as input
- As T_{M_1} decides the language L, therefore, for $w \in L$ after a finite no. of moves, T_{M_1} outputs "Yes" which is given as input to T_{M_2}, which in turn returns "No"
- Similary for $w \notin L$, T_{M_1} returns "Yes".

Hence there exists a T_M-complement for Σ^*-L. So it is turing decidable, that is recursive.

Tutorial:

Prove:

1. Intersection of two recursive language is also recursive.
2. Intersection of two recursive enumerable language is also recursive enumerable.

6.13 Recursive Function Theory

1. Recursion

- A function which calls itself directly or indirectly and terminates after infinite no. of steps is known as Recursive function.
- In recursive function, terminating point is also known as base point.
- Each and every time, the function calls itself, it should be nearer to the base point.

2. Initial function for Natural numbers

Let $N = \{0, 1, 2, \ldots\}$ be a set of Natural numbers, We have three initial functions over N defined below:-

1) Zero function
   - denoted by Z and defined as:
   $Z(n) = 0$ for $\forall n \in N$ Eg: $Z(2) = 0$
2) Successor function
   - denoted by S and defined as.
   $S(n) = n+1$ for $\forall n \in N$
   Eg. $S(2) = 2+1 = 3$
3) Projection function: (P_i^n)
   - defined as:-
   $P_i^n(a_1, a_2, \ldots a_n) = a_i$ where
   where, $a_i \in N$ for $i = 1, 2, \ldots n$ and $i \leq n$

Projection function takes n arguments and returns their i^{th} argument.

3. Computation of function

We can define a new function by the combination of two or more functions. Such defined functions are called composition function.

For eg: $S(Z(q)) = S(0) = 1$

$S(S(Z(n))) = S(S(0)) = S(1) = 2$

4. Primitive Recursive function

A function is primitive if:

i) It is an initial function

ii) It is obtained from recursion or composition of initial functions.

5. Partial Recursive function

A partial function is recursive if:

i) It is an initial function over N, or

ii) It is obtained by applying recursion or composition or minimization on initial function over N.

6. Total Recursive function

A ~~total recursive~~ function is said to be total recursive function if it is defined for all of its arguments.

Let $f(q_1, q_2, q_3 \dots q_n)$ be a function and defined on function $g(b_1, b_2, \dots b_m)$, then f is total function if every element of f is assigned to some unique element of function g.

End of Chapter 6

7.1 Computational complexity theory

- Computational Complexity theory is a branch of TOC in computer science & mathematics that focuses on classifying computational problems according to their inherent difficulty (or complexity).
- It includes :
  * the efficiency of algorithms
  * the inherent difficulty of problems of practical and/or Theoretical importance.
- It deals with the resources required during computation to solve a given problem.
- The most common resources are :
  * time (how many steps it takes to solve a problem)
  * space (how much memory it takes).
- Other resources may be; how many parallel processes are needed to solve a problem in parallel.
- Much of complexity theory deals with decision problems. A decision problem is a problem where the answer is always Yes / No.
- The goal of Complexity Theory is to classify decidable problems according to the amount of resources required to solve them. Space and Time are the two most common measures of complexity.

7.2 Computability theory

- Computability Theory deals only with whether a problem can be solved at all, regardless of the resources required.
- It deals primarily with the question of whether a problem is solvable at all on the computer.
- The statement that the halting problem cannot be solved by a TM is one of the most important results in computability theory.
- Computability theory addresses four main questions:
  1. What problems can TM solve?
  2. What other systems are equivalent to TMs?
  3. What problems require more powerful machines?
  4. What problems can be solved by less powerful machines?
- The computability theory doesn't ~~focus~~ deal with the time and space factor of a machine.

7.3 Space and Time Complexity

Space Complexity

- Space complexity is defined as the process of determining a formula for prediction of how much memory space will be required for the successful execution of the algorithm.
- The memory space we generally consider is the space of primary memory.
- Space complexity specifies the amount of temporary storage required for running the algorithm.

Time Complexity

- The time complexity is defined as the process of determining a formula for total time required towards execution of that algorithm.
- This calculation will be independent of implementation details and programming language.

7.4 Polynomial-Time and Exponential Algorithm

The basic idea for problem classification lies on whether there exists a polynomial-time (computer) algorithm that can provide the solution to a problem. The algorithms can be classified into two categories:- a) Polynomial-time algorithm
b) Exponential algorithm.

Polynomial-Time Algorithm

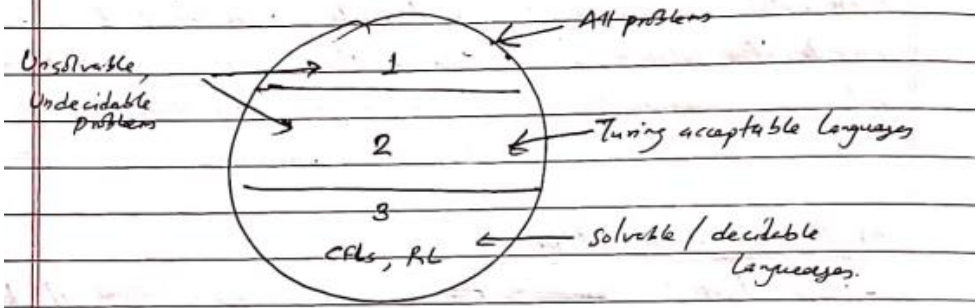
- Algorithms run in polynomial-time if for a problem of size n, the number of steps needed to find the solution is a polynomial function of n.
- Their order-of-magnitude time performance is bounded by a polynomial function of n, where n is the size of its input.
- Polynomial functions are like $O(1)$, $O(\log n)$, $O(n)$, $O(n \times \log n)$, $O(n^2)$, $O(n^3)$, and so on.
- Polynomial algorithms are considered to be efficient because the execution time grows less rapidly as the problem size increases.

Exponential Algorithm

- Exponential algorithms are not bounded by polynomial-time.
- Exponential functions are like $O(k^n)$, $O(n!)$, $O(n^n)$.
- Exponential functions are considered to be inefficient because the execution times of the latter grow much more rapidly as the problem size increases.

7.5 Intractable and Intractable problems

The figure below ~~never~~ generates the idea about the complexity of the problems:



1. Strictly unsolvable problems
2. Turing acceptable but not turing-decidable problems
3. Solvable / decidable problems

a) Tractable problems

- A decision problem is tractable if there is an algorithm to solve the given problem and time required is expressed as a polynomial $P(n)$, n being the length of the input string.
- Simply tractable problems are those if there exists an algorithm of polynomial complexity for all instances and solves the problem.
- The P and NP classes of problems are fall in tractable problems.

b) Intractable problems

- A problem will be said to be computationally intractable (also computationally complex or computationally hard) if the optimal algorithm for solving the problem cannot solve all of its instances in polynomial time.
- Usually problems are intractable if the time required for any of the algorithm (which can solve the problem) is at least $f(n)$, where f is an exponential function of n.
- NP-hard, and NP-Complete problems fall in this category.

Examples of Tractable problems:

1. Searching an unordered list
2. Searching an ordered list
3. Sorting a list
4. Multiplication of integers
5. Finding a minimum spanning tree in a graph

Examples of Intractable problems:

1. Towers of Hanoi
2. TSP

- Problems that are solvable in theory, but cannot be solved in practice are called intractable.

7.6 Tractable Problems: class P and class NP

a) Class P

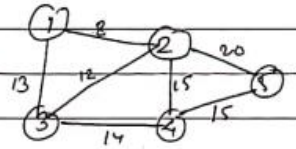
- A problem which can be solved on polynomial time is known as P-class problem.
- Eg: All sorting and searching algorithms
- P is a class of problems that can be solved deterministically in polynomial time.
- These problems can be solved in time $O(n^k)$ in worst-case, where k is constant.
- The class P is important because:
  - i) it is invariant over all models of computations
  - ii) Practical problems in P-class have efficient (low-degree polynomial) algorithms.
- Class P is also a sub-set of NP class, but not P=NP.
- The P-class problems are easy to solve and also easy to verify in polynomial time.

Examples pf P -class:

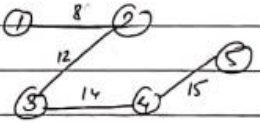
i) Kruskal's Algorithm: Minimum Weight Spanning Tree

A minimum weight spanning tree is a sub-graph that connects all the nodes (or vertices) together with least possible weight of the edges, yet there are no cycles.

Eg:
XYZ company is providing cable services to 5 new housing developments. The goal is to determine the most economical cable network. Find the minimal spanning tree for below:



The solution is (using Kruskal's algorithm):



This problem is class P problem because it is possible to implement Kruskal's algorithm (using computer) on a graph with m nodes and x edges in time $O(m + x \log x)$.

ii) Problem of finding the maximum number

Let us consider there are 5 numbers (n1, n2, n3, n4, n5).
The problem is to find the largest among them.
This can be solved as:-

```
int max;
    max = n1;
if (n2 > max)
        max = n2
if (n3 > max)
        max = n3
if (n4 > max)
        max = n4
if (n5 > max)
        max = n5
```

If there are N numbers in the list, this takes roughly N steps.
N is a polynomial function of N, so this problem is the example of P-class.

b) Class NP

- NP is the set of problems that can be solved in non-deterministic polynomial time. (but in exponential time)
- A problem which cannot be solved on polynomial time, but is verified in polynomial time is called known as Non-deterministic polynomial or NP-class problem.
- Eg: Su-Do-Ku, Prime factor, Scheduling, Travelling Salesman problems.
- For class NP problems, solving the problem is difficult, but verifying it is easy.
- The class NP is also invariant over all reasonal models of computation.

Example: The Su-Do-Ku Problem

Goal: Fill the 9X9 grid with numbers so that each row, column and 3x3 section contains all of the digits between 1 and 9 without repetition.

	8			7		3		5
2	5	7			8		9	6
					5		7	
4		9	3	6				
	6						4	
3	7			1			2	
	3				1	9		4
7		5				1	8	2
		6	4	8	2			

It is very difficult to solve the problem. It takes an exponential time to solve this problem.
However, it takes very less time to verify if the problem is solved or not.
Hence, it is a class NP problem.

7.7 P-class Vs NP-class

The P class problems, not only it is solved on polynomial time but it is verified also in polynomial time.

The NP class problems cannot be solved in exponential time, but it is verified in polynomial time.

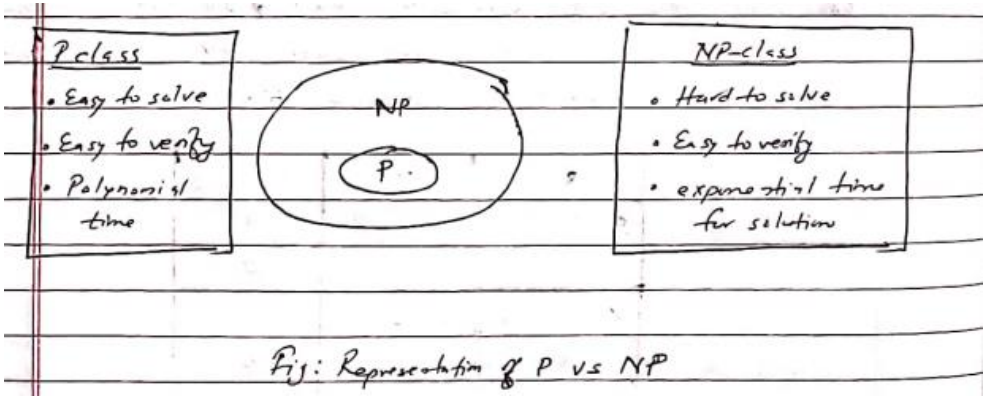


Fig: Representation of P vs NP

- If P=NP, then cracking nearly every kind of encryption would suddenly become much, much easier.
- The above is the disadvantage if P=NP.
- There would have been advantages also if P=NP.
- Everything would be more efficient such as transportation, scheduling, understanding DNA, etc.
- So if P=NP, then the world would be a profoundly different place than we usually assume it to be.

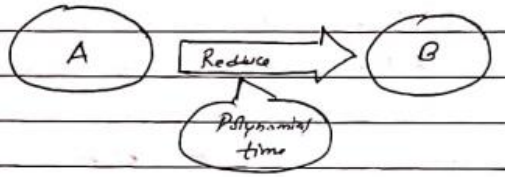
7.8 IS P=NP?

- This is one of the seven unsolved mathematical problems laid out by the Clay Mathematical Institute, each with a $1 million prize for those who solve them.
- Is P=NP? This simply asks whether computationally hard problems actually contain hidden, and computationally easy solutions.
- Proving that P ≠ NP would be a major milestone. If P ≠ NP then we could prove that there are some problems that can never be solved.
- The majority of modern cryptography relies on codes that are hard to crack but easy to check. The encryption behind securing our credit card number when ordering something on Amazon too is an example of NP cryptography.

7.9 Intractable Problems: NP-hard and NP-complete

Before discussing on NP-hard and NP-complete classes, Let us get into the topic "Polynomial-Time Reduction".

Polynomial Time Reduction



Let A and B are two problems.

- Then A reduces to B iff there is a way to solve A by deterministic algorithm that solve B in polynomial time.
- If A is reducible to B, we denote it by $A \propto B$

NP-hard and NP-Complete problems:

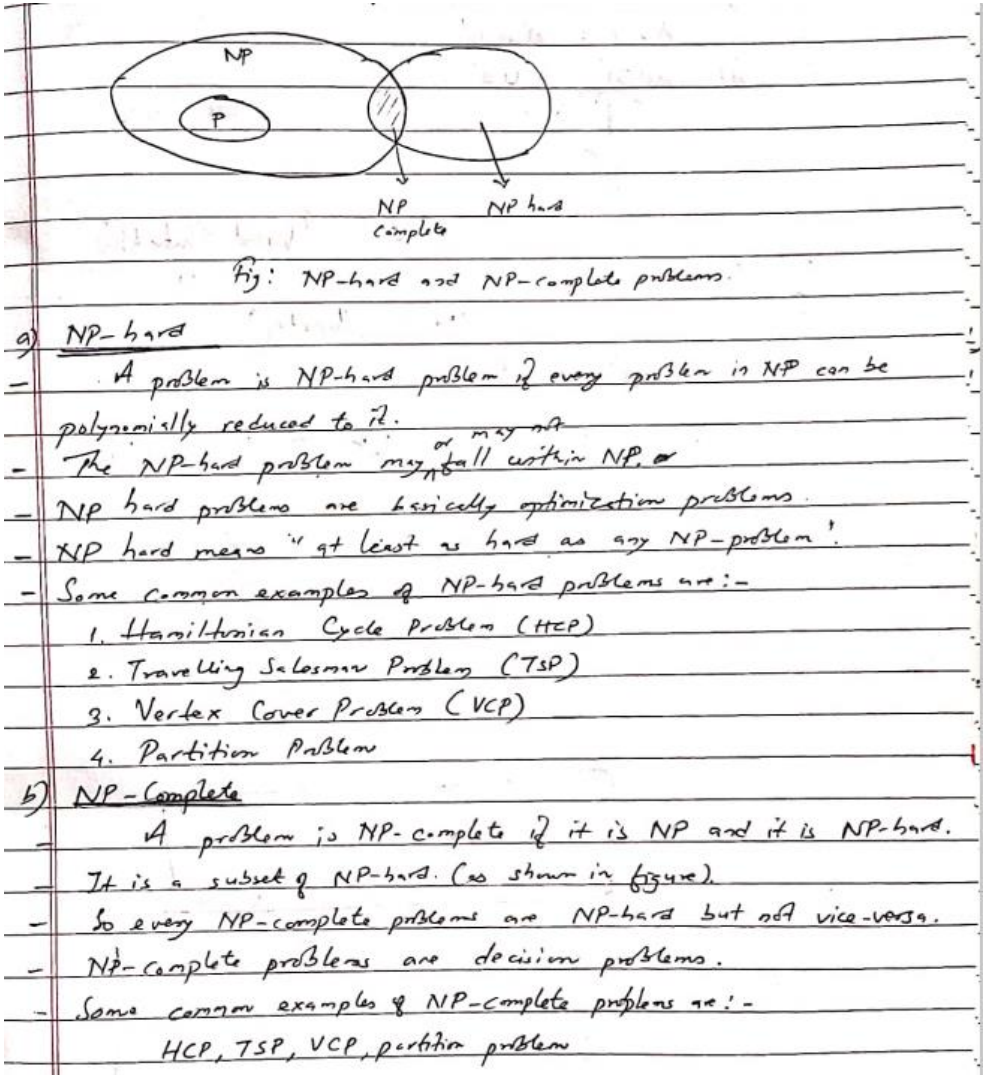


Fig: NP-hard and NP-complete problems

a) NP-hard

- A problem is NP-hard problem if every problem in NP can be polynomially reduced to it.
- The NP-hard problem may or may not fall within NP.
- NP hard problems are basically optimization problems.
- NP hard means "at least as hard as any NP-problem".
- Some common examples of NP-hard problems are:-
  1. Hamiltonian Cycle Problem (HCP)
  2. Travelling Salesman Problem (TSP)
  3. Vertex Cover Problem (VCP)
  4. Partition Problem

b) NP-Complete

- A problem is NP-complete if it is NP and it is NP-hard.
- It is a subset of NP-hard. (as shown in figure).
- So every NP-complete problems are NP-hard but not vice-versa.
- NP-complete problems are decision problems.
- Some common examples of NP-complete problems are:-
  HCP, TSP, VCP, partition problem

End of Chapter 7