

WELCOME TO  
THEORY OF  
COMPUTATION

Dr. Arshi Husain,  
Asst. Professor  
USICT, GGSIPU



TOC / TAFL Playlist:

<https://shorturl.at/xzulM>



arshihusaincs@gmail.com



<https://shorturl.at/4sLZo>

# Theory Of Computation

## What is Automata Theory?

**Automata theory** is the study of abstract machines and automata,& the computational problems that can be solved using them. It is the combinational study of theoretical computer science and discrete mathematics.

TOC is a branch of computer science that explores the fundamental capabilities and limitations of computers. It describes the various abstract models of computation, represented mathematically & focuses on understanding what problems can be solved by a computer, how efficiently they can be solved, and how to design systems to perform those computations. It focuses on:

1. **What can we compute?** (Decidability)
2. **How can we compute it?** (Models of Computation)
3. **How efficiently can we compute it?** (Complexity)

# Theory Of Computation

The Theory of Computation is divided into three main areas:

1. **Automata Theory:** Focuses on abstract machines (like finite automata and Turing machines) and how they process inputs.
2. **Formal Languages:** Studies the structure of languages that computers can understand, like programming languages.
1. **Computational Complexity:** Examines how hard or easy a problem is to solve, based on resources needed.



# Basic Terminology

**Automata:** It is a self-operating machine or abstract computational systems that follows predefined rules to process inputs and produce outputs. Eg, Elevator, Automatic doors etc

**Symbol:** Basic building blocks. Example: a,b,c,... , 0,1,2,3,...,9 and special characters.

**Alphabet :-** is a finite , nonempty set of symbols denoted by  $\Sigma$  in theory of computation. Example1-: $\Sigma = \{0,1\}$  Example2-: $\Sigma = \{a,b,c,\dots,z\}$

**String or Word:-** Finite sequence of symbols (Symbols could be chosen from alphabet) and denoted by symbol w (depends on writer) Example-:  $\Sigma = \{0,1\}$  Then 0101001 , 010, 00, 11, 111 ,0, 0011,..... etc are the string we can form by sequence of symbols. Then a,b,ab,aa,bb,aab,abb,....etc are the string we can write by sequence of symbol.

**Empty String :** Symbol ' $\epsilon$ ' (epsilon) is used to denote empty string and that is the string consisting of zero symbol. It is also called as null string

# Basic Terminology

**Length of String :** The length of a string is number of symbols in the string. It denoted by  $|w|$ .

**Example 1:** If  $\Sigma = \{0,1\}$  &  $w = 01010$  Then,  $|w| = 5$  **Example 2:** If  $w = \epsilon$  then  $|w| = 0$

**Prefix:** It is any number of leading symbol of string. Example:- let  $w = abc$  Prefix of  $w$  are  $\epsilon, a, ab, abc$ .

**Suffix:** Any number of trailing symbols of string. If  $w = abc$ , suffix are  $\epsilon, c, bc, abc$

**Substring:** Any sequence of symbol over the given string. Example:-  $w = abc$   
The substring of  $w$  are  $\epsilon, a, b, c, bc, ab, abc$ .

**Concatenation of String -:** if  $w$  and  $x$  are two string then the concatenation of two string is denoted by  $wx$ . Example:-  $w = 1110$  and  $x = 0101$  Then  $xy = 11100101$

# Basic Terminology

**Language**-: It is the set of string over the alphabet  $\Sigma$  which follows a set of rules called the grammar L Or otherwise we can say set of string generated by grammar taking symbol from  $\Sigma$  .

Example 1:-All string start with 11 from  $\Sigma=\{0,1\}$

$$L=\{110,111,1100,1111,1101,1110,\dots\dots\}$$

Example 2:-set of binary number whose value is prime.  $L=\{10,11,101,111,1011,\dots\dots\}$

Note-:  $L=\{\emptyset\}$  is called empty language. And  $L=\{\epsilon\}$  Language consisting of only the empty string.

**Grammar**-: A set of rules used to generate the string for a particular language.

# Power of an alphabet

If  $\Sigma$  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  $\Sigma^k$  and is the set of strings of length k.

Example: If  $\Sigma = \{0,1\}$

$$\text{Then } \Sigma^0 = \{\epsilon\}$$

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

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

$$\Sigma^3 = \{000,001,011,\dots,111\}$$

$\Sigma^n = \{ w , |w| = n \}$  using the symbols from the alphabet

Set of all strings from alphabet  $\Sigma$  of length exactly n

# Kleene Closure / Star $\Sigma^*$

$$\Sigma^* = \bigcup_{i=0}^{i=\infty} \{w \mid |w| = i\}$$

- Kleene Closure is the infinite set of all possible strings of all possible lengths including  $\epsilon$
- It is denoted by  $\Sigma^*$
- So  $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$
- For example over  $\Sigma = \{0,1\}$

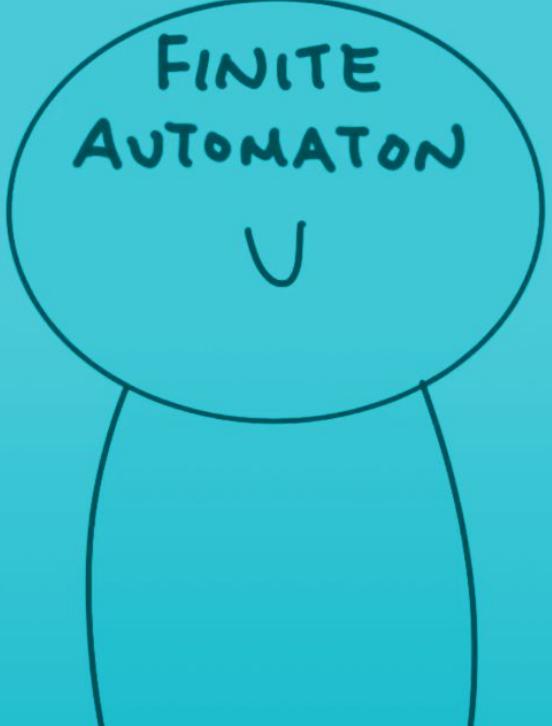
$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \dots\}$$

# Positive Closure $\Sigma^+$

- Positive closure is the infinite set of all possible strings of all possible lengths excluding  $\epsilon$
- It is denoted by  $\Sigma^+$
- So  $\Sigma^+ = \Sigma^* - \{ \epsilon \}$
- So  $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$
- For example over  $\Sigma = \{0,1\}$

$$\Sigma^+ = \bigcup_{i=1}^{i=\infty} \{w \mid |w| = i\}$$

# Finite Automata



FINITE  
AUTOMATON

# Finite Automata

- Finite Automata(FA) is the simplest machine to recognize patterns. It is a mathematical model of a system, with discrete inputs, outputs, finite states and set of transitions from state to state that occurs on input symbols from alphabet  $\Sigma$ .
- 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. They are widely used for Pattern matching, Lexical Analysis, Parsing and other applications.

It representations:

- Graphical (Transition/State Diagram or Transition Table)
- Tabular (Transition Table)
- Mathematical (Transition Function/ID or Mapping Function)

# Finite Automata

## Applications:

- ❑ In switching theory and design and analysis of digital circuits automata theory is applied.
- ❑ Design and analysis of complex software and hardware systems.
- ❑ It plays an important role in compiler design
- ❑ To prove the correctness of the program automata theory is used.
- ❑ It is base for the formal languages and these formal languages are useful of the programming languages.

# Finite Automata Model

The various components of the block diagram are as follows:

- 1) **Input tape:** The input tape is divided into cells, each cell containing a single symbol from the input alphabet  $\Sigma$ . The end cells of the tape contain the end marker  $\$$  at the left end and the end marker  $\$$  at the right end. The absence of end markers indicates that the tape is of infinite length. The left-to-right sequence of symbols between the two end markers is the input string to be processed

- 1) **Finite control:** It represents the "brain" or logic of the finite automaton. It decides the next state on receiving particular input from input tape.
- 1) **Reading head :** The head is unidirectional, it examines only one cell at a time and reads the cells one by one from left to right,

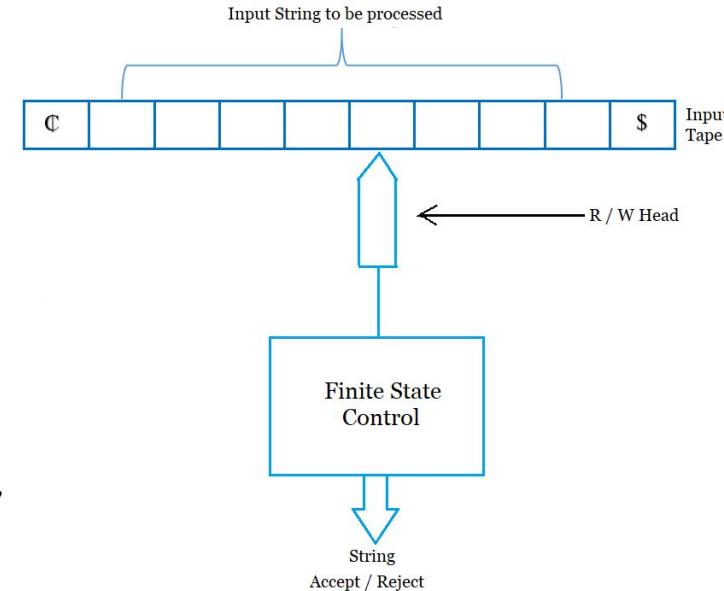


Fig: Block Diagram of Finite Automata

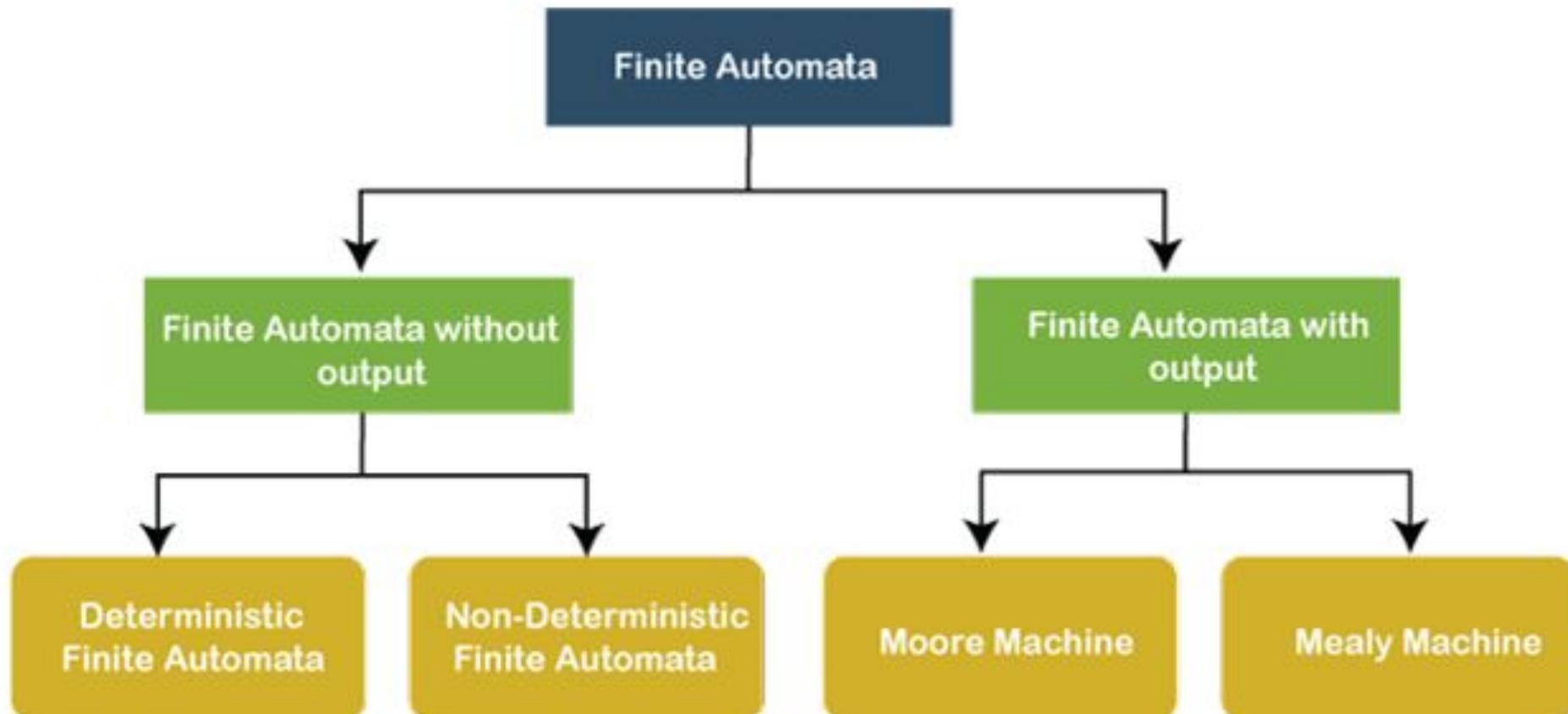
# Computation of Finite Automata

- The automata M receives the input string 'w' designed from the alphabet S.
- After reading each symbol, control moves from one state to another along the transition that has that symbol as its label.
- When M reads the last symbol of w, then the string 'w' is said to be accepted by the finite automata, if there exist a transition for which it starts at the initial state and ends in any one of the final states.,  $\delta^*(q_0, w) = q_f$  for some  $q_f \in F$ .

Mathematically, it can be represented as:

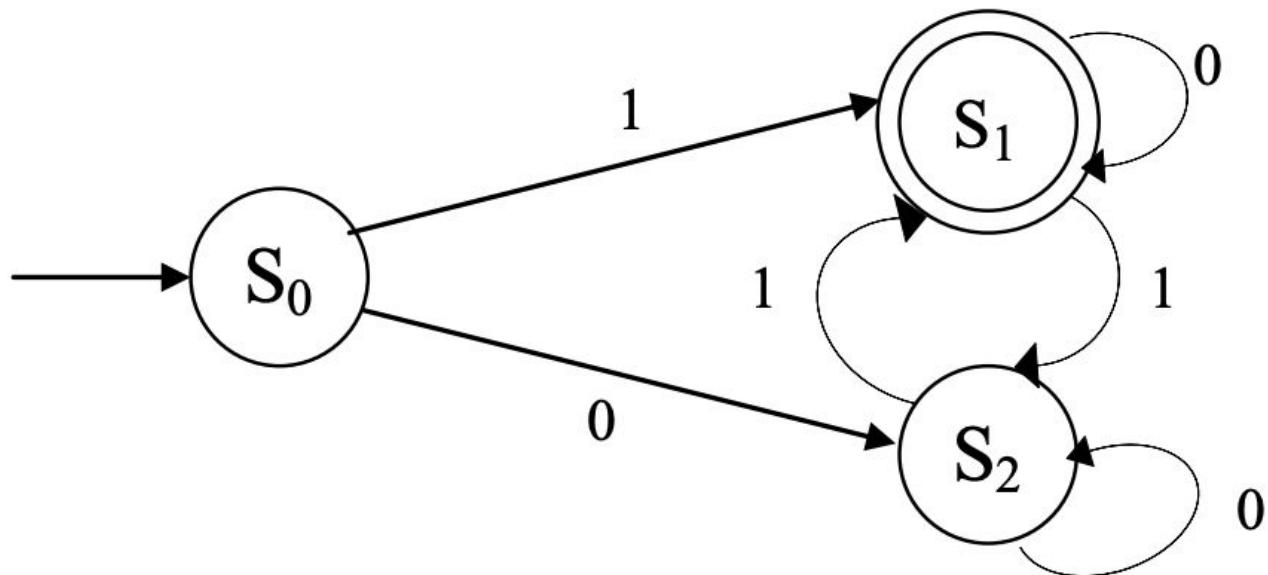
$$L(M) = \{w \in \Sigma^* \mid \delta^*(S, w) \in F\}$$

# Types of Finite Automata



# Deterministic Finite Automata (DFA)

Deterministic Finite Automaton is a FA in which there is exactly one path for a specific input from current state to next state. i.e., there is a unique transition on each input symbol.



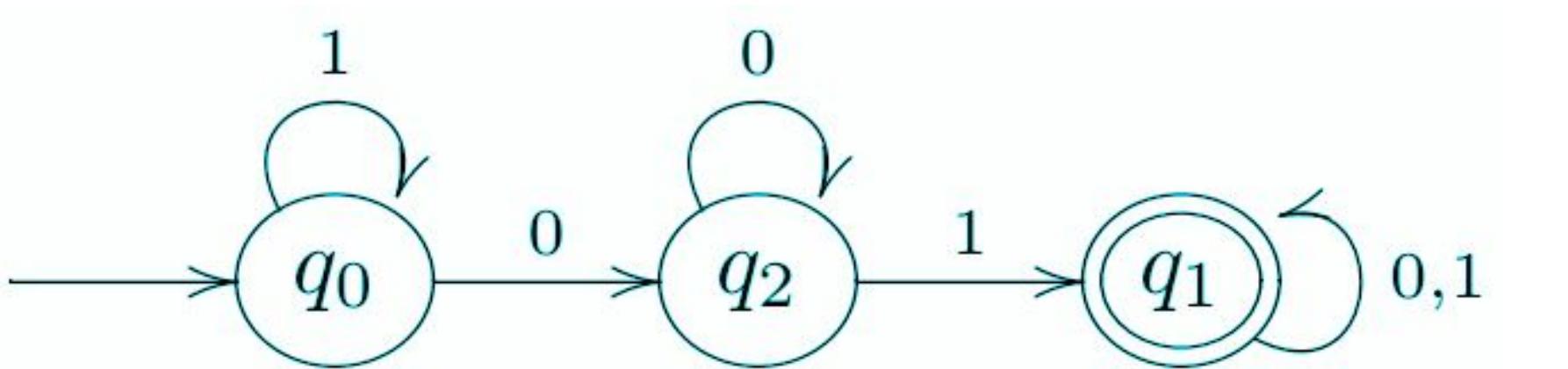
# Formal Definition of a DFA

A deterministic finite automaton (DFA) is defined by 5-tuple  $(Q, \Sigma, \delta, S, F)$  where:

- $Q$  is a finite and non-empty set of states
- $\Sigma$  is a finite non-empty set of finite input alphabet
- $\delta$  is a transition function,  $(\delta: Q \times \Sigma \rightarrow Q)$
- $S$  is initial state (always one) ( $S \in Q$ )
- $F$  is a set of final states ( $F \subseteq Q$ ) ( $0 \leq |F| \leq N$ , where  $n$  is the number of states)

# Representation

1. **Transition State Diagram (Transition graph):** It is a finite 5-tuple directed graph in which each circle represents a state and the edges represent the transition of one state to another state. The initial state is represented by a circle with an arrow pointing towards it, the final state by two concentric circles.



Start or Initial State

Final or Accepting State

# Representation (cont.)

2. **Transition Table:** It is a two-dimensional table where number of columns is equal to number of input alphabets and number of rows is equal to number of states.

States	INPUT	
	0	1
→ $q_0$	$q_1$	$q_0$
$q_1$	$q_2$	$q_1$
$q_2$	$q_2$	$q_2$

# Representation (cont.)

3. Transition ID/Function: The mapping/transition function is denoted by  $\delta$ .

Two parameters are passed to this transition function: (i) current state and (ii) input symbol. The transition function returns a state which can be called as next state.

$$\delta(\text{current\_state}, \text{current\_input\_symbol}) = \text{next\_state}$$

Example 1:  $\delta(q_0, a) = q_1$ , means on a state  $q_0$  take an input symbol  $a$  machine will make a transition  $q_1$

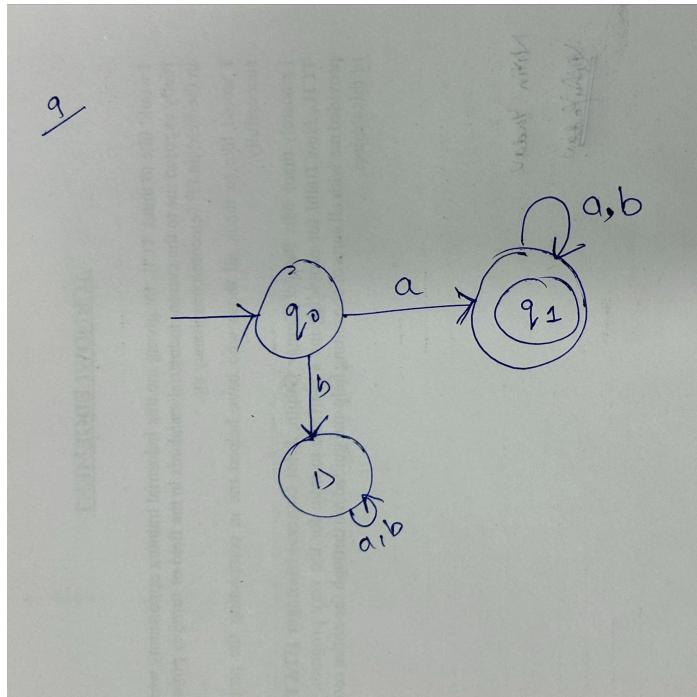
Example 2:  $\delta\{q_i, a\} = q_j$ , this means that on a state  $q_i$  take an input symbol  $a$  machine will make a transition  $q_j$

# Practice Questions

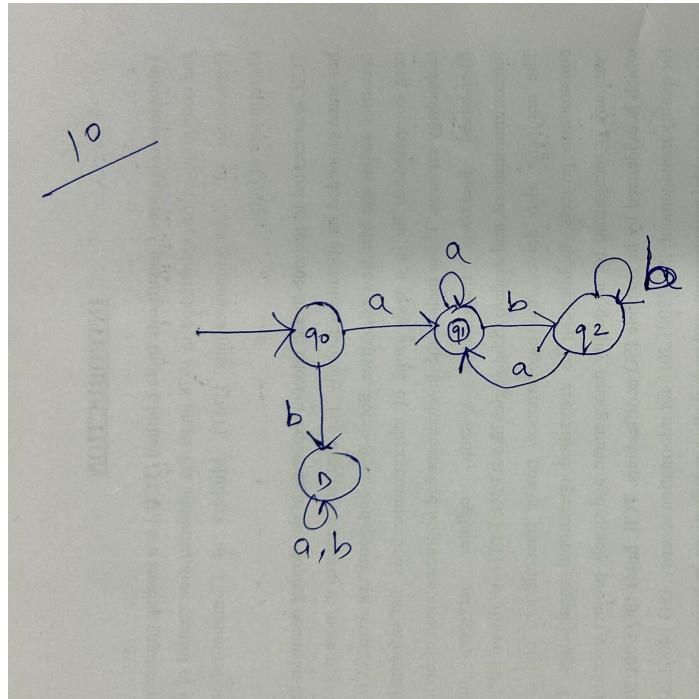
Design a minimal DFA that accepts

- 1)  $L = \{\epsilon, a\}, \Sigma = \{a\}$
- 2)  $L = \{a^2\}, \Sigma = \{a\}$
- 3)  $L = \{a, a^2\}, \Sigma = \{a\}$
- 4)  $L = \{\epsilon, a, a^2\}, \Sigma = \{a\}$
- 5)  $L = \{a^n \mid n \geq 0\}, \Sigma = \{a\}$
- 6)  $L = \{a^n \mid n \geq 1\}, \Sigma = \{a\}$
- 7)  $L = \{a, b\}^*, \Sigma = \{a, b\}$
- 8)  $L = \{a, b\}^+, \Sigma = \{a, b\}$

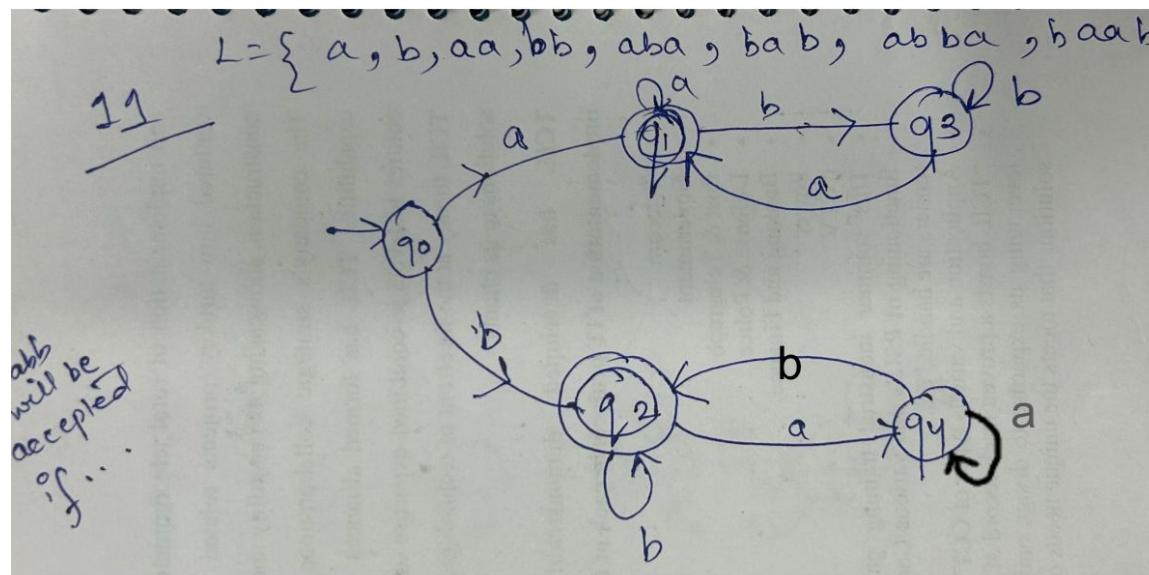
**Q9** Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , starts with 'a'



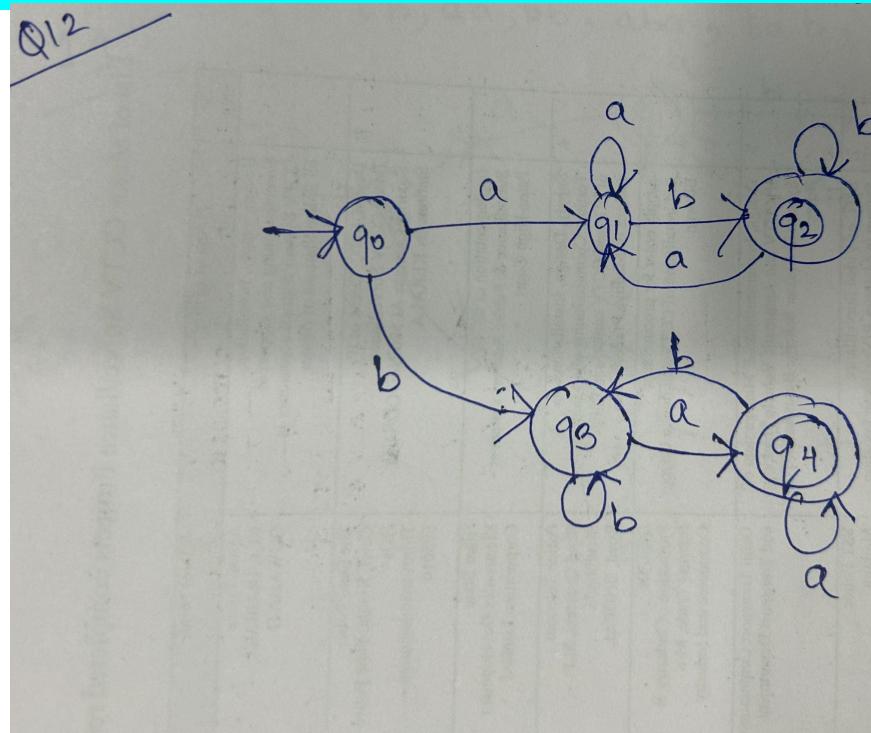
Q10 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , starts and ends with 'a'



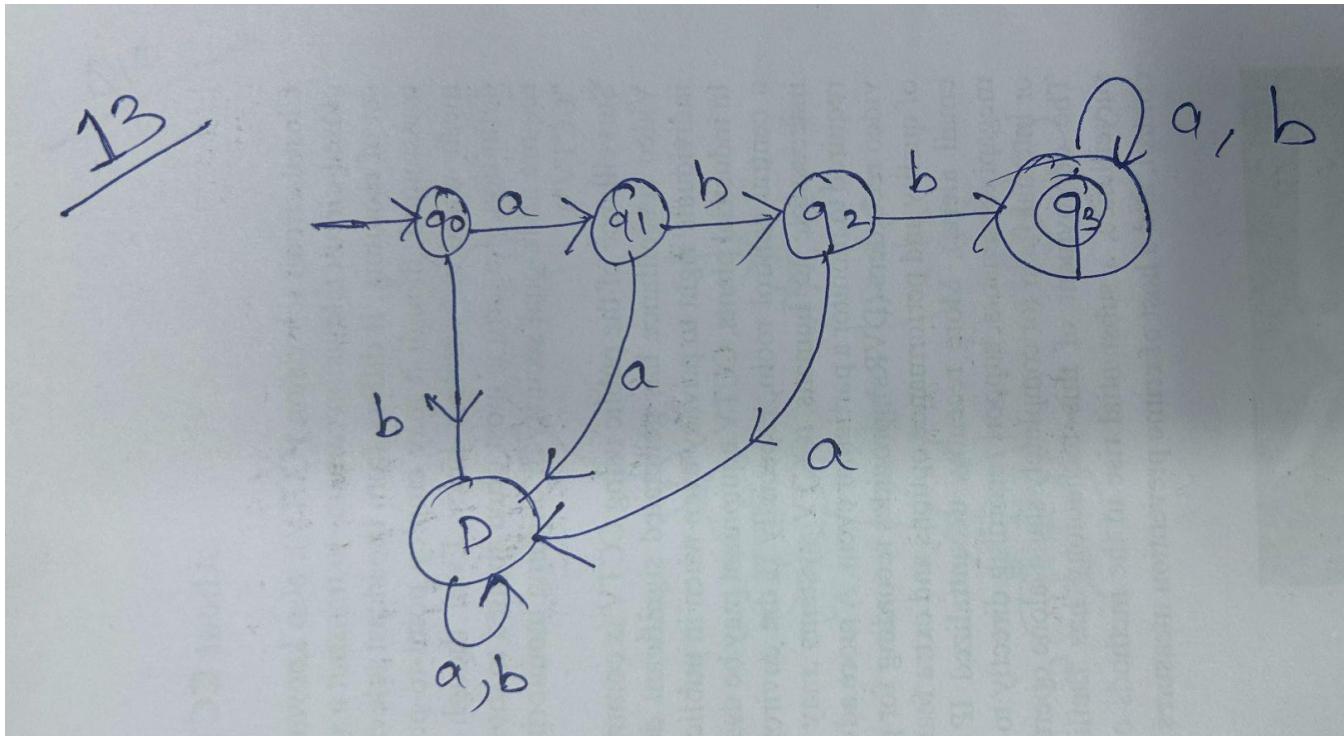
Q11 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , starts and ends with the same symbol



Q12 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , starts and ends with the different symbol

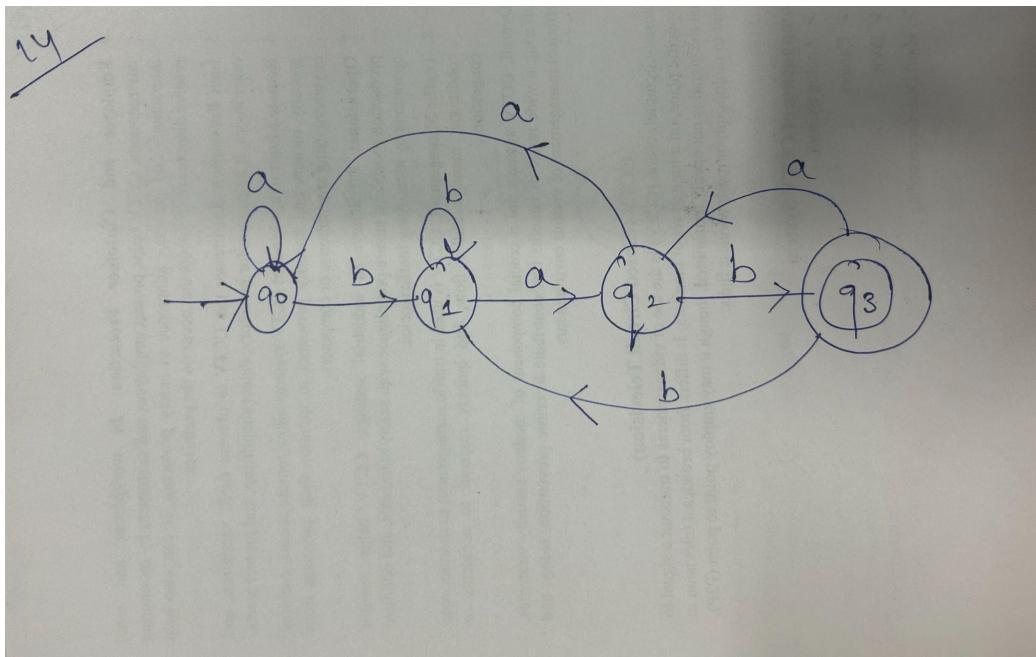


Q13 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , starts with the substring  $s='abb'$

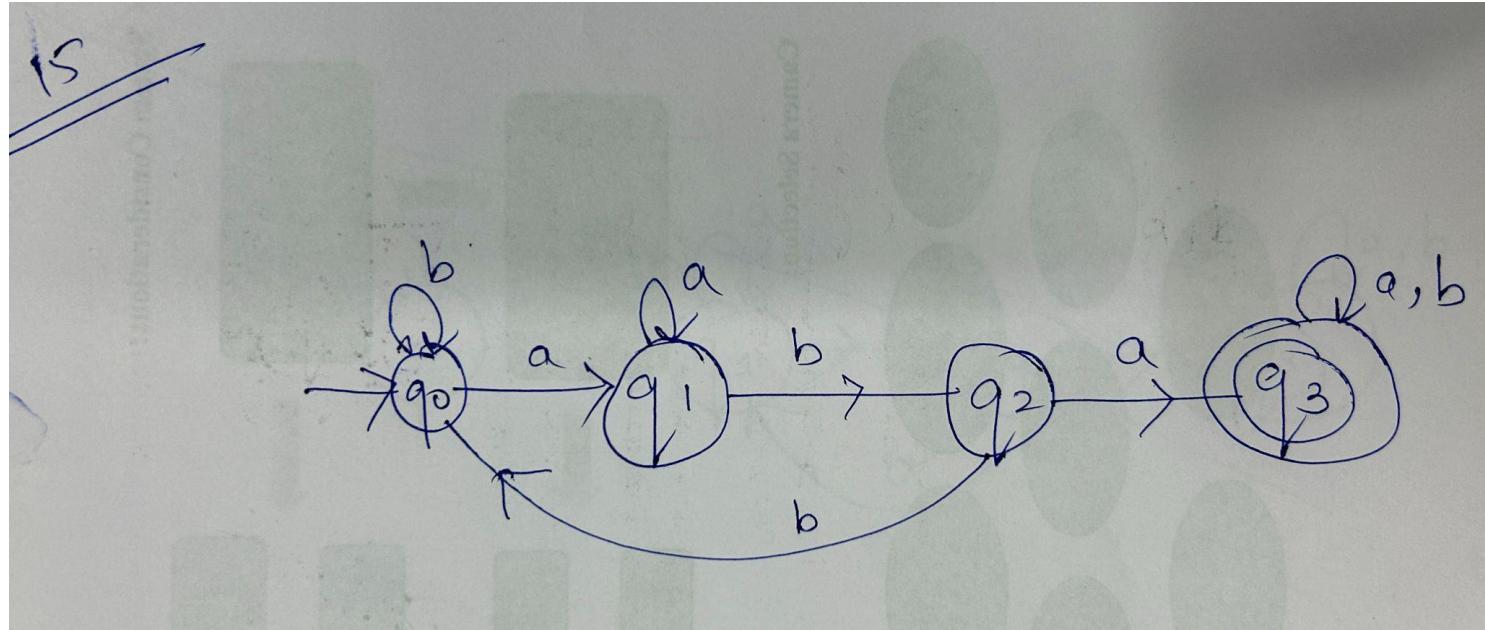


Q14 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , ends with the substring  $s='bab'$

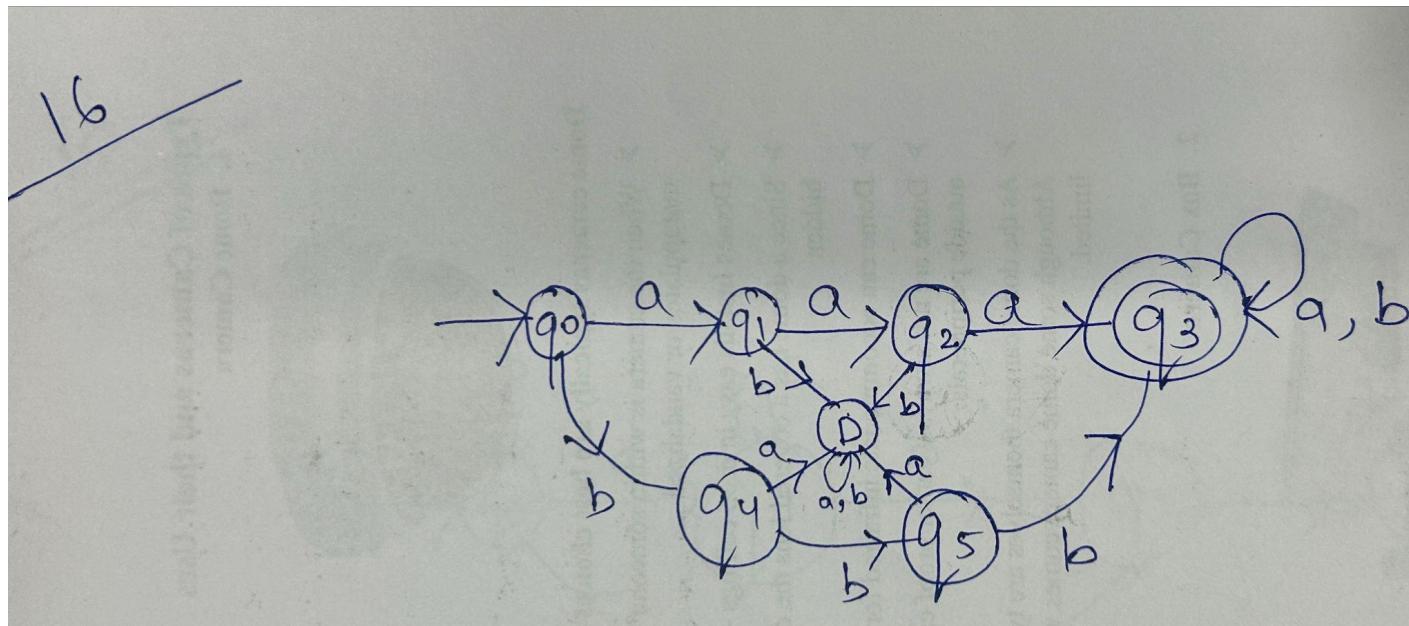
Q14 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , ends with the substring  $s='bab'$



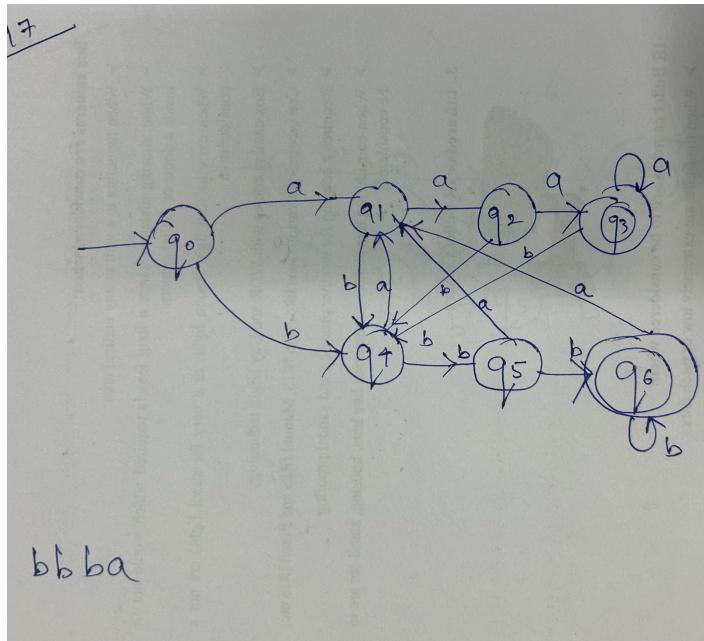
Q15 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , contains the substring  $s='aba'$



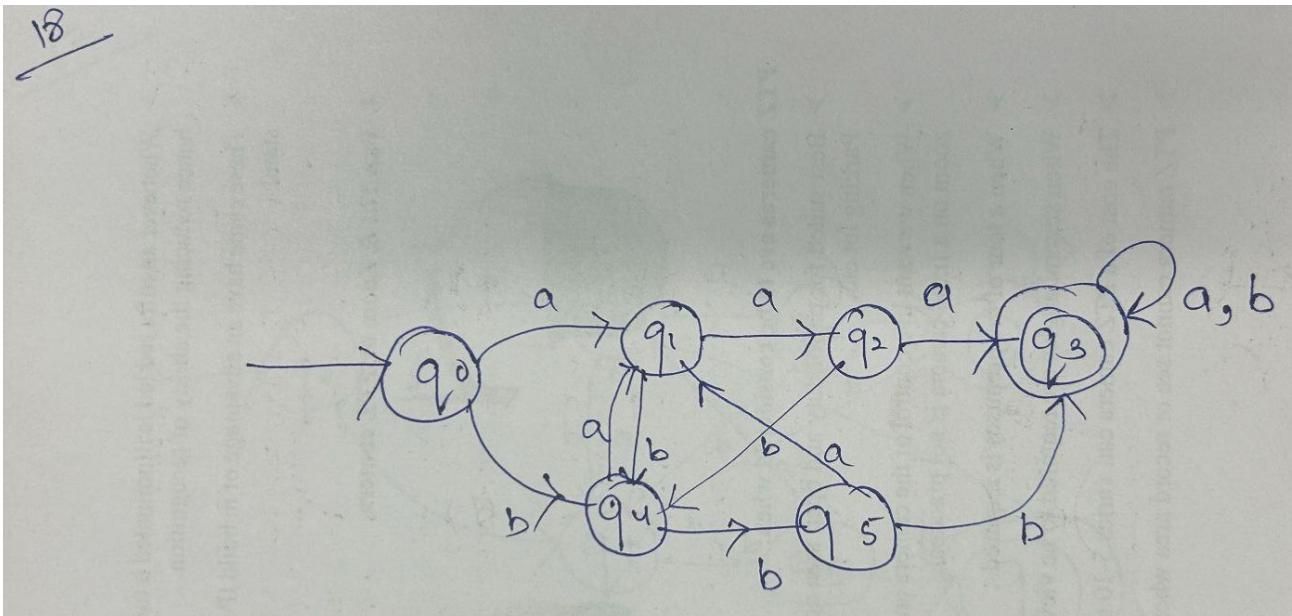
Q16 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , is like  $w=AX$ .  $A=aaa/bbb?$



Q17. Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , is like  $w=XS$ .  $S=aaa/bbb?$

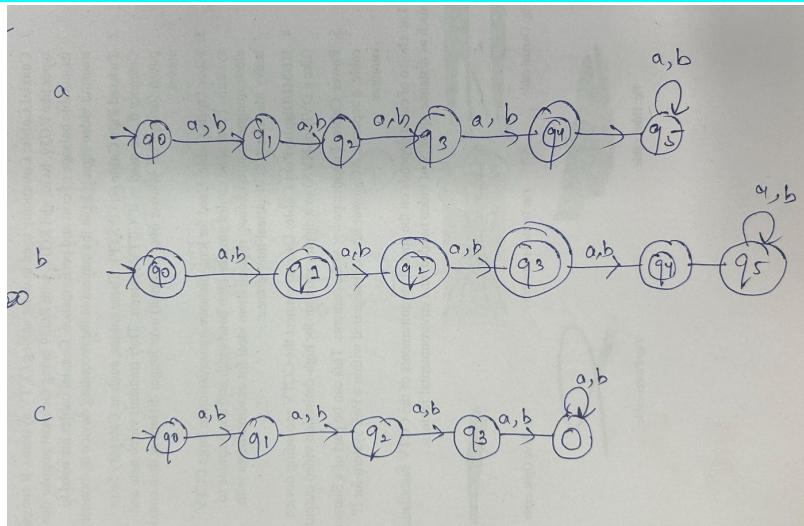


Q18. Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , is like  $w=XSX$ .  $S=aaa/bbb?$

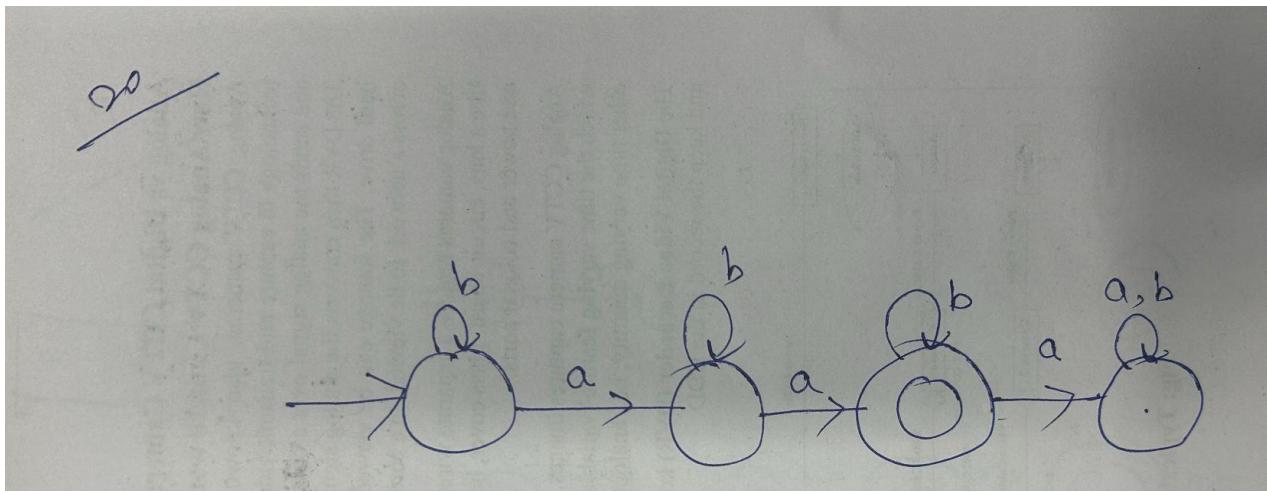


Q19 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string 'w' should be:

- a.  $|w|=4$
- b.  $|w| \leq 4$
- c.  $|w| \geq 4$

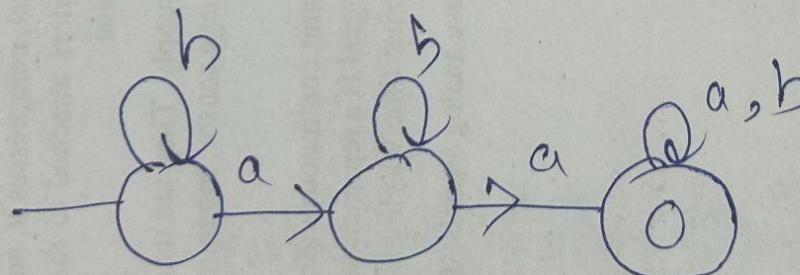


Q20 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string 'w' must consist of exactly two a's., i.e.  $|w|_a=2$

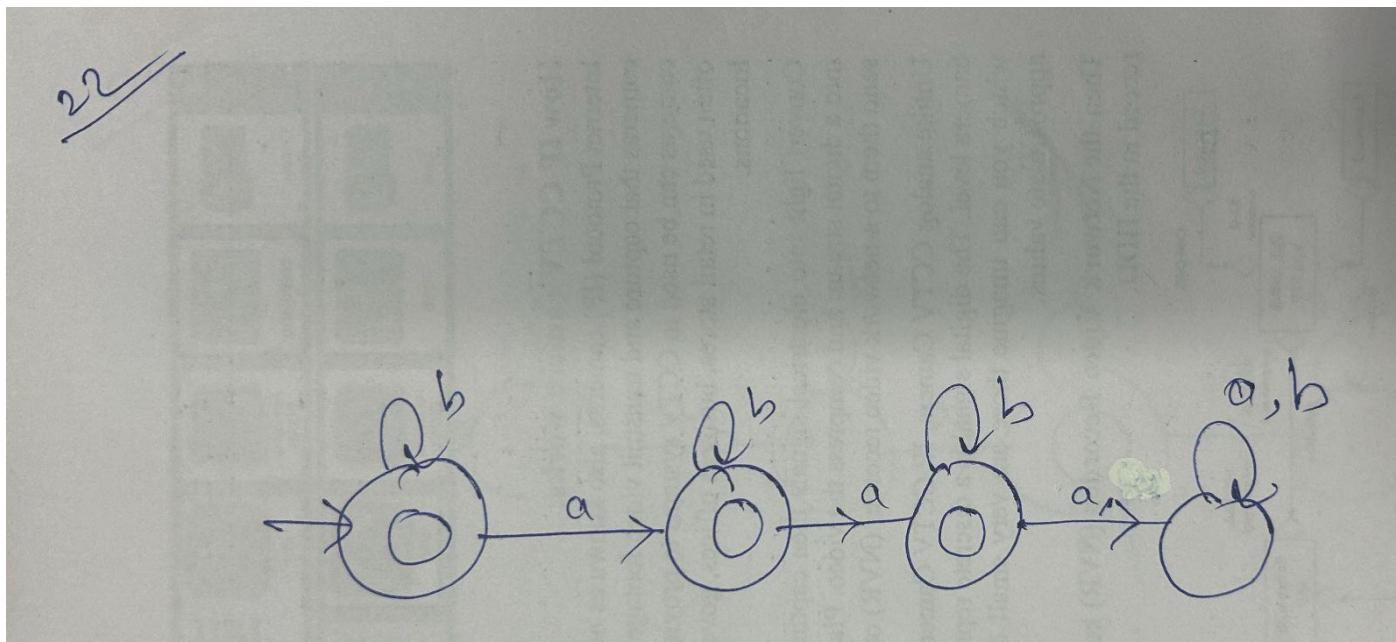


Q21 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string 'w' must consist of at least two a's., i.e.  $|w|_a >= 2$

21



Q22 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string 'w' must consist of at most two a's., i.e.  $|w|_a \leq 2$

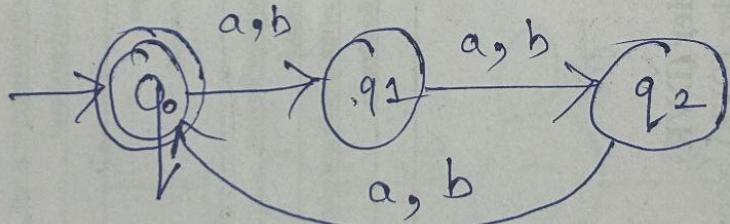


Q23 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string 'w' should be like  $|w| \equiv 0 \pmod{3}$

23

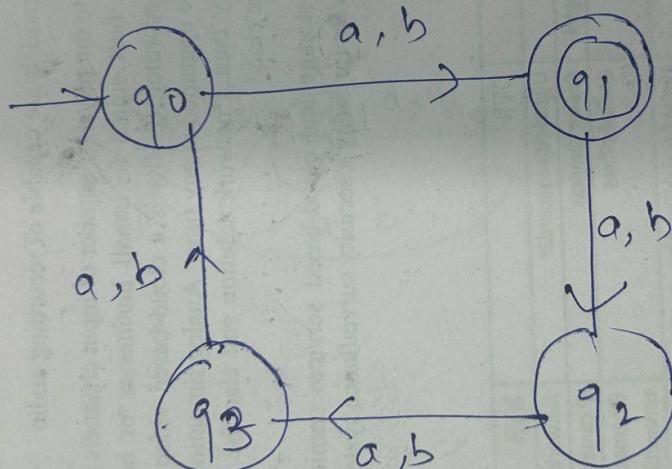
$$|w| \equiv 0 \pmod{3}$$

$\therefore L = \{ \text{all strings with length } 3, 6, 9, 12, 15, \dots \}$



Q24 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string 'w' should be like  $|w| \equiv 1 \pmod{4}$

24



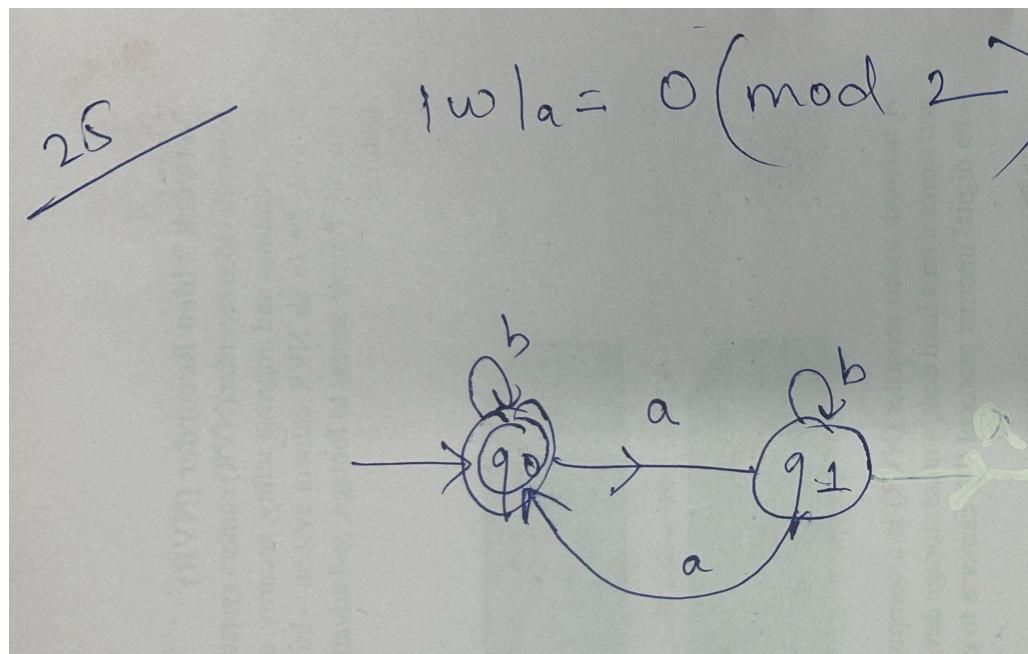
$L = \{ \text{strings } w \text{ length } \}$

$5, 9, 13, 17 \dots$   
 $\therefore \lambda = 1 \text{ when } \div 4$

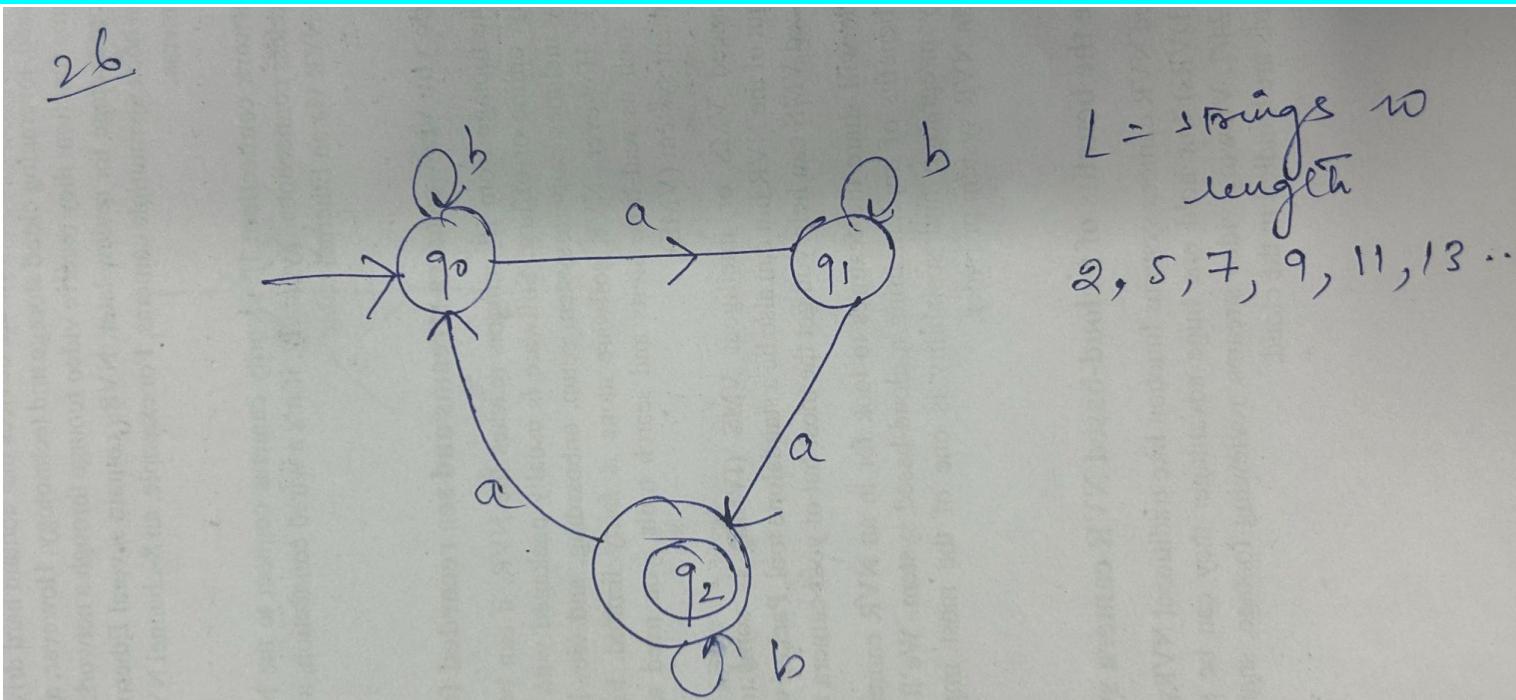
aa, abaab

Q25 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string 'w', must contain  $0(\text{mod } 2)$  number of a's, i.e.

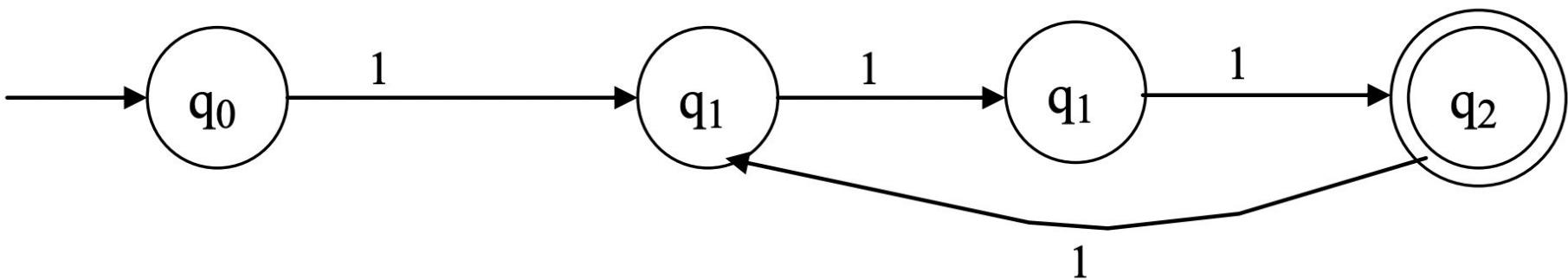
$$|w|_a = 0(\text{mod } 2)$$



Q26 Design a minimal DFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string 'w', must contain  $2(\text{mod } 3)$  number of a's, i.e.  $|w|_a = 2(\text{mod } 3)$



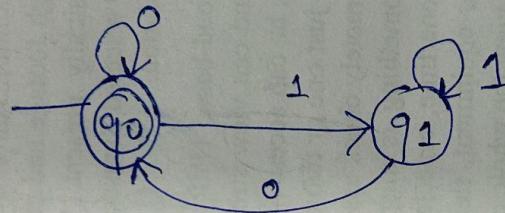
**Q27 Design a minimal DFA to check whether given unary number is divisible by three.**



**Q28 Design a minimal DFA to check whether given binary number is divisible by 2.**

28

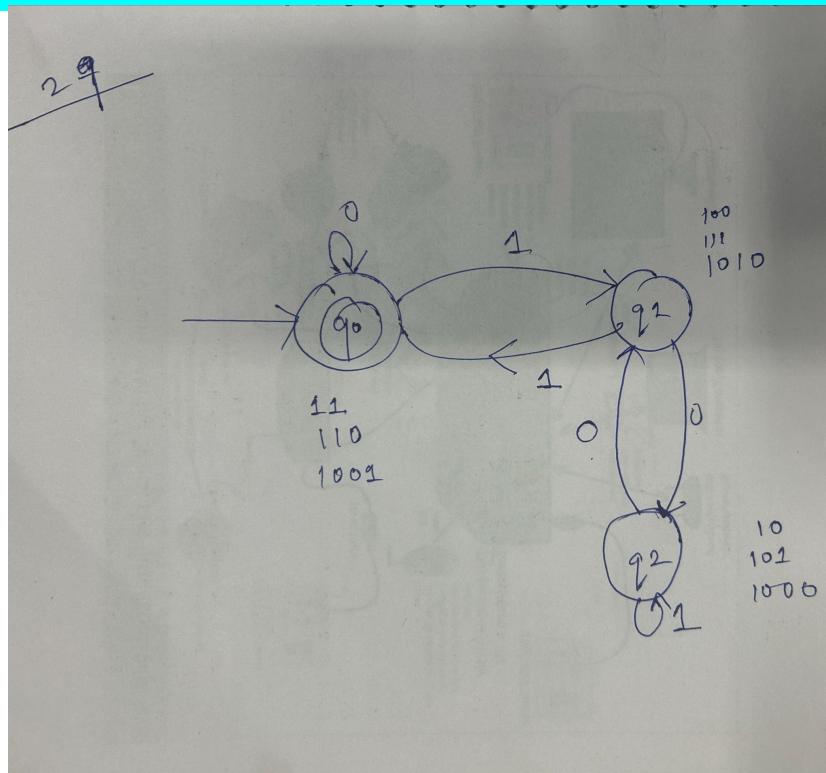
Binary #s  $\div$  by 2 ,    0, 2, 4, 6, 8, 10, ...  
                      00, 10, 100, 110, 1000, 1010.



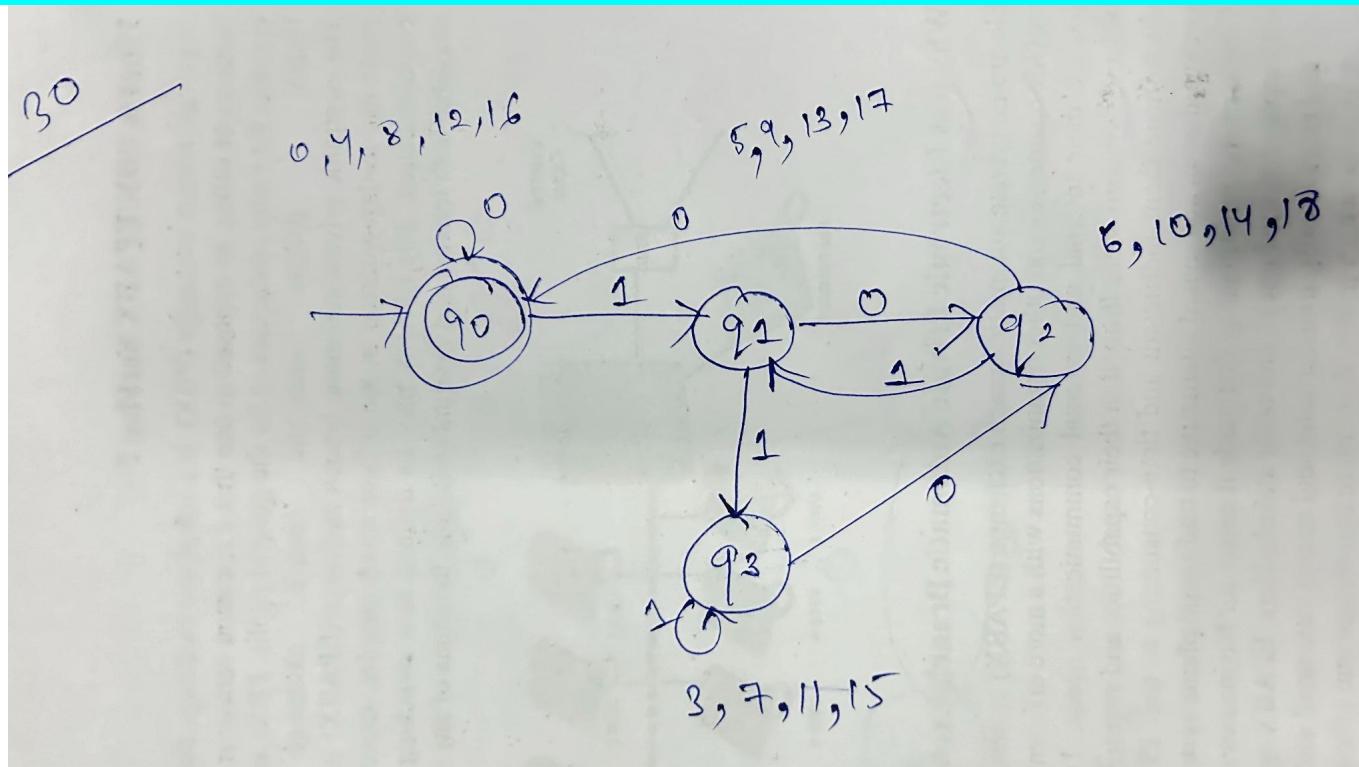
0  
10  
100  
1100

1  
11  
101  
111

**Q29 Design a minimal DFA to check whether given binary number is divisible by 3.**



Q30 Design a minimal DFA to check whether given binary number is divisible by 4.



# NFA (Non-Deterministic finite automata)

---

- A Non-Deterministic finite automaton (NDFA) is a 5-tuple  $(Q, \Sigma, \delta, S, F)$  where:

1.  $Q$ : finite set of states
2.  $\Sigma$ : finite set of the input symbol
3.  $S$ : initial state
4.  $F$ : final state
5.  $\delta$ : Transition function

$$\delta: Q \times \Sigma \rightarrow 2^Q$$

# POINTS TO REMEMBER

**Every DFA is also an NFA.**

**Accepting power of NDFA= Accepting power of DFA**

**Every NFA can be translated to an equivalent DFA, so their language accepting capability is same.**

**NFAs like DFA's only recognize regular languages.**

**It need not to be a complete system. There can be a state that doesn't have any transition on some input symbol.**

**No concept of dead state**

**It is possible that a single state led to multiple transition on same input to different states.**

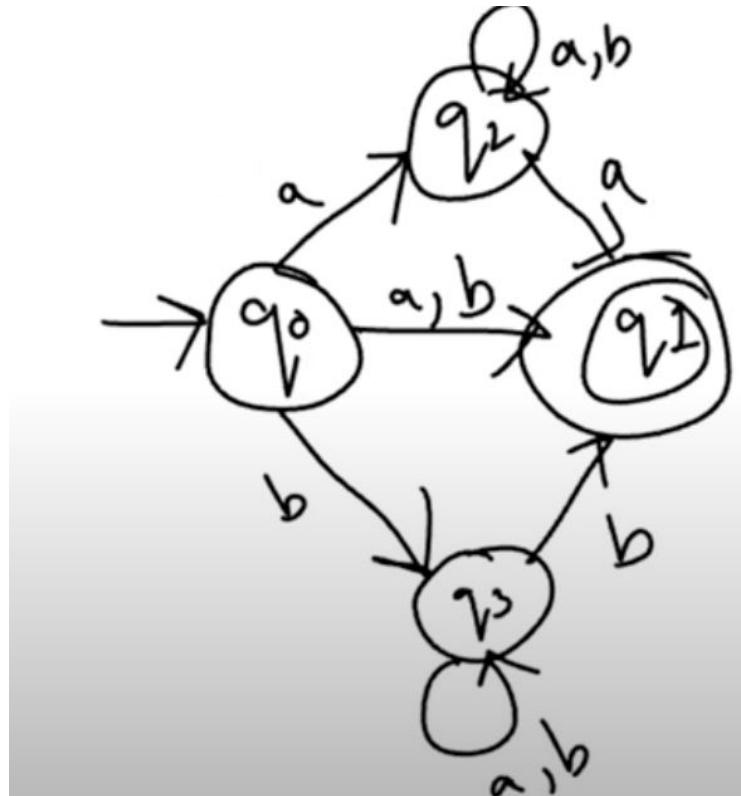
**NOTE- A null transition is also possible for NFA, such special NFA are called Null-NFA. We will**

## Acceptance by NFA

Let 'w' be any string defined over the alphabet S, corresponding to w, there can be multiple transitions for NFA starting from initial state, if there exist at least one transition for which we start at the initial state and ends in any one of the final state, then the string 'w' is said to be accepted by the non-deterministic finite automata, otherwise not.

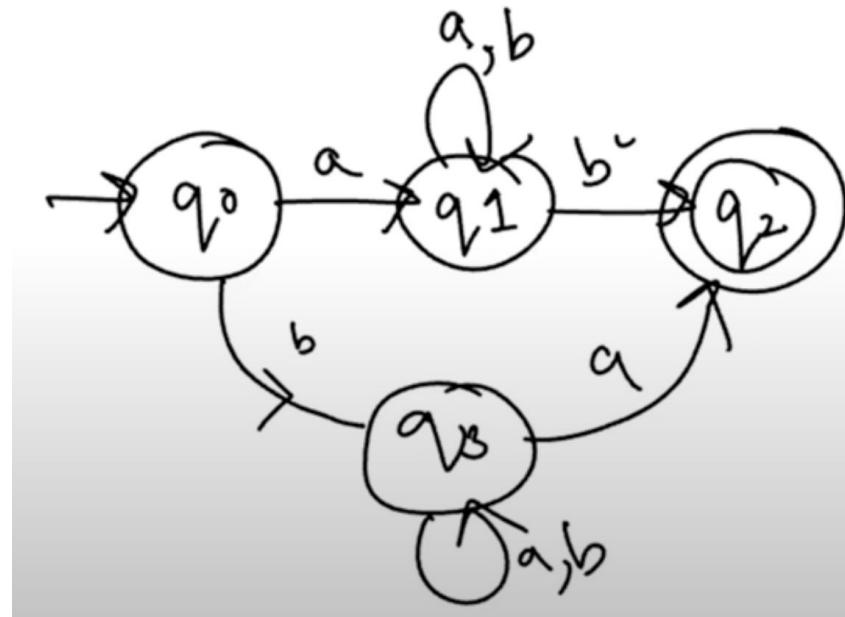
Q1. Design a NFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string start and end with same symbol

Q1. Design a NFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string start and end with same symbol



Q2. Design a NFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string start and end with different symbol

Q2. Design a NFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string start and end with different symbol



Q3. Design a minimal NFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$ . where every accepted string 'w' contains substring s, Where s = 'aba'

Q4. Design a NFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$ . where every accepted string 'w' ends with substring 's', Where s = 'bab'?

Q5. Design a NFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$ , where every accepted string 'w' starts with substring s, Where s = 'aba'

Q6. Design a NDFA that accepts all strings over the alphabet  $\Sigma = \{a, b\}$  such that for every accepted string 3rd from right end is always a ?

# DFA Vs NFA

Aspect	DFA (Deterministic Finite Automata)	NDFA (Nondeterministic Finite Automata)
State Transition	On each input symbol, transitions to exactly one state.	Can transition to multiple states or none on the same input symbol.
Determinism and Uniqueness	Each state has a unique transition for each input symbol.	A state can have multiple transitions for the same input symbol.
Computation Path	Always has a single, unique computation path for any input string.	May have multiple computation paths for the same input string.
Ease of Construction	Generally simpler and more straightforward to construct.	Can be more complex to construct due to non-determinism.
Acceptance of Input	Accepts an input if it reaches a final state after processing all input symbols	Accepts an input if at least one computation path reaches a final state

## Conversion from NFA to DFA(NFA and DFA Equivalence)

- 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.
- Since every NFA and DFA has equal power that means, for every language if a NFA is possible, then DFA is also possible. So, every NFA can be converted to DFA
- Let,  $M = (Q, \Sigma, \delta, q_0, F)$  is an NFA which accepts the language  $L(M)$ . There should be equivalent DFA denoted by  $M' = (Q', \Sigma', q_0', \delta', F')$  such that  $L(M) = L(M')$ .

## Steps for converting NFA to DFA:

Step 1: Convert the given NFA to its equivalent transition table

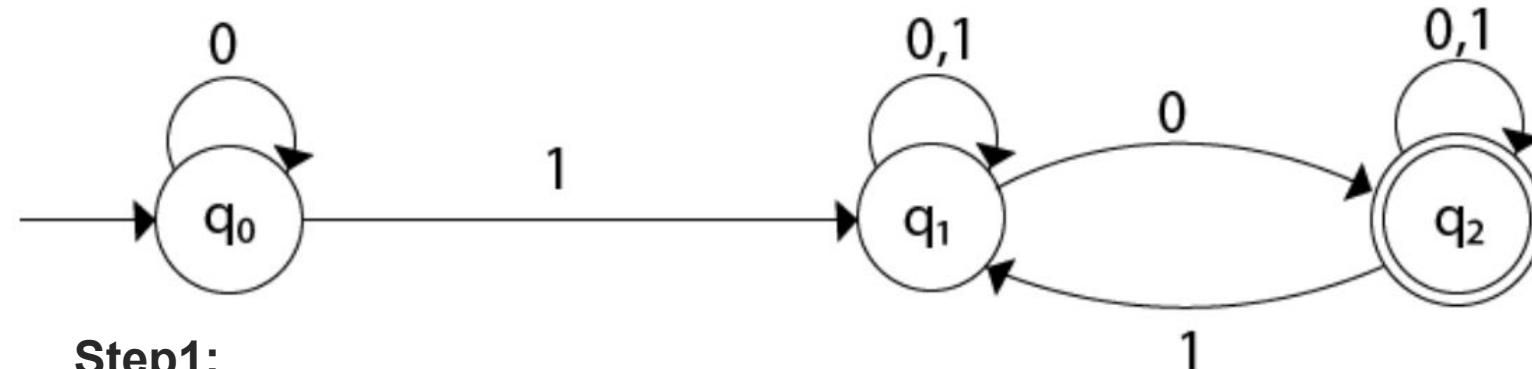
Step 2: Create the DFA's start state. Initial state will always remain same.

Step 3: Create the DFA's transition table

Step 4: Create the DFA's final states  
The DFA's final states are the sets of states that contain at least one final state from the NFA.

Step 5: Simplify the DFA

Convert the given NFA to DFA.

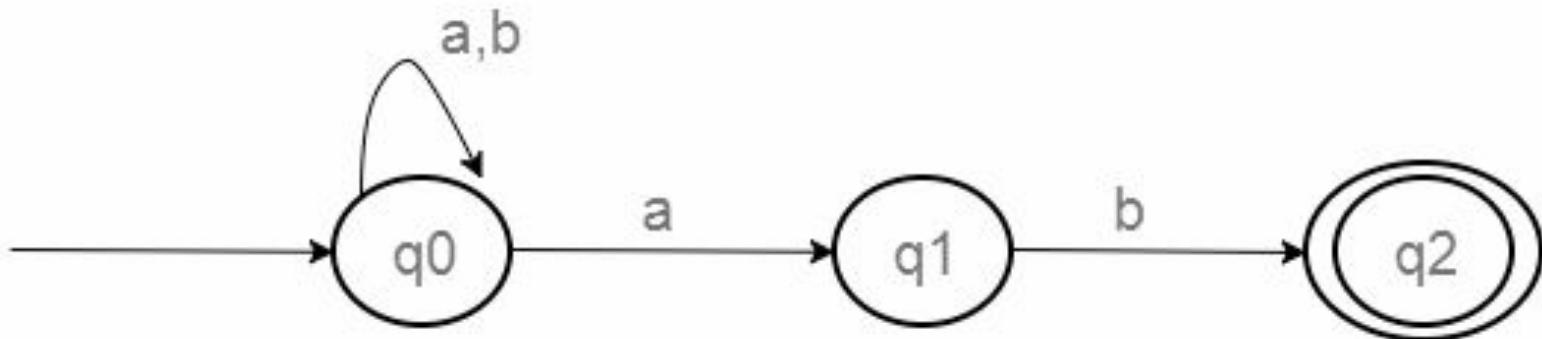


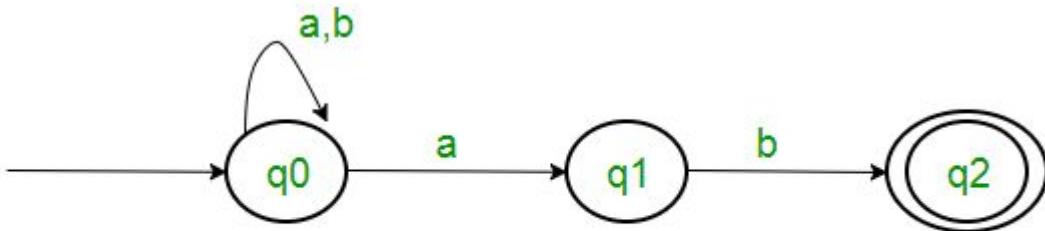
**Step1:**

State	Input '0'	Input '1'
$q_0$	$\{q_0\}$	$\{q_1\}$
$q_1$	$\{q_1, q_2\}$	$\{q_1\}$
$q_2$	$\{q_2\}$	$\{q_1, q_2\}$

Step 2:

DFA State	Input '0'	Input '1'
$q_0$	$q_0$	$q_1$
$q_1$	$q_1q_2$	$q_1$
$q_1q_2$	$q_1q_2$	$q_1q_2$

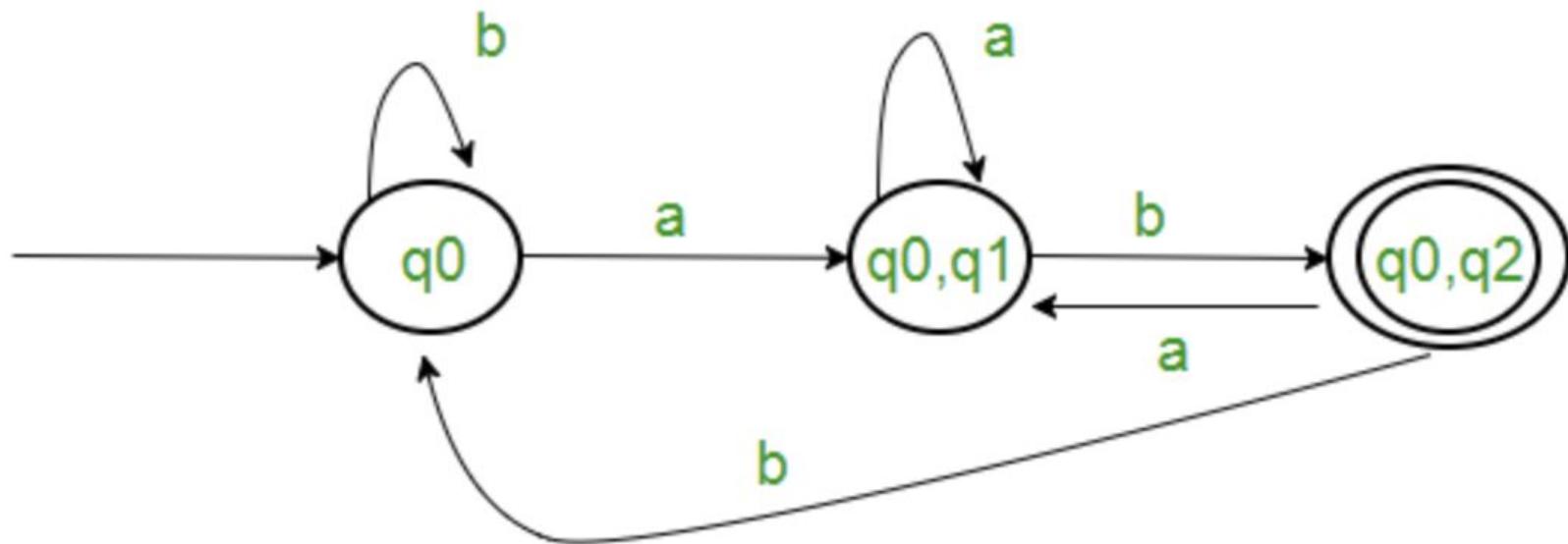


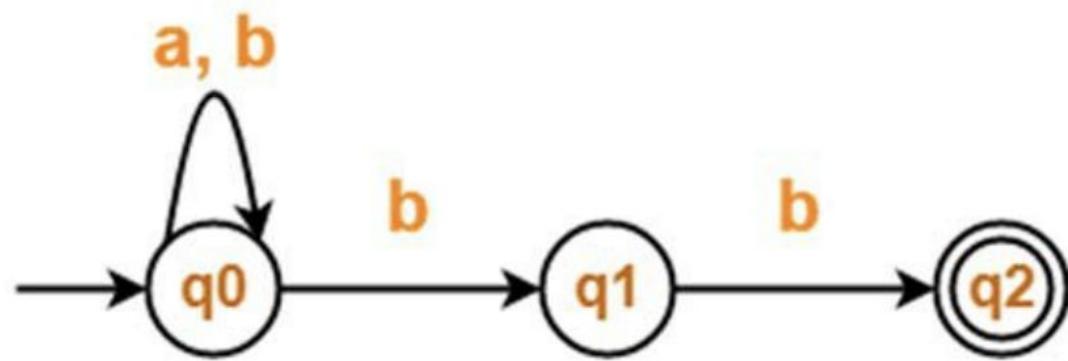


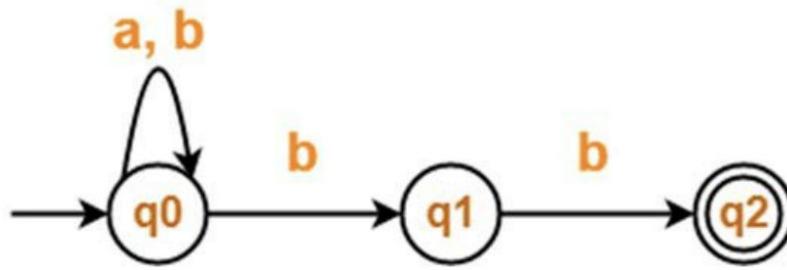
State	Input 'a'	Input 'b'
$q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$q_1$	$\emptyset$	$\{q_2\}$
$q_2$	$\emptyset$	$\emptyset$

State	Input 'a'	Input 'b'
$q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$q_1$	$\emptyset (\phi)$	$\{q_2\}$
$q_2$	$\emptyset (\phi)$	$\emptyset (\phi)$

DFA State	Input 'a'	Input 'b'
$\rightarrow q_0$	$q_0q_1$	$q_0$
$q_0q_1$	$q_0q_1$	$q_0q_2$
$q_0q_2$	$q_0q_1$	$q_0$



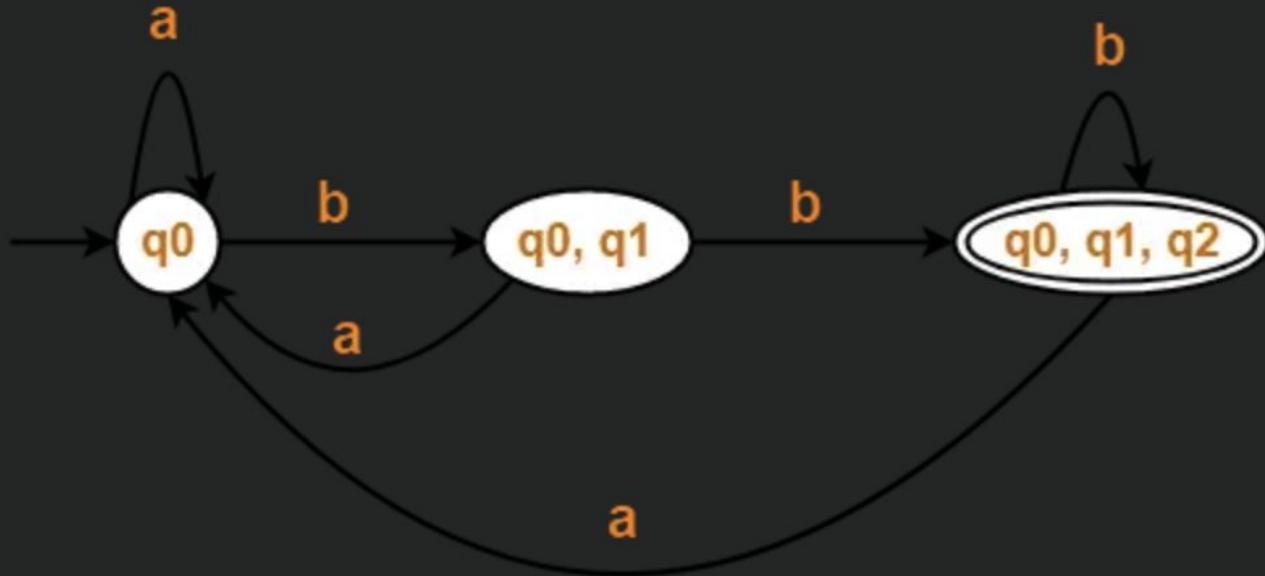




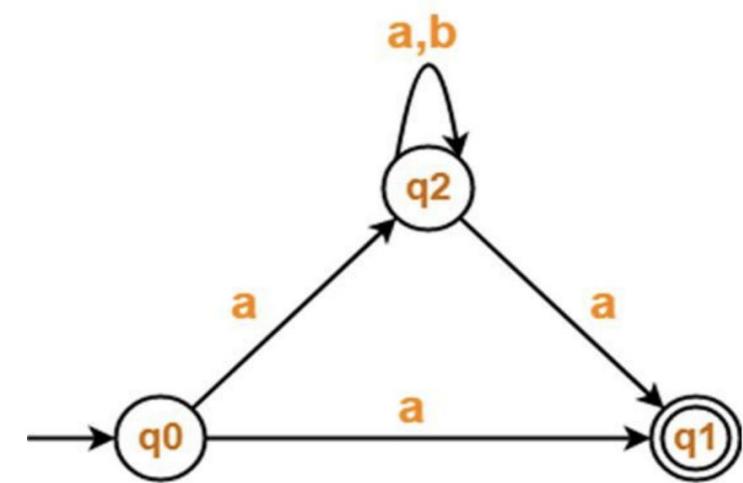
State / Alphabet	a	b
$\rightarrow q_0$	$q_0$	$q_0, q_1$
$q_1$	—	$*q_2$
$*q_2$	—	—

<b>State / Alphabet</b>	<b>a</b>	<b>b</b>
$\rightarrow q_0$	q0	q0, q1
q1	—	*q2
*q2	—	—

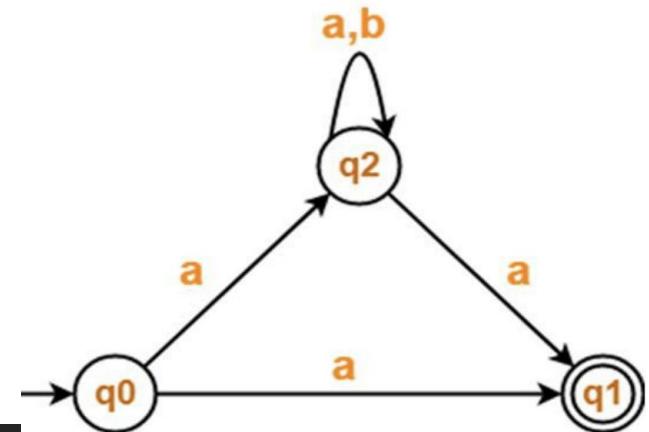
<b>State / Alphabet</b>	<b>a</b>	<b>b</b>
$\rightarrow q_0$	q0	{q0, q1}
{q0, q1}	q0	*{q0, q1, q2}
*{q0, q1, q2}	q0	*{q0, q1, q2}



Deterministic Finite Automata (DFA)



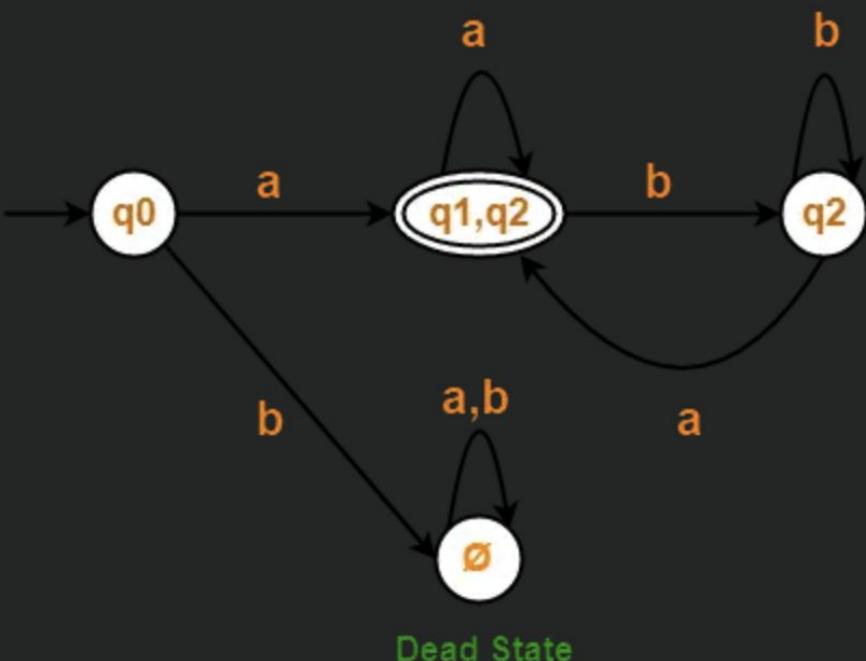
# Step 1



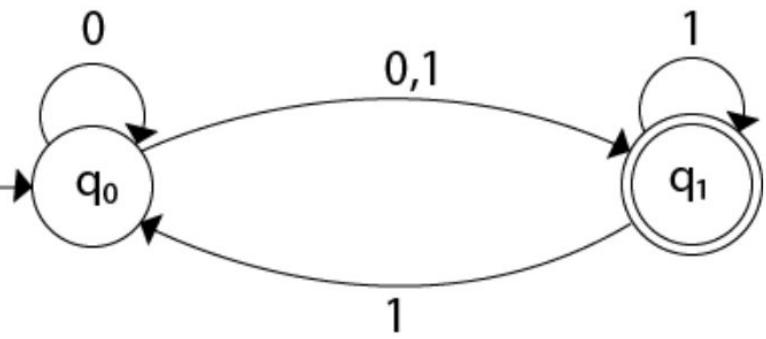
State / Alphabet	a	b
$\rightarrow q_0$	$*q_1, q_2$	-
$*q_1$	-	-
$q_2$	$*q_1, q_2$	$q_2$

<b>State / Alphabet</b>	<b>a</b>	<b>b</b>
$\rightarrow q_0$	$*q_1, q_2$	-
$*q_1$	-	-
$q_2$	$*q_1, q_2$	$q_2$

<b>State / Alphabet</b>	<b>a</b>	<b>b</b>
$\rightarrow q_0$	$\{*q_1, q_2\}$	$\emptyset$
$\{*q_1, q_2\}$	$\{*q_1, q_2\}$	$q_2$
$q_2$	$\{*q_1, q_2\}$	$q_2$
$\emptyset$	$\emptyset$	$\emptyset$



Deterministic Finite Automata (DFA)



## Points to Remember

- After conversion, the number of states in the resulting DFA may or may not be same as NFA.
- The maximum no. of states that may be present in the DFA are  $2^{\text{Number of states in the NFA}}$ .

In general, the following relationship exists between the number of states in the NFA and DFA

---

$$1 \leq n \leq 2^m$$

---

where m = Number of states in the NFA

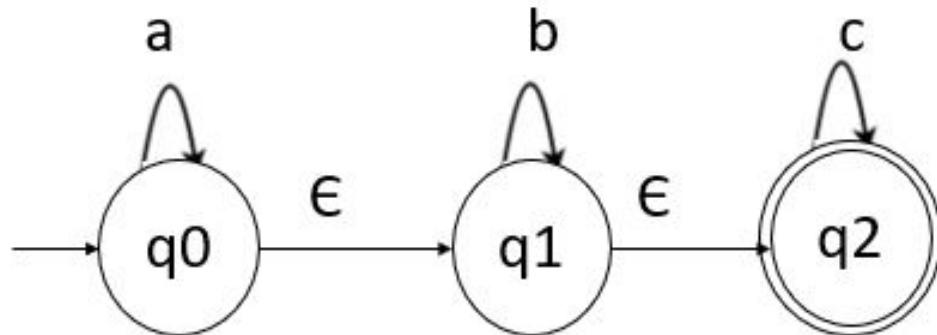
n = Number of states in the DFA

# NFA WITH EPSILON MOVES ( $\epsilon$ -NFA)

- An automaton that consists of null transitions is called a Null- NFA i.e. we allow a transition on null means empty string.
- $\epsilon$ -NFA is a 5-tuple  $(Q, \Sigma, \delta, S, F)$  where:
  - $Q$  is a finite and non-empty set of states
  - $\Sigma$  is a finite non-empty set of finite input alphabet
  - $\delta$  is a transition function  $\delta: (Q \times (\Sigma \cup \{\epsilon\})) \rightarrow 2^Q$
  - $S$  is initial state (always one) ( $S \in Q$ )
  - $F$  is a set of final states ( $F \subseteq Q$ ) ( $0 \leq |F| \leq n$ , where  $n$  is the number of states)

# NULL/ $\epsilon$ Closure

The  $\epsilon$  closure( $P$ ) is a set of states which are reachable from state  $P$  along  $\epsilon$  labelled transition path.



The Null closure  $Q$  is always a non-empty and finite state, because every state's null closure is that state only.

$$\epsilon\text{-closure}(\Phi) = \Phi$$

## EQUIVALENCE BETWEEN NULL NFA TO NFA

- There will be no change in the initial state.
- No change in the total no. of states
- May be change in the number of final states.
- All the states whose  $\epsilon$ -closure consists of at least one final state in the initial  $\epsilon$ -NFA will get the status of the final state in the resulting NFA

# How to convert Epsilon NFA to NFA

Step 1: Find null closure of all states

Step 2: Follow the process of null closure, input, null closure

$$\square(q_0, a) = \varepsilon\text{-Closure}[\square[\varepsilon\text{-Closure}(q_0), a]]$$

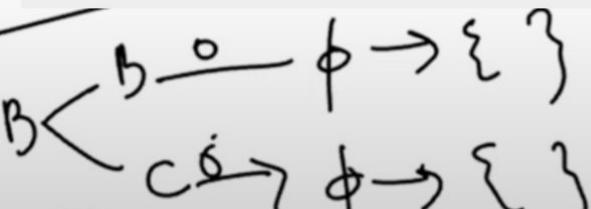
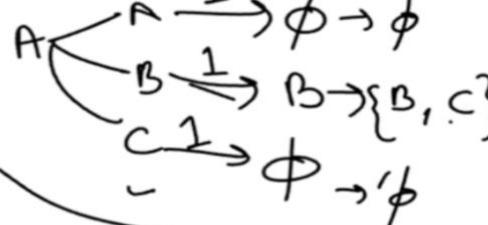
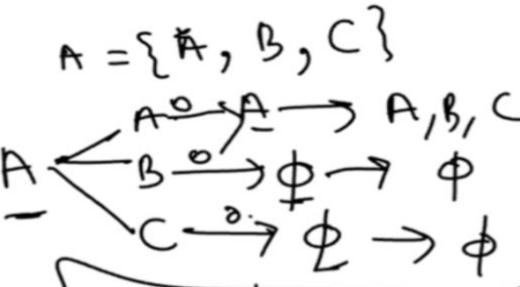
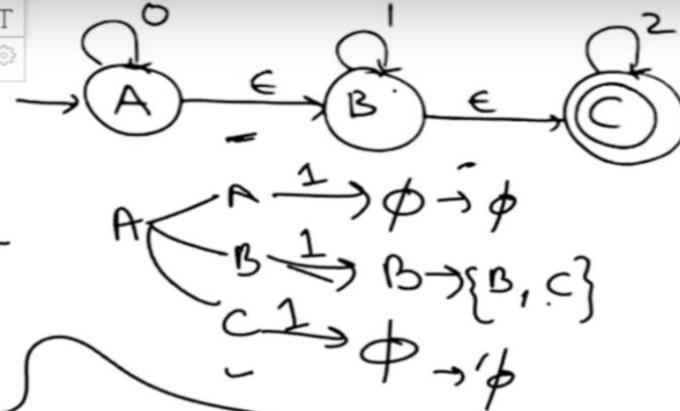
Step 3: Resulting NFA

NF

	0	1	2
A	$\{B, C\}$	$\{B, C\}$	C
B	$\{\epsilon\}$	$\{B, C\}$	C
C	$\{\epsilon\}$	$\{\epsilon\}$	C
	$\epsilon \xrightarrow{2} \phi \rightarrow \phi$		
A	$\epsilon \xrightarrow{2} \phi \rightarrow \phi$		
C	$\epsilon \xrightarrow{2} C \rightarrow C$		



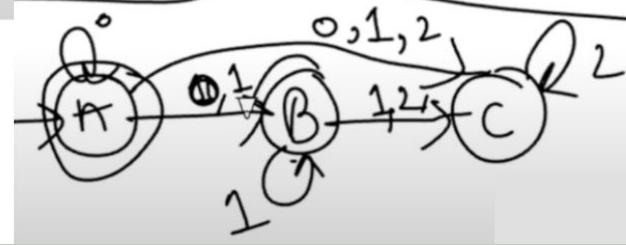
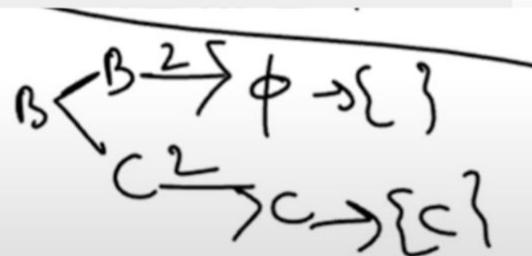
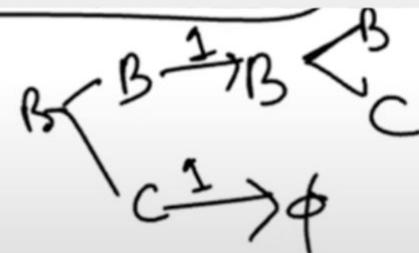
$\Sigma\text{-NFA} \rightarrow \text{NFA}$

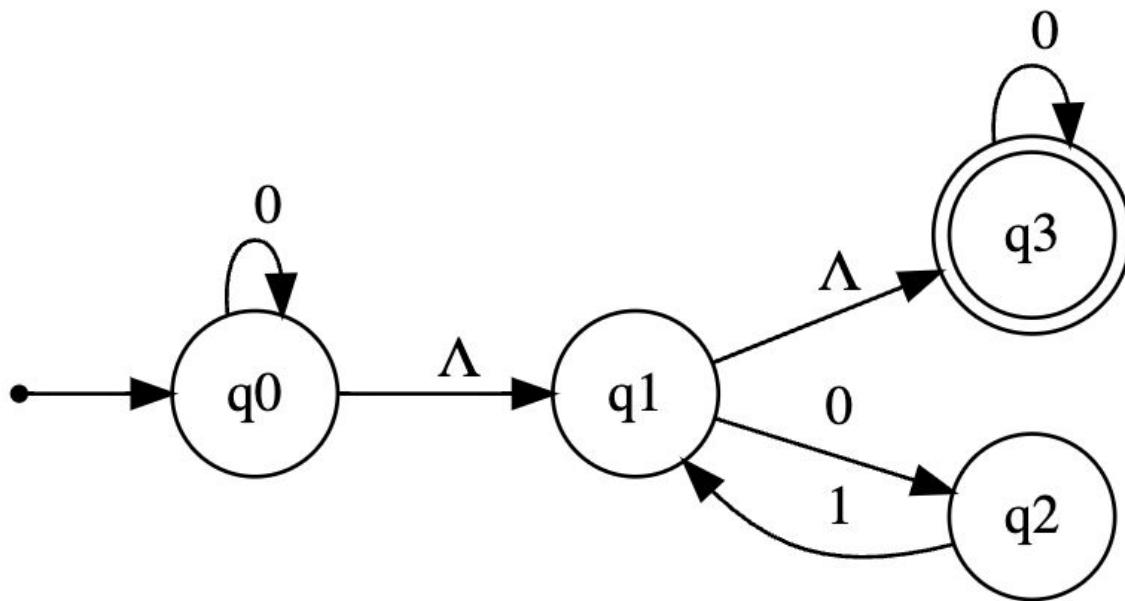


$C \xleftarrow{C \circ C} \phi \rightarrow \{\}$

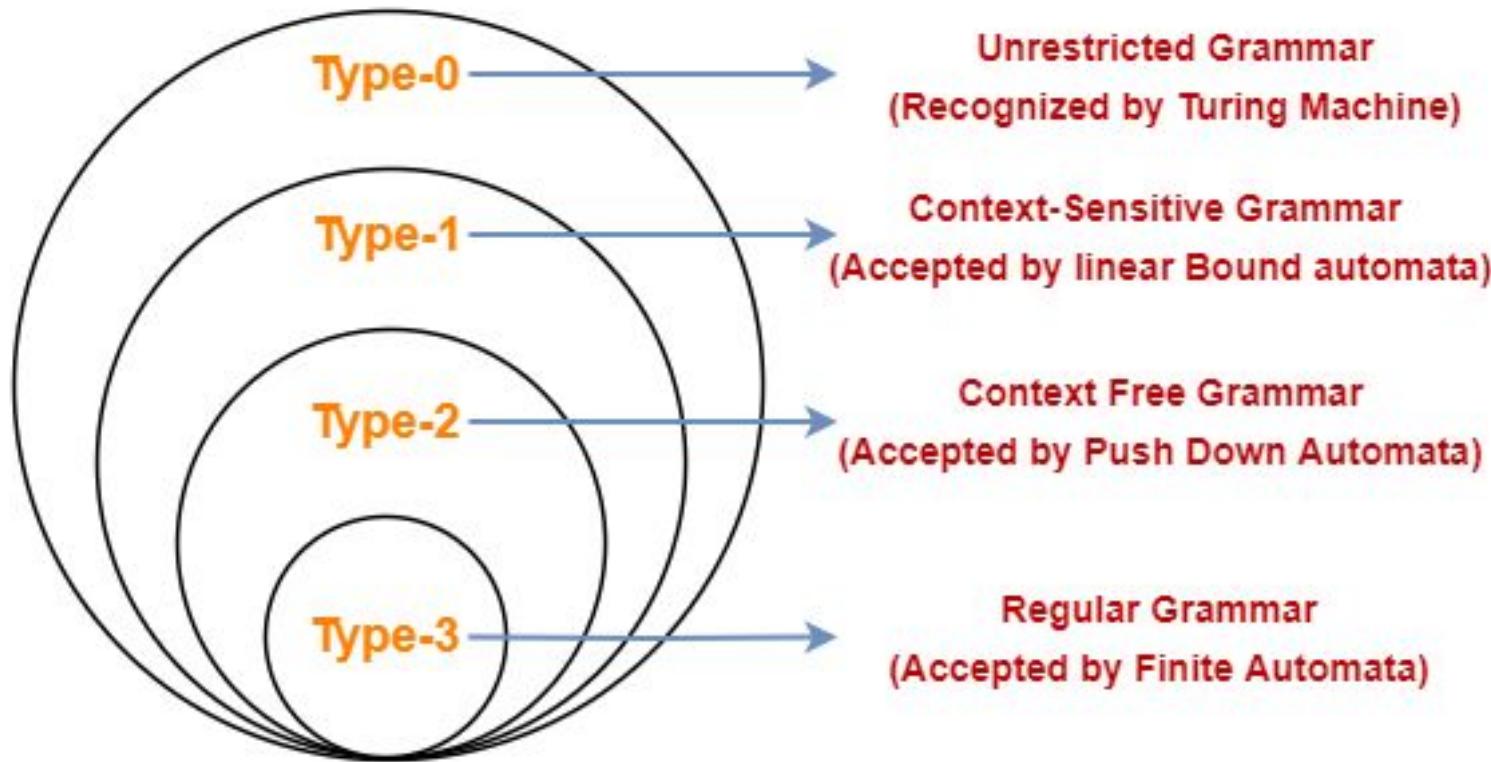
$C \xleftarrow{C \circ C} \phi \rightarrow \{\}$

$C \xleftarrow{C \circ C} C \rightarrow \{C\}$





# Chomsky hierarchy



## Type 0: Unrestricted Grammars

Language recognized by Turing Machine is known as Type 0 Grammar. They are also known as Recursively Enumerable Languages.

Grammar Production for Type 0 is given by

$\alpha \rightarrow \beta$

where  $\alpha$  and  $\beta$  are strings of non-terminal and terminal symbols, with  $\alpha$  containing at least one non-terminal.

## Type 1: Context-Sensitive Grammars

Languages recognized by Linear Bounded Automata are known as **Type 1 Grammar**. Context-sensitive grammar represents context-sensitive languages.

$$\alpha A \beta \rightarrow \gamma$$

where  $\alpha$ ,  $\beta$ , and  $\gamma$  are strings of terminals and non-terminals, and  $A$  is a non-terminal. The length of the right-hand side (RHS) must be at least as long as the left-hand side (LHS):

$$|\alpha A \beta| \leq |\gamma|$$

**Special Case for Start Symbol**  $S \rightarrow \epsilon$

The only exception to this rule is when the start symbol  $S$  produces the empty string directly: This special case is allowed to ensure that the grammar can generate the empty string as part of the language. However, this exception applies strictly to the start symbol and not to other non-terminals.

## Type 2: Context-Free Grammar

Languages recognized by Pushdown Automata are known as **Type 2 Grammar**.

Context-free grammar represents context-free languages.

For grammar to be context-free, it must be context-sensitive. Grammar Production for Type 2 is given by

$A \rightarrow \gamma$ , where A is a single non-terminal and  $\gamma$  is a string of terminals and non-terminals. The left-hand side of the rule is always a single non-terminal.

## Type 1: Regular Grammars

Languages recognized by Finite Automata are known as **Type 3 Grammar**.

Regular grammar represents regular languages. Rules are of the form  $A \rightarrow aB$  or  $A \rightarrow a$ , where A and B are non-terminals and a is a terminal. Regular grammars can be further divided into right-linear and left-linear grammars based on whether the non-terminal appears on the right or left of the terminal in the production rules.

# Summary

## Restrictions on Production Rules:

- **Type 0 (Unrestricted):** No restrictions.
- **Type 1 (Context-Sensitive):** The length of the left-hand side string cannot be greater than the length of the right-hand side string.
- **Type 2 (Context-Free):** The left-hand side must consist of a single non-terminal.
- **Type 3 (Regular):** The right-hand side must consist of a single terminal followed optionally by a single non-terminal (right-linear), or a single non-terminal followed by a single terminal (left-linear).

# **Minimization of DFA**

Minimization of DFA means reducing the number of states (whose presence or absence doesn't affect the language accepting capability of the given FA.)

NOTE: A minimal FA is always unique for a language

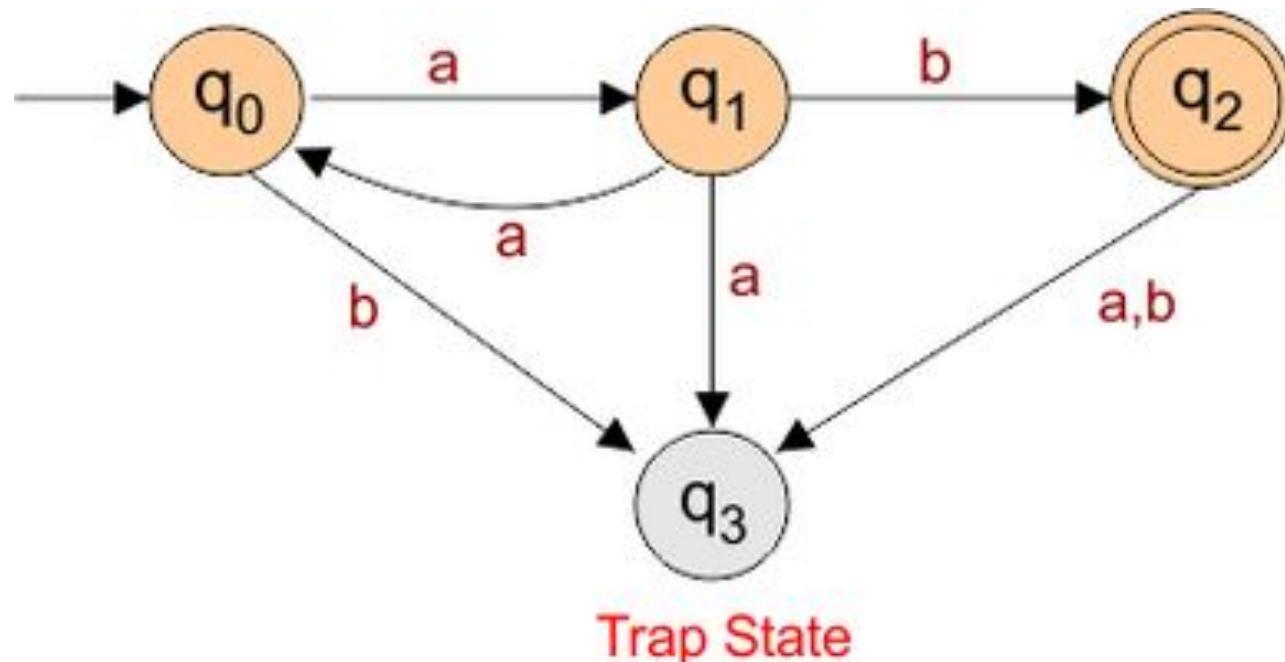
## **ELIMINATION OF NON- PRODUCTIVE STATES-**

These states don't add anything to the language accepting power to the machine. They can further be divided into

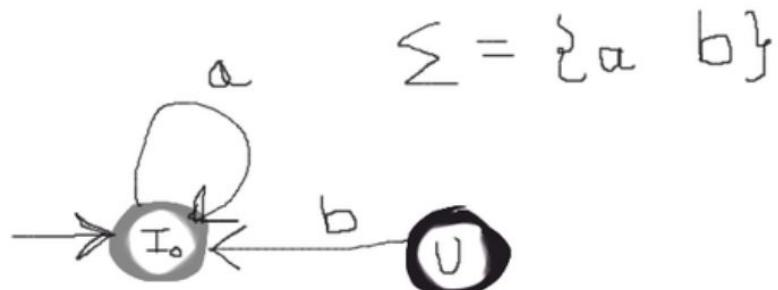
- Unreachable State • Dead State • Equal State

## NON- PRODUCTIVE STATES

**Dead State**- It is basically created to make the system complete, can be defined as a state from which there is no transition possible to the final state

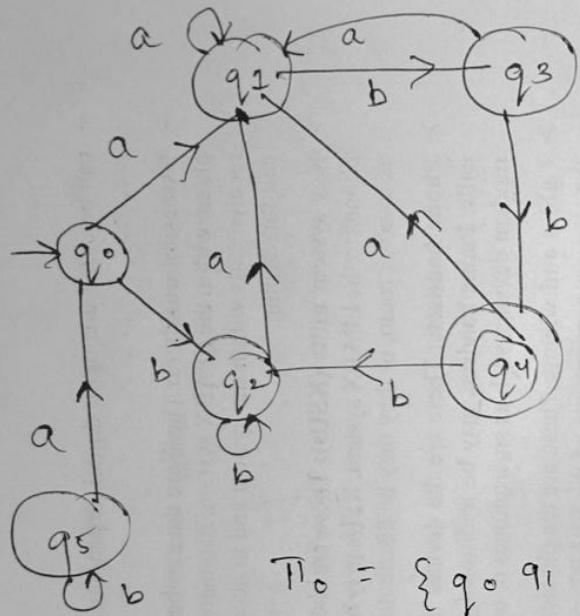


**Unreachable State-** It is that state which cannot be reached starting from initial state by parsing any input string



**Equivalent states-** Two states  $q_1$  and  $q_2$  are equivalent (denoted by  $q_1 \equiv q_2$ ) if both  $\delta(q_1, x)$  and  $\delta(q_2, x)$  are final states or both of them are non-final states for all  $x \in \Sigma^*$ . If  $q_1$  and  $q_2$  are  $k$ -equivalent for all  $k \geq 0$ , then they are  $k$ -equivalent.

These states behave in same manner on each and every input string. That is for any string  $w$  where  $w \in S^*$  either both of the states will go to final state or both will go to non-final state



$$\pi_0 = \{q_0, q_1, q_2, q_3\} \quad \{q_4\}$$

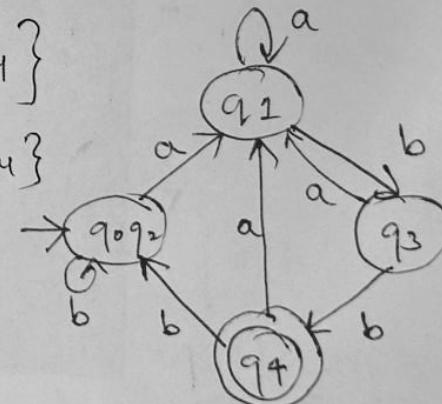
$$\pi_1 = \{q_0, q_1, q_2\} \quad \{q_3\} \quad \{q_4\}$$

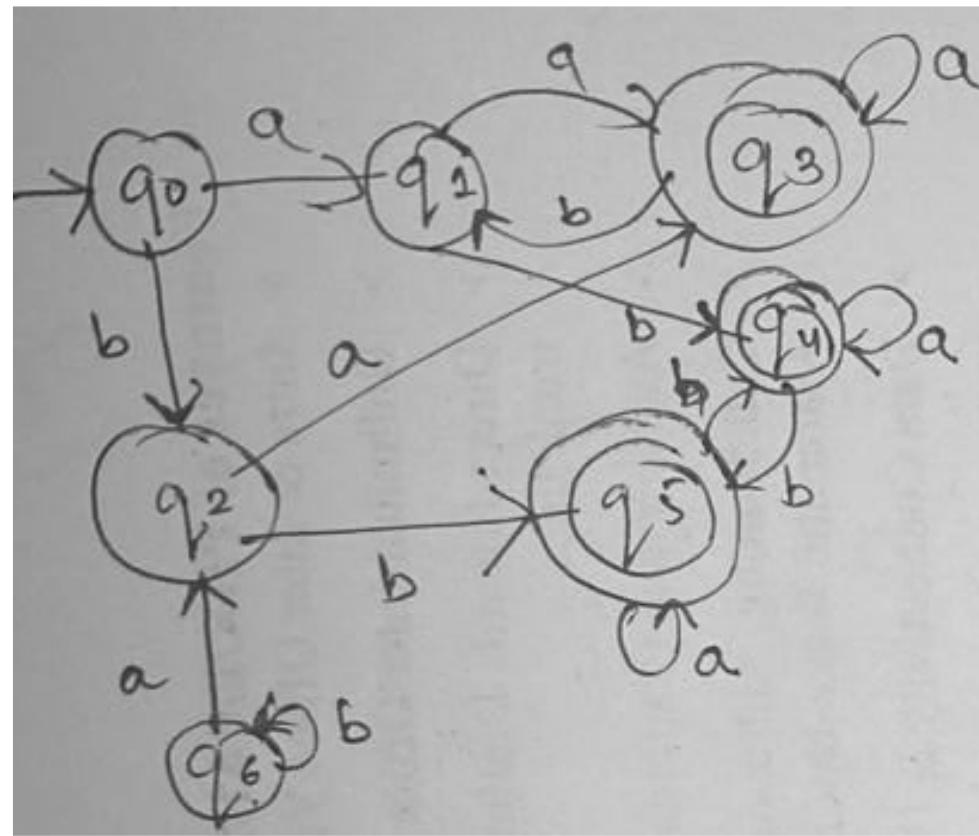
$$\pi_2 = \{q_0, q_2\} \quad \{q_1\} \quad \{q_3\} \quad \{q_4\}$$

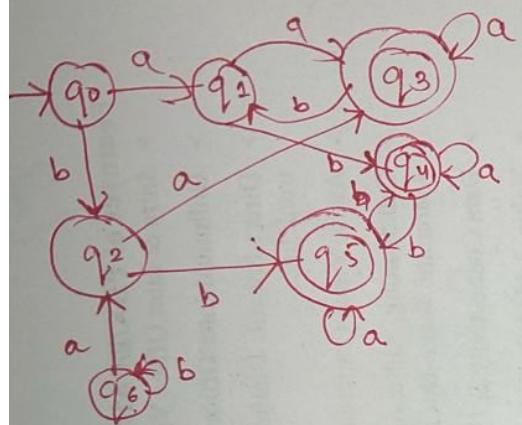
$$\pi_3 = \text{same } \{ \text{stop} \}$$

step 1 remove  $q_5$

	a	b
$q_0$	$q_1$	$q_2$
$q_1$	$q_2$	$q_3$
$q_2$	$q_1$	$q^2$
$q_3$	$q_2$	$q^4$
$q_4$	$q_1$	$q_2$







	a	b
$q_0$	$q_1$	$q_2$
$q_1$	$q_3$	$q_4$
$q_2$	$q_3$	$q_5$
$q_3$	$q_3$	$q_1$
$q_4$	$q_4$	$q_5$
$q_5$	$q_5$	$q_4$
$q_6$	$q_2$	$q_6$

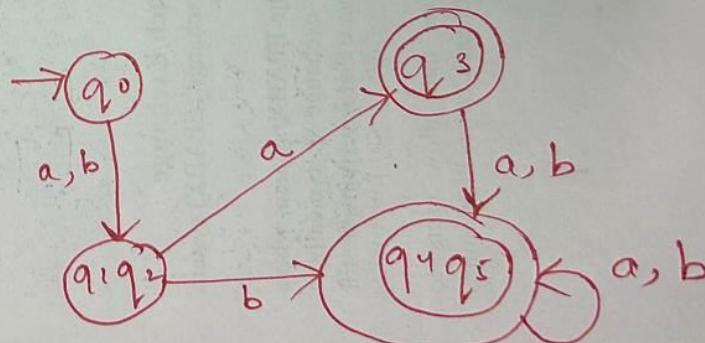
Step 2 remove  $q_6$

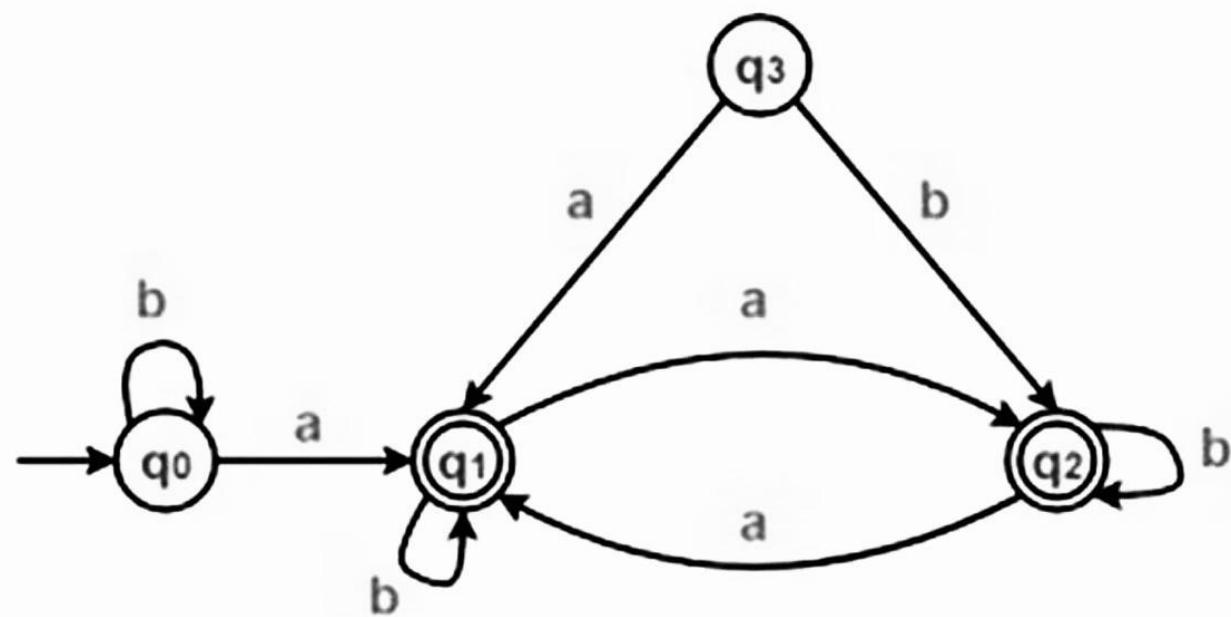
Step 3

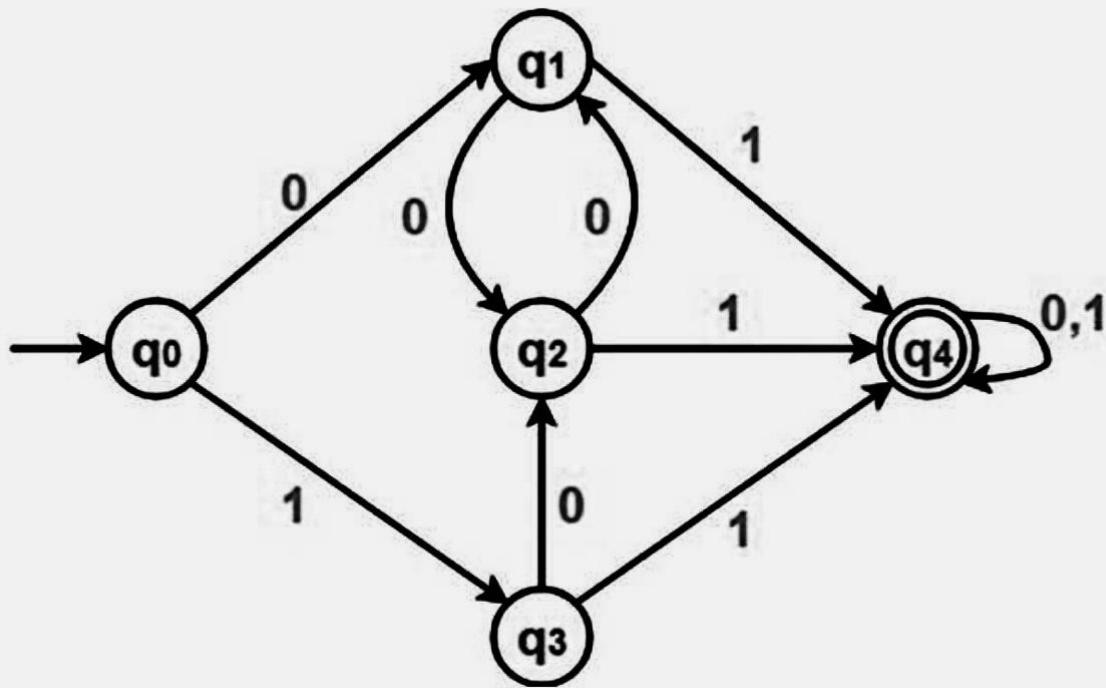
$$\pi_0 = \{ q_0 \mid q_1 \mid q_2 \} \quad \{ q_3 \mid q_4 \mid q_5 \}$$

$$\pi_1 = \{ q_0 \} \quad \{ q_1 \mid q_2 \} \quad \{ q_3 \} \quad \{ q_4 \mid q_5 \}$$

$\pi_2 \Rightarrow$  same







# 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.
- Two regular expressions P and Q are equivalent (we write  $P = Q$ ) if P and Q represent the same set of strings.

**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, .....}

# Regular Language

**Regular languages are formal languages that regular expressions can describe and can also be recognized by finite automata.**

If for instance,  $a, b \in \Sigma$ , then

- $R = a$  denotes the  $L = \{a\}$
- $R = a.b$  denotes  $L = \{ab\}$  concatenation
- $R = a + b$  denotes  $L = \{a, b\}$  Union
- $R = a^*$  denotes the set  $\{\epsilon, a, aa, aaa, \dots\}$  known as Kleene closure.
- $R = a^+$  Positive closure  $\{a, aa, aaa\dots\}$
- $R = (a + b)^*$  denotes  $\{a, b\}^*$

# Operators

When we view  $a$  in  $\Sigma$  as a regular expression, we denote it by  $a$ .

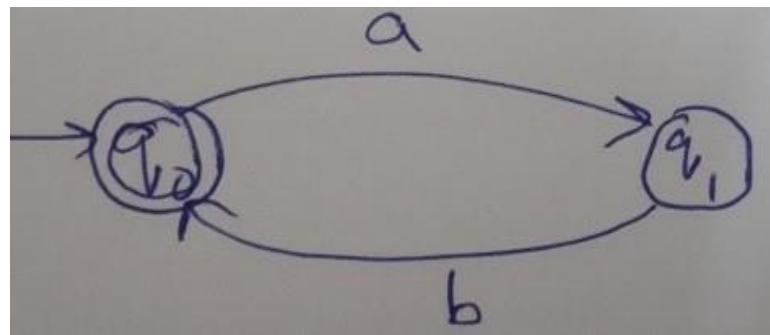
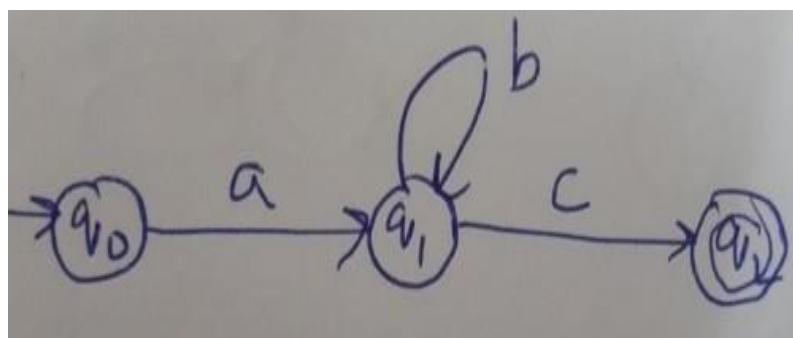
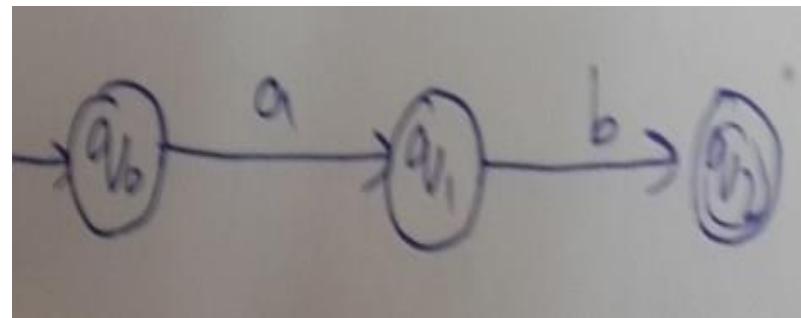
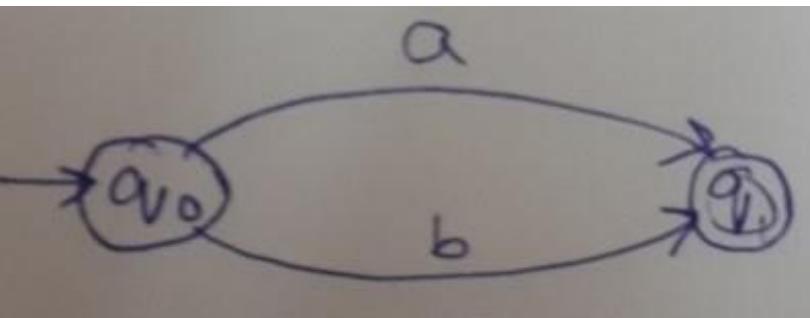
- If  $R$  is a regular expression, then  $(R)$  is also a regular expression.
- The iteration (or closure) of a regular expression  $R$  written as  $R^*$ , is also a regular expression.
- The iteration (or closure) of a regular expression  $R$  written as  $R^+$  , is also a regular expression.
  - The concatenation of two regular expressions  $R_1$  and  $R_2$ , written as  $R_1 R_2$ , is also a regular expression.
  - The union of two regular expressions  $R_1$  and  $R_2$ , written as  $R_1 + R_2$ , is also a regular expression

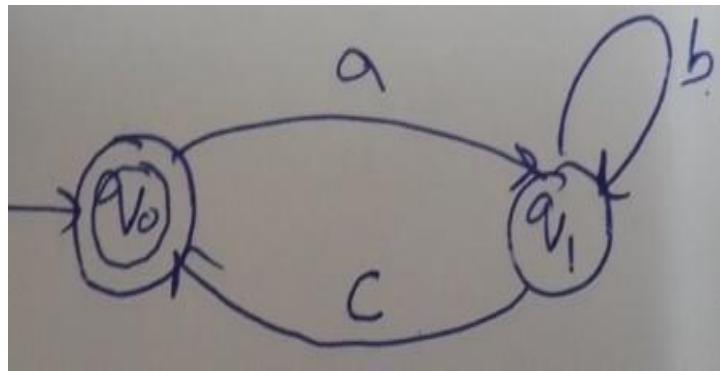
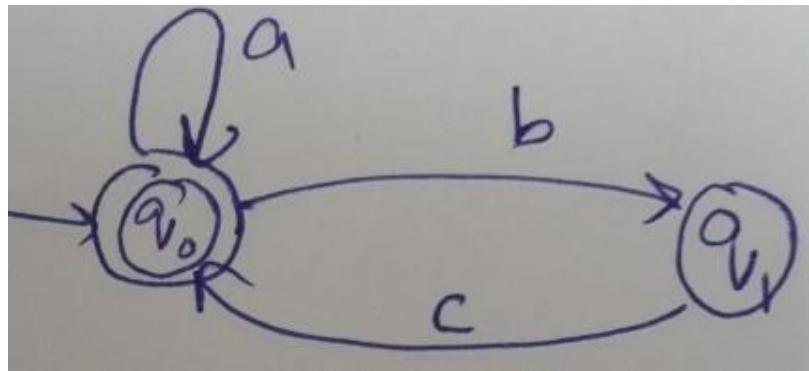
# **Operator precedence**

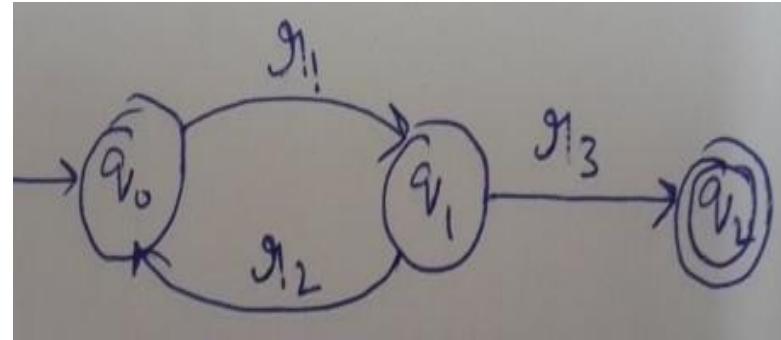
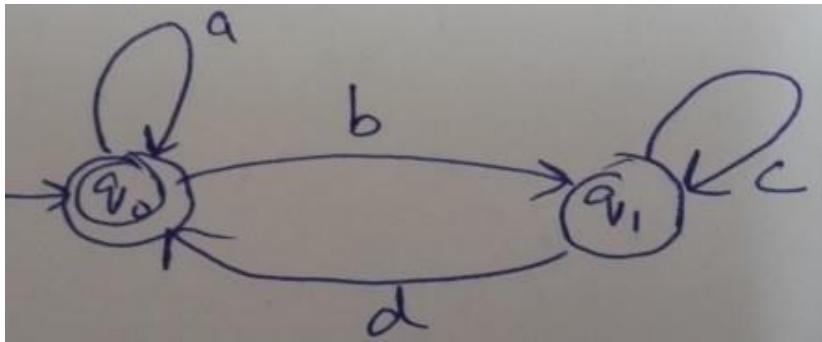
**The precedence order to solve is**

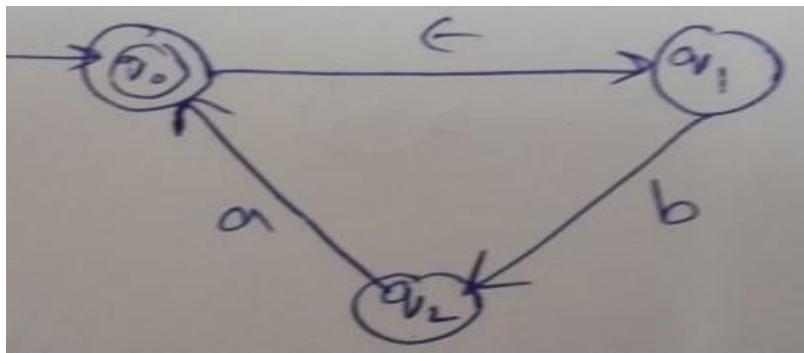
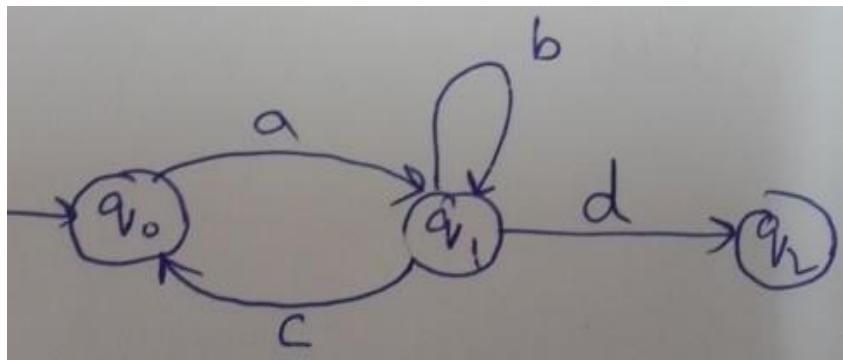
- () Bracket
- \* (Kleene Closure)
- + Positive Closure
- Concatenation
- Union

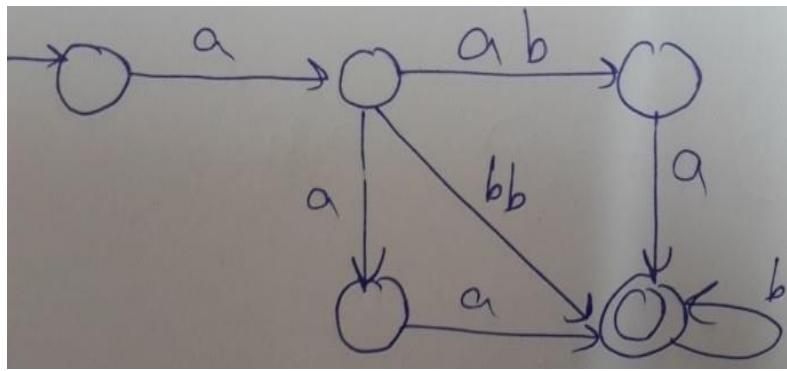
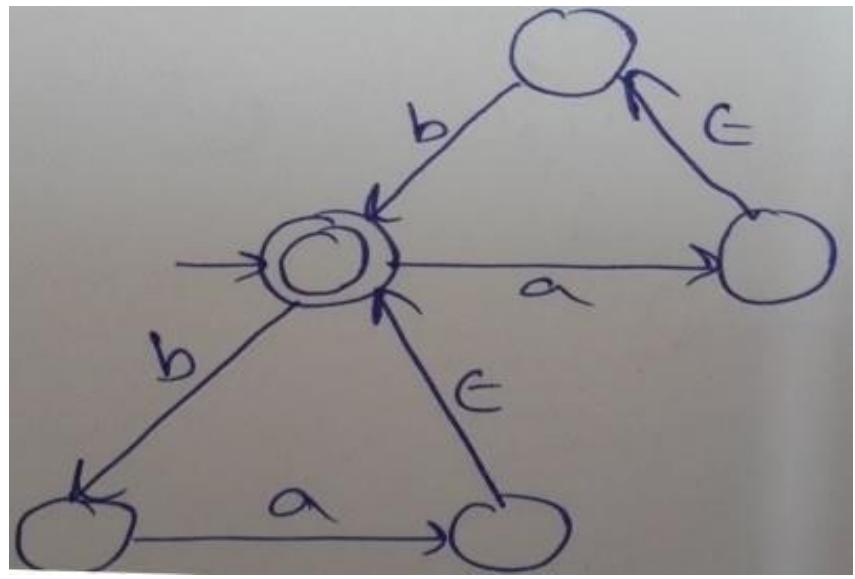
# Finite Automata to regular expression Conversion

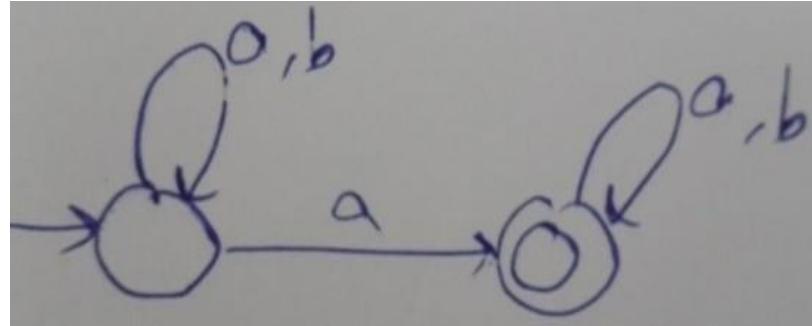
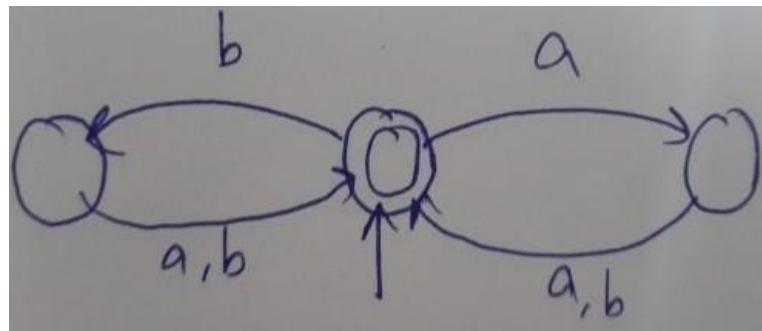


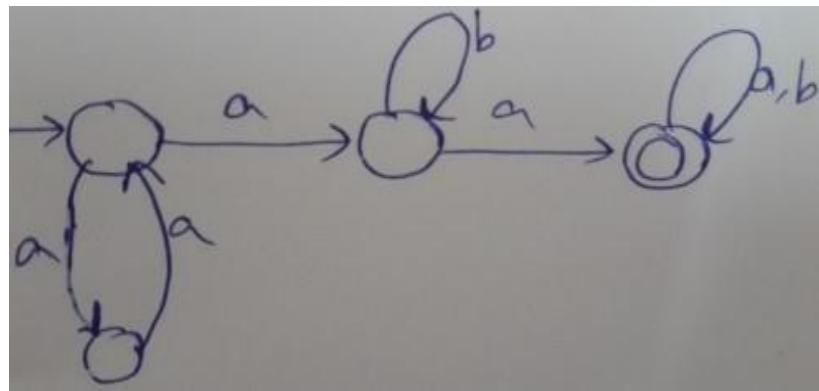
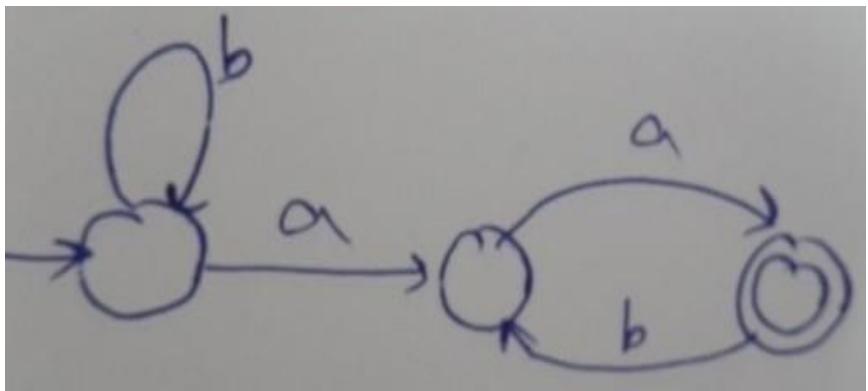


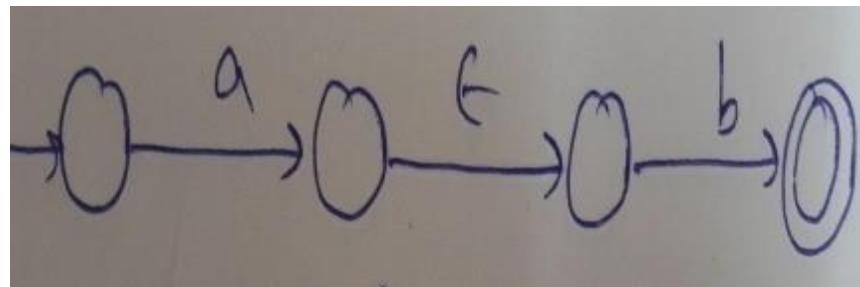
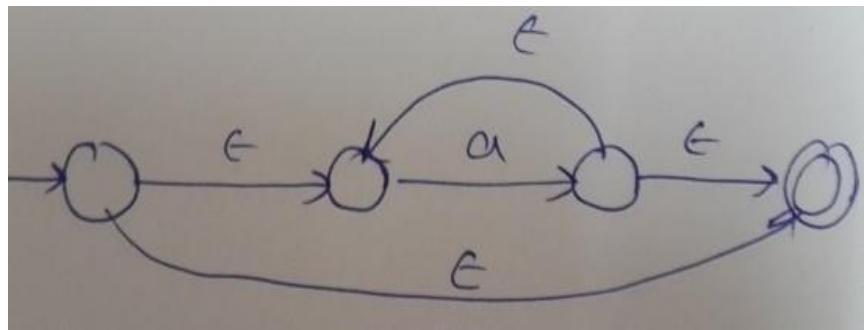


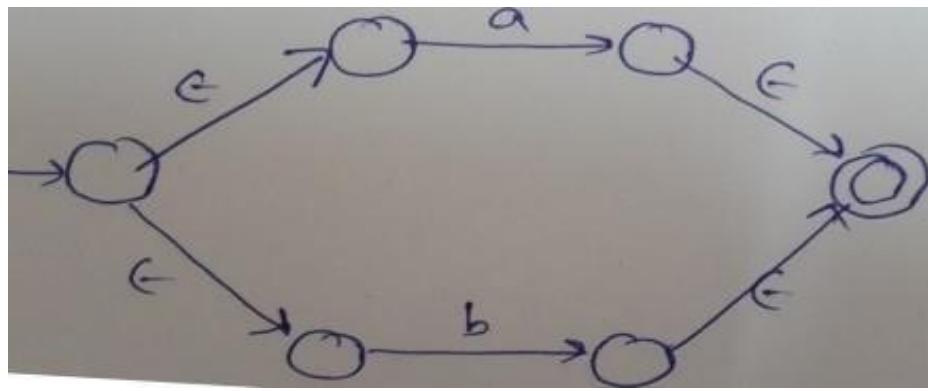












**Q** design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$ , where every accepted string 'w' starts with substring s = 'abb'

Q design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$ . where every accepted string 'w' ends with substring s = 'bab'?

**Q** Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$ . where every accepted string 'w' contains substring s = 'aba' ?

**Q** Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string start and end with same symbol?

Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string start and end with different symbol ?

Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , is like  $w=AX$  where  $A = 'aaa/bbb'$  ?

Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , is like  $w=XS$ .  $S = 'aaa/bbb'$  ?

Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$  such that every accepted string  $w$ , is like  $w=XSX$ .  $S = aaa/bbb$  ?

**Q** Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$ , such that every string 'w' accepted must be like i)  $|w| = 3$  ii)  $|w| \leq 3$  iii)  $|w| \geq 3$

Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$  such that for every accepted string 2nd symbol from left end is always b

Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$  such that for every accepted string 4th symbol from right end is always a.

**Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$ , such that every string 'w' where  $|W| = 0(\text{mod } 3)$ ?**

**Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$ , such that every string 'w' where  $|W| = 3(\text{mod } 4)$ ?**

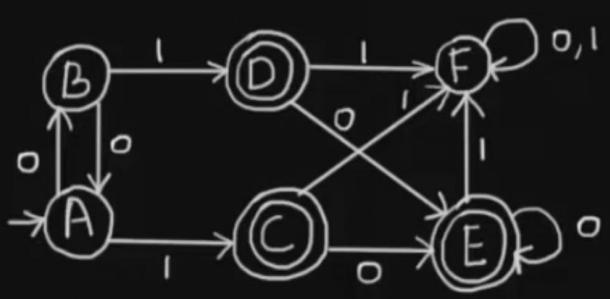
Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$ , such that every string 'w' where  $|W|a = 0(\text{mod } 3)$

Q Design a regular expression that represent all strings over the alphabet  $\Sigma = \{a, b\}$ , such that every string 'w' where  $|W|b = 2(\text{mod } 3)$

# Closure properties of Regular languages

Operation	Description
Union	Combining two languages, L1 and L2, by taking all strings that are in either L1 or L2.
Concatenation	Combining two languages, L1 and L2, by creating all strings where a string from L1 is followed by a string from L2.
Closure (Kleene Star)	Creating a new language by taking all possible strings formed by concatenating zero or more copies of strings from the original language.
Complement	Creating a new language by taking all strings over the alphabet that are not present in the original language.
Intersection	Creating a new language by taking all strings that are present in both L1 and L2.
Difference	Creating a new language by taking all strings that are in L1 but not in L2.
Reversal	Creating a new language by reversing each string in the original language.

# Myhill-Nerode Theorem



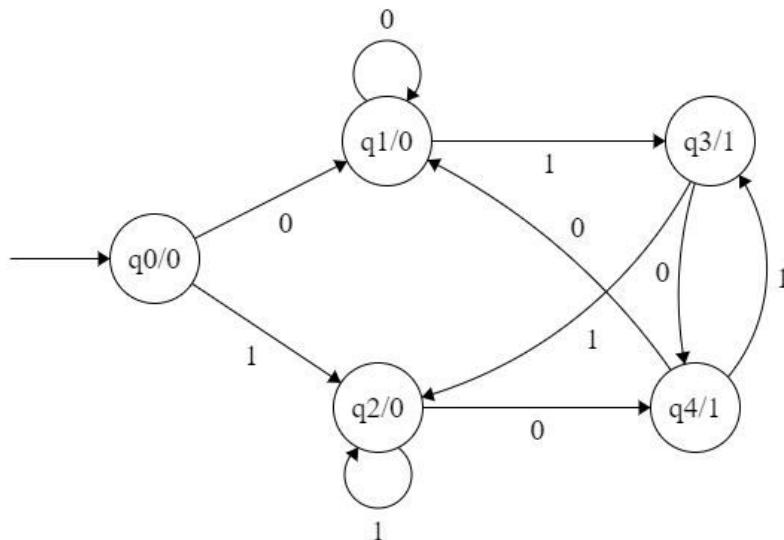
# Myhill-Nerode Theorem

Algorithm

- **Input** – DFA
- **Output** – Minimized DFA
- **Step 1** – Draw a table for all pairs of states  $(Q_i, Q_j)$  not necessarily connected directly [All are unmarked initially]
- **Step 2** – Consider every state pair  $(Q_i, Q_j)$  in the DFA where  $Q_i \in F$  and  $Q_j \notin F$  or vice versa and mark them. [Here  $F$  is the set of final states]
- **Step 3** – Repeat this step until we cannot mark anymore states –
  - If there is an unmarked pair  $(Q_i, Q_j)$ , mark it if the pair  $\{\delta(Q_i, A), \delta(Q_j, A)\}$  is marked for some input alphabet.
- **Step 4** – Combine all the unmarked pair  $(Q_i, Q_j)$  and make them a single state in the reduced DFA.

# Moore Machine

- In a Moore machine, each state is associated with a specific output
- They are FS M/c's that help in producing outputs
- There is no concept of final states and dead states and no concepts of final states



- Input: 11
- Output: 000

- If the input string has a length of  $n$ , then the length of the output string will be  $n + 1$
- A Moore machine's response to an empty input string is the output associated with the initial state.

A Moore machine is a six-tuple  $(Q, \Sigma, O, \delta, \lambda, q_0)$ , where

- $Q$  is a finite set of states
- $\Sigma$  is the input alphabet:
- $O$  is the output alphabet.
- $\delta$  is the transition function  $Q \times \Sigma$  into  $Q$
- $\lambda$  is the output function mapping  $Q$  into  $O$
- $q_0$  is the initial state

# Design a Moore m/c from the given transition table

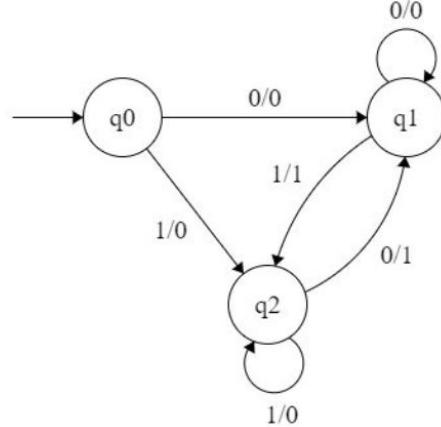
Present state	Next State		output
	Input 0	Input 1	
→ $q_0$	$q_1$	$q_2$	a
$q_1$	$q_1$	$q_3$	c
$q_2$	$q_2$	$q_3$	a
$q_3$	$q_3$	$q_2$	b

Construct a Moore machine that prints 'a' whenever the sequence '01' occurs,  
 $\Sigma = \{0, 1\}$ ,  $\Delta = \{a, b\}$ ?

Construct a Moore machine that counts the occurrences of sequence 'ab' in terms of 1,  $\Sigma = \{a, b\}$ ,  $\Delta = \{0, 1\}$ ?

# Mealy machines

- In a Mealy machine, the output symbol depends on the current state and the current input.
- They are FS M/c's that help in producing outputs
- There is no concept of final states and dead states and no concepts of final states
- Mealy machine do not response for empty string
- If the input string has a length of n, then the length of the output string will be n
  - Input: 11
  - Output: 00



# Mealy machines

Mealy machines are also finite state machines with output value and their output depends on the present state and current input symbol. It can be defined as  $(Q, q_0, \Sigma, O, \delta, \lambda')$  where:

- $Q$  is a finite set of states.
- $q_0$  is the initial state.
- $\Sigma$  is the input alphabet.
- $O$  is the output alphabet.
- $\delta$  is the transition function which maps  $Q \times \Sigma \rightarrow Q$ .
- ' $\lambda'$ ' is the output function that maps  $Q \times \Sigma \rightarrow O$ .

# Design a Mealy m/c from the given transition table

Present state	Next state			
	a = 0		a = 1	
state	output	state	output	
$\rightarrow q_1$	$q_3$	0	$q_2$	0
$q_2$	$q_4$	1	$q_4$	0
$q_3$	$q_2$	1	$q_1$	1
$q_4$	$q_4$	1	$q_3$	0

Construct a Mealy machine that prints 'a' whenever the sequence '01' occurs,  
 $\Sigma = \{0, 1\}$ ,  $\Delta = \{a, b\}$ ?

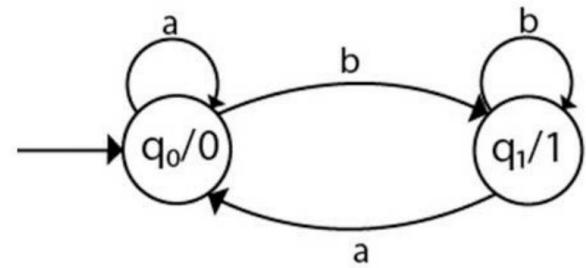
Construct a Mealy machine that counts the occurrences of sequence 'ab' in terms of 1,  $\Sigma = \{a, b\}$ ,  $\Delta = \{0, 1\}$ ?

Moore	Mealy
Output depends only upon the present state.	Output depends on the present state as well as present input.
Produces an output corresponding to the initial state.	Does not produce any output.
Moore machine also places its output in the state.	Mealy Machine places its output on the transition.
Generates an output string of length $n+1$ .	Generates an output string of length $n$
If input changes, output does not change, as moore machine is a type of finite state machine where the output is determined solely by the current state, not by the current input.	If input changes, output also changes, because in mealy machine output depends on the present state as well as present input.

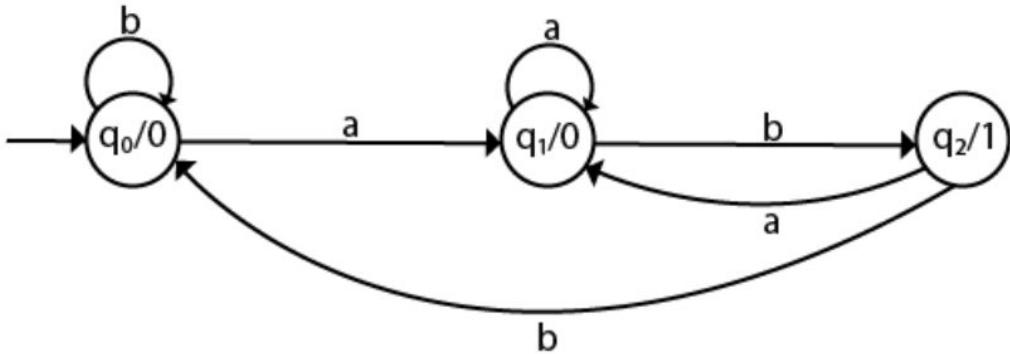
## Conversion from Moore machine to Mealy Machine

Q	a	b	Output( $\lambda$ )
q0	q0	q1	0
q1	q0	q1	1

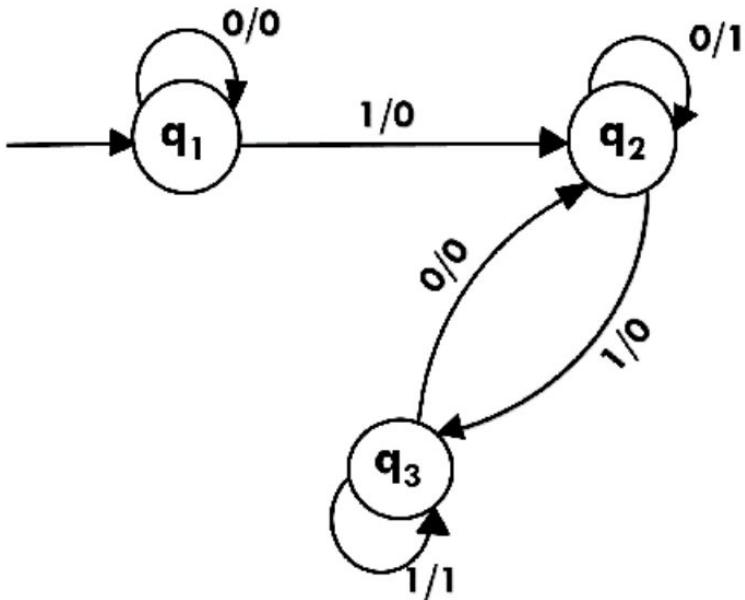
# Conversion from Moore machine to Mealy Machine



# Conversion from Moore machine to Mealy Machine

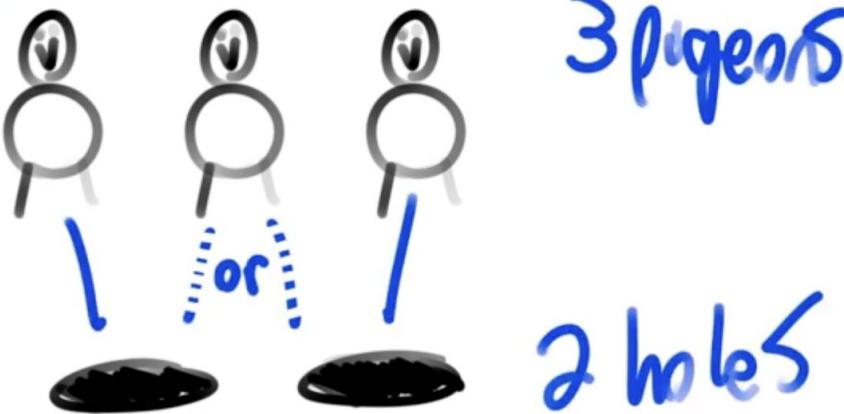


# Conversion from Moore machine to Mealy Machine



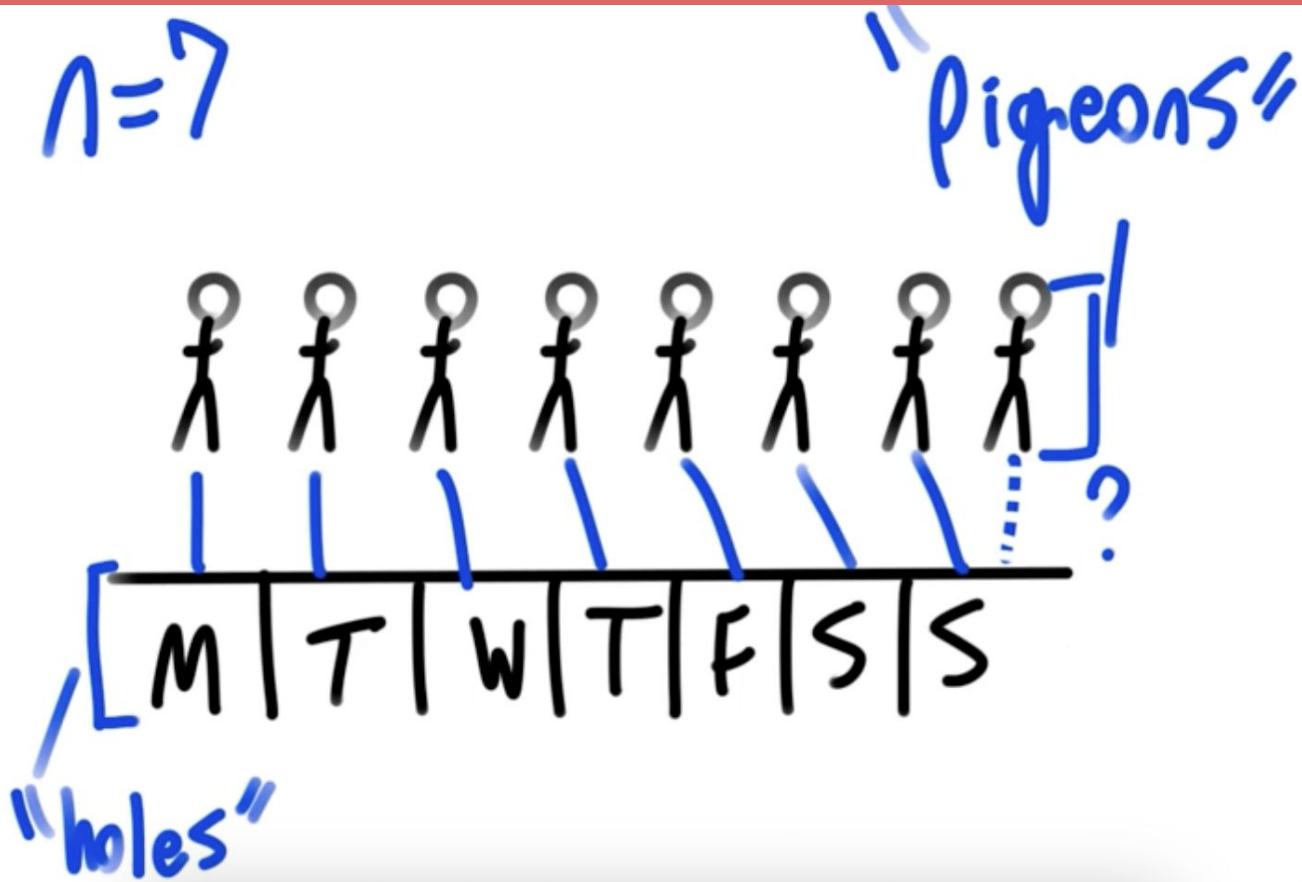
# Pigeonhole Principle

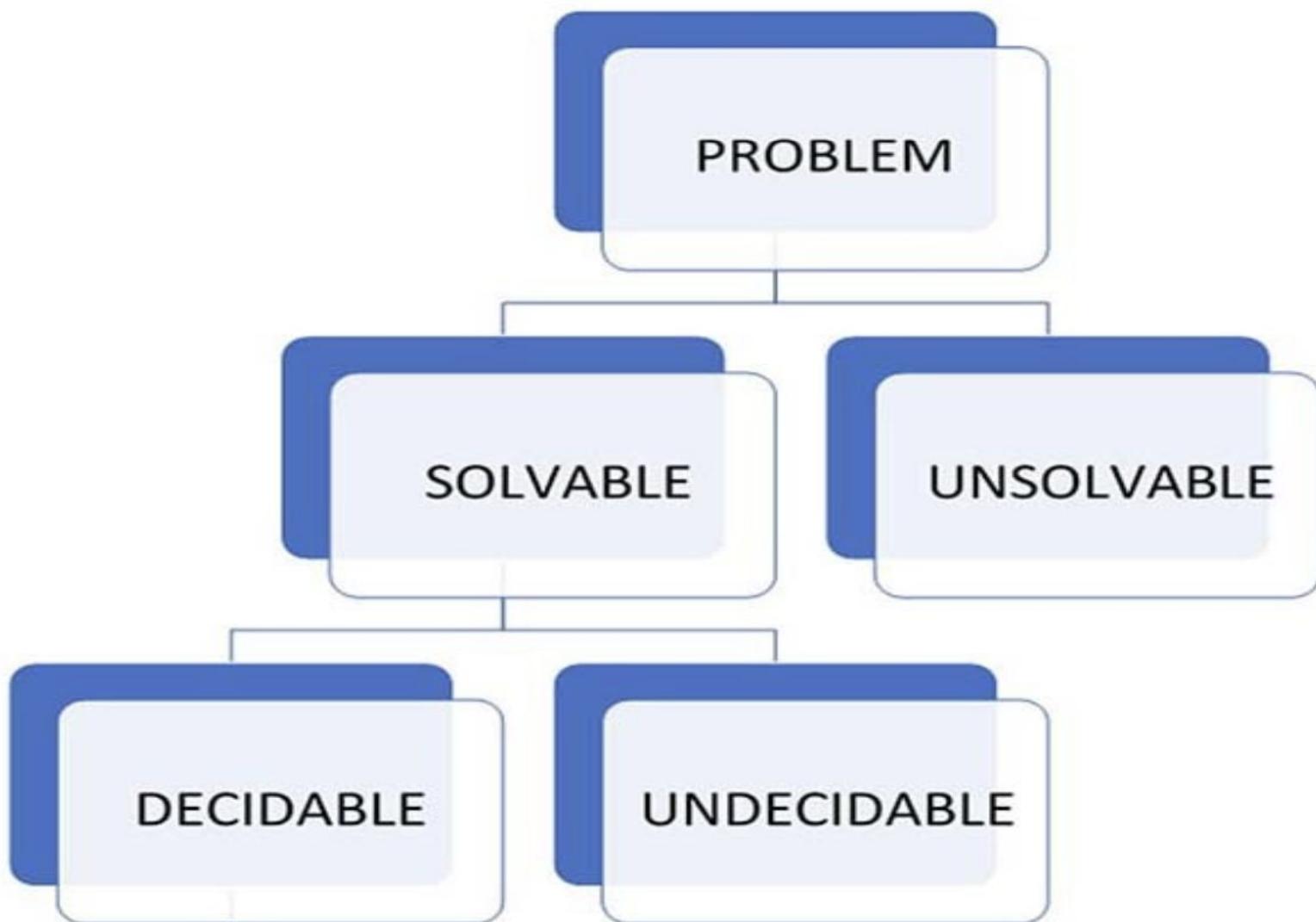
let  $n=2$



If  $n$  pigeonholes are occupied by  $n+1$  or more pigeons, then at least one pigeonhole is occupied by greater than one pigeon.

Eg: At least how many people must have their birthday on the same day if 8 people are assembled in a room?





# Decidability and Undecidability

**Decidable** Decidable problem is one for which there exists an algorithm that can determine the answer (yes or no) for every input in a finite amount of time. In the context of regular languages, decidability means that there exists a finite procedure (algorithm) to determine certain properties of these languages.

**Undecidable** Undecidable problems are those for which no algorithm can determine the answer (yes or no) for all possible inputs.

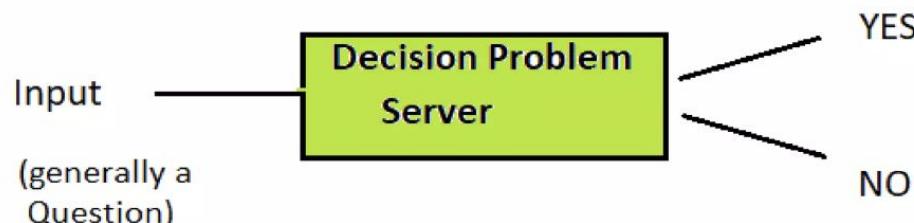
**Decision properties** of regular languages are specific questions about these languages, such as emptiness, finiteness, membership, equivalence.

# Decision properties

- Approximately all the properties are decidable in case a finite automaton. Here we will use machine model to proof decision properties.

- i) Emptiness
- ii) Non-emptiness
- iii) Finiteness
- iv) Infiniteness
- v) Membership
- vi) Equality

Any "Decision Property" looks like this:



## Membership

**Problem:** Determine whether a string  $www$  belongs to a regular language  $L$ .

**Algorithm:**

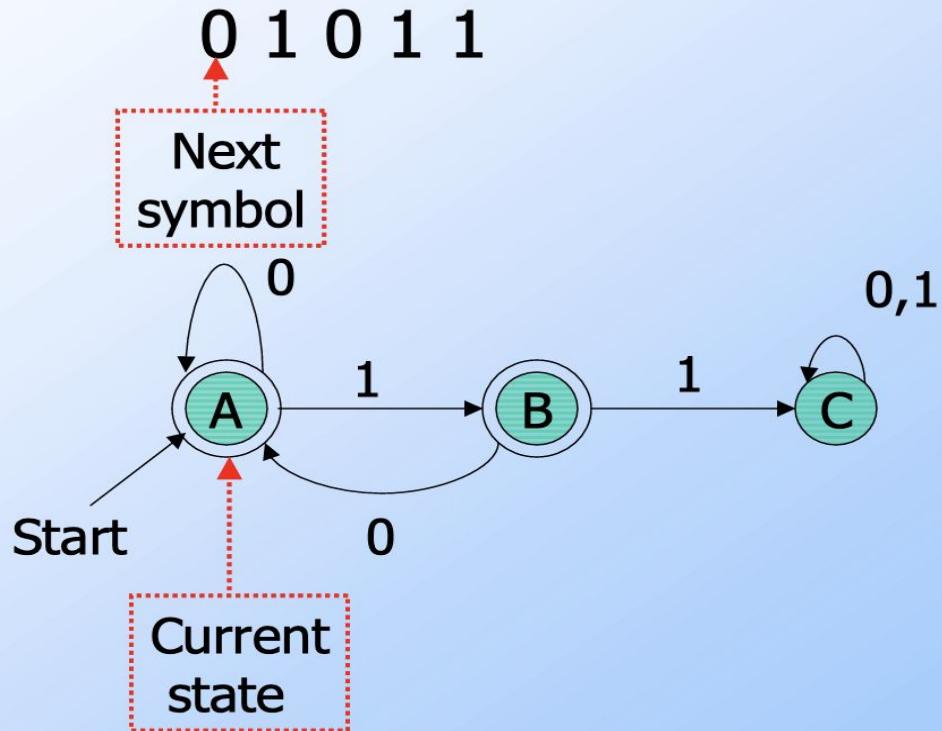
- Construct a finite automaton  $M$  for  $L$ .
- Simulate  $M$  on input  $w$ .
- If  $M$  ends in an accepting state,  $w$  is in  $L$ ; otherwise, it is not.

**Example:**

Regular expression:  $a^*b$

- String "aab" is in the language.
- String "aaa" is not in the language.

# Example: Testing Membership



## 1. Emptiness

**Problem:** Determine whether a regular language  $L$  is empty, meaning it contains no strings.

**Step 1:** – select the state that cannot be reached from the initial states & delete them (remove unreachable states)

**Step 2:** – if the resulting machine contains at least one final states, so then the finite automata accepts the non-empty language.

**Step 3:** – if the resulting machine is free from final state, then finite automata accepts empty language.

**Example:**

- Regular expression:  $a^*$ 
  - The language is not empty (contains strings like "", "a", "aa", etc.).

# Finiteness & Infiniteness

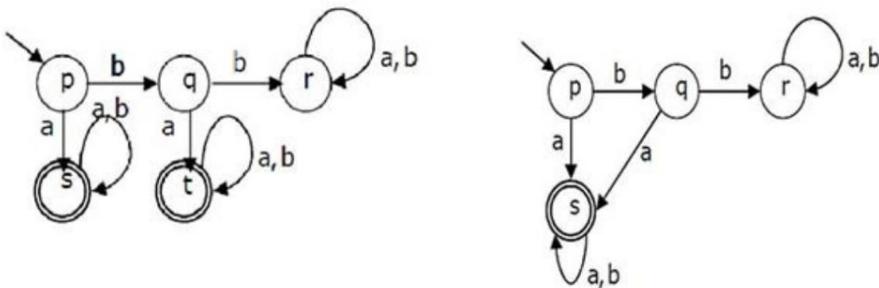
- Step 1: – Select the state that cannot be reached from the initial state & delete them (remove unreachable states)
- Step 2: – Select the state from which we cannot reach the final state & delete them (remove dead states)
- Step 3: – If the resulting machine contains loops or cycles then the finite automata accepts infinite language

Step 4: – If the resulting machine do not contain loops or cycles then the finite automata accepts finite language.

- Regular expression:  $a^+$ 
  - The language is infinite (contains strings like "a", "aa", "aaa", etc.).

# Equality

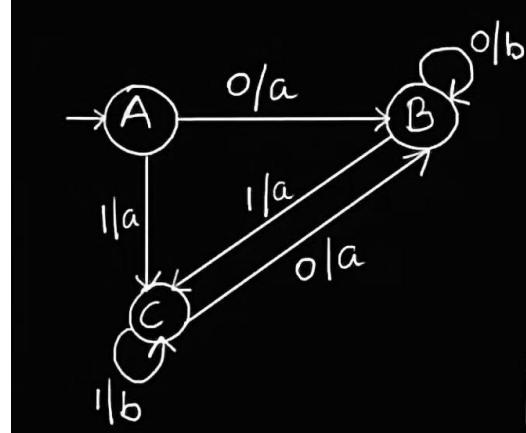
- Two finite state automata  $M_1$  &  $M_2$  is said to be equal if and only if, they accept the same language.
- Minimise the finite state automata and and the minimal DFA will be unique.



# Conversion from Mealy machine to Moore Machine

State	a	b
$\rightarrow q_0$	$q_3, 0$	$q_1, 1$
$q_1$	$q_0, 1$	$q_3, 0$
$q_2$	$q_2, 1$	$q_2, 0$
$q_3$	$q_1, 0$	$q_0, 1$

# Conversion from Mealy machine to Moore Machine



# Conversion from Mealy machine to Moore Machine

Present State	Next State			
	a		b	
	State	O/P	State	O/P
$q_1$	$q_1$	1	$q_2$	0
$q_2$	$q_4$	1	$q_4$	1
$q_3$	$q_2$	1	$q_3$	1
$q_4$	$q_3$	0	$q_1$	1

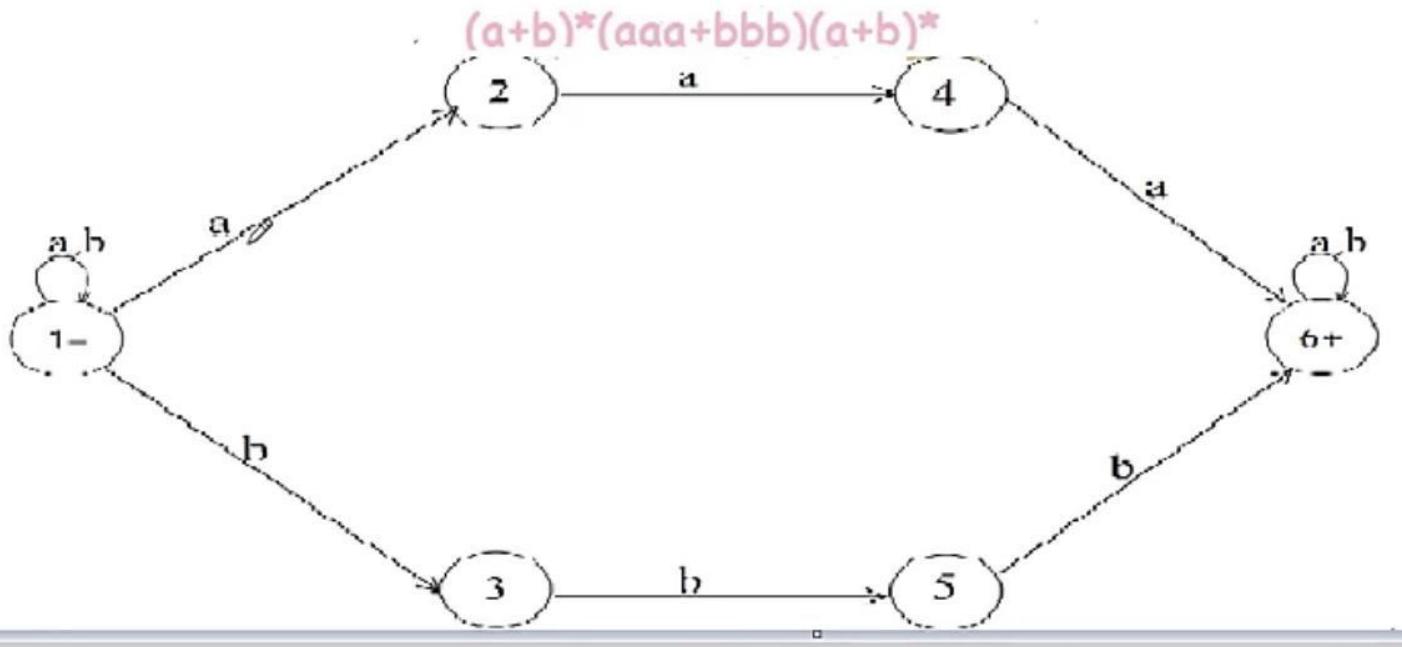
**For Practice Questions, refer to unit 2 playlist**

# Transition Graphs

1. Finite number of states in which at least one initial state and final state  
*(i.e., can have more than one initial/final state)*
2. Dead End State is not required
3. One letter can move to more than one state
4. Can read more than one character at a time  
*(i.e., it allows substring transition)*
5. Can move to other state without any input like  $\epsilon$ -nfa  
*(i.e., it allows epsilon / null / lambda transition)*

## Example

if  $\Sigma = \{a, b\}$  Draw TG for all strings containing  $bbb$  or  $aaa$



Q: Design a TG of a language that accepts the strings having 00 as substring

$$R = (0+1)^*00(0+1)^*$$

Q: Design a TG of a language that accepts the strings having different starting and ending of letters defined over  $\Sigma=\{p,q\}$

$$R = p(p+q)^*q + q(p+q)^*p$$

# Kleene's Theorem

Kleene's Theorem says two main things:

- First part: If you have a regular expression, you can construct a finite automaton that recognizes the same language (set of strings) described by that regular expression.
- Second part: If you have a finite automaton (DFA or NFA), you can write a regular expression that describes the same language that the automaton recognizes.

Regular expressions and finite automata are equivalent in power for defining regular languages.

**Theorem 5.4** (Kleene's theorem) The class of regular sets over  $\Sigma$  is the smallest class  $\mathcal{R}$  containing  $\{a\}$  for every  $a \in \Sigma$  and closed under union, concatenation and closure.

**Union:** If you have two regular sets, their union is also a regular set.

For example, if  $R_1$  and  $R_2$  are regular sets, then  $R_1 \cup R_2$  is also a regular set.

**Concatenation:** If you have two regular sets, their concatenation is also a regular set.

**Closure (Kleene Star):** If you have a regular set, the Kleene star of that set is also a regular set. This means any number of repetitions (including zero) of the strings in the set. For example, if  $R$  is the set of strings that match  $a$ , then  $R^*$  is the set of strings that match  $a^*$ .

# Regular Grammar to Finite Automata

Note: No. of States in the automata will be equal to no. of non-terminals plus one. Each state in automata represents each non-terminal in regular grammar. Additional state will be Final State.

## Steps for Transition of Automata

1. For every production  $A \rightarrow aB$ :

- Create a transition  $\delta(A, a) = B$  (edge labeled 'a' from  $A$  to  $B$ ).

2. For every production  $A \rightarrow a$ :

- Create a transition  $\delta(A, a) = \text{Final State}$ .

3. For every production  $A \rightarrow \epsilon$ :

- Create a transition  $\delta(A, \epsilon) = A$  and make  $A$  a final state.

**S->01S/1**

$S \rightarrow 011S/01$

$$S \rightarrow aS \mid bA \mid b$$
$$A \rightarrow aA \mid bS \mid a$$

S -> 0A/1B/0/1

A -> 0S/1B/1

B -> 0A/1S

$S \rightarrow aS \quad | \quad bS \quad | \quad aA$

$A \rightarrow bB$

$B \rightarrow aC$

$C \rightarrow a$

S->001S/10A

A->101A/0/1

**S->01S/1**

# Non-regular languages

Non-regular languages are languages that **cannot** be recognized by a finite automaton (DFA or NFA) and **cannot** be expressed using regular expressions. These languages require more computational power, such as a pushdown automaton (for context-free languages) or a Turing machine. They fail the **pumping lemma for regular languages**, which is often used to prove their non-regularity.

## Example:

The language  $L=\{ a^n b^n \mid n \geq 0 \}$  is non-regular because a finite automaton cannot remember how many aa's have been seen to match them with bb's.

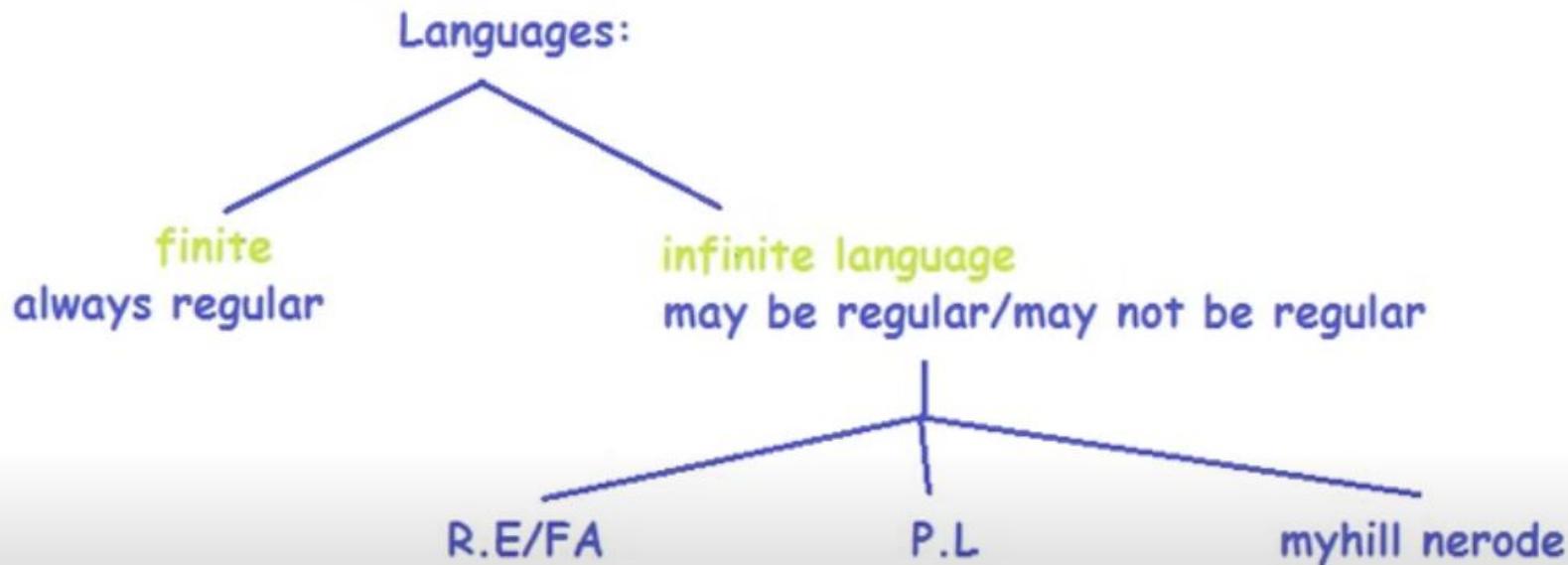
## Proving non-regularity using Pumping Lemma.:

① Assume language is regular.

② Given  $ppp$ , choose a string that is at least length  $ppp$ .

③ Show how pumping & splitting the string does **NOT** satisfy the pumping lemma.

# Pumping Lemma



# PUMPING LEMMA

"Pumping" refers to the idea that a certain segment of a string within a language can be repeated ("pumped") multiple times without causing the string to fall out of the language.



## Pumping Lemma For Regular Languages

- 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.
- Pumping Lemma is used to prove that some of the language is non-regular.
- For the pumping lemma, i/p is NRL & o/p is also NRL.

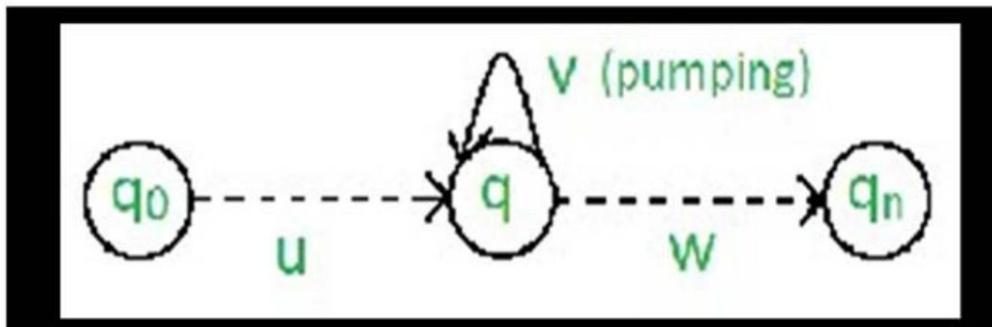
# Pumping Lemma For Regular Languages

- For any regular language  $L$ , there exists an integer  $n$ , such that for all  $z \in L$  with  $|z| \geq n$ , there exists  $u, v, w \in \Sigma^*$ , such that  $z = uvw$ , and

- $|uv| \leq n$

- $|v| \geq 1$

- for all  $i \geq 0$ :  $uv^iw \in L$



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

## Process of pumping Lemma

- Suppose that we need to prove that language L is a non-regular.
- Assume that L is a regular language, then L must satisfy pumping lemma property.
- Choose  $z \in L$  such that  $|z| \leq n$ , split z into 3 parts.
  - $|uv| \leq n$
  - $|v| \geq 1$
- If there exist at least one variable for i such that  $uv^iw \notin L$ , then L does not satisfy pumping lemma property so it is a contradiction, therefore language L is non-regular.

Q Proof that language  $L = \{a^m b^n \mid m=n\}$  is non-regular?

$$L = \{ab, aabb, aaabbb, aaaabbbb, \dots\}$$

$$z \in L$$

$$z = a^k b^k = a^{k-1} ab^k$$

$$\text{For } i=1 \rightarrow uv^i w \rightarrow a^k b^k$$

$$\text{For } i=2 \rightarrow uv^i w \rightarrow a^{k+1} b^k$$

**Q Proof that language  $L = \{a^m b^n \mid m < n\}$  is non-regular?**

# Arden's Theorem

If  $P$  &  $Q$  are two regular expressions over  $\Sigma$  & if  $P$  does not contain  $\epsilon$ , then the following equation in  $R$  given by

$$R = Q + RP$$

has a unique solution

$$R = QP^*$$

## Conditions

- FA should not contain  $\epsilon$ -transition
- FA should have only one initial state

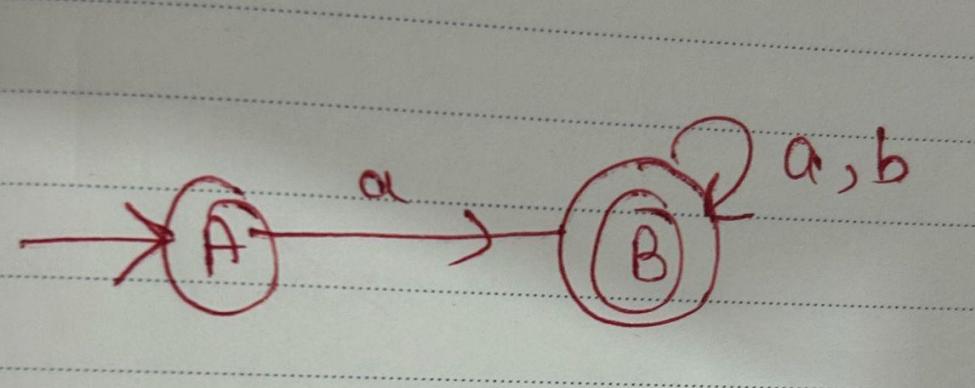
## Why?

### Arden's Theorem

- To check the equivalence between two REs
- Conversion of DFA to RE

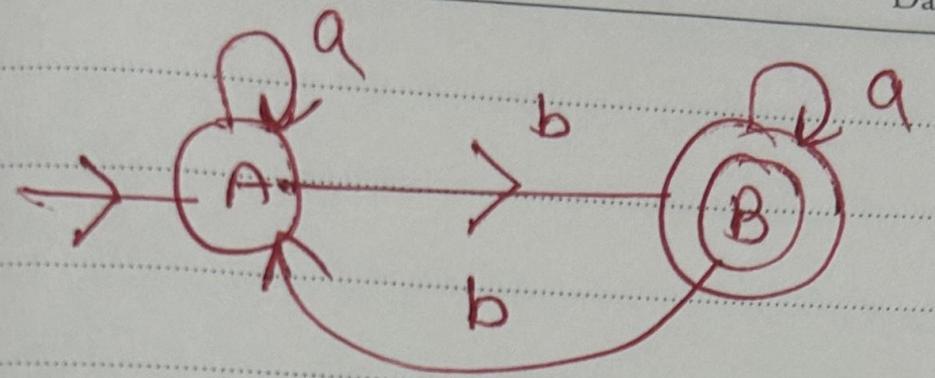
# STEPS

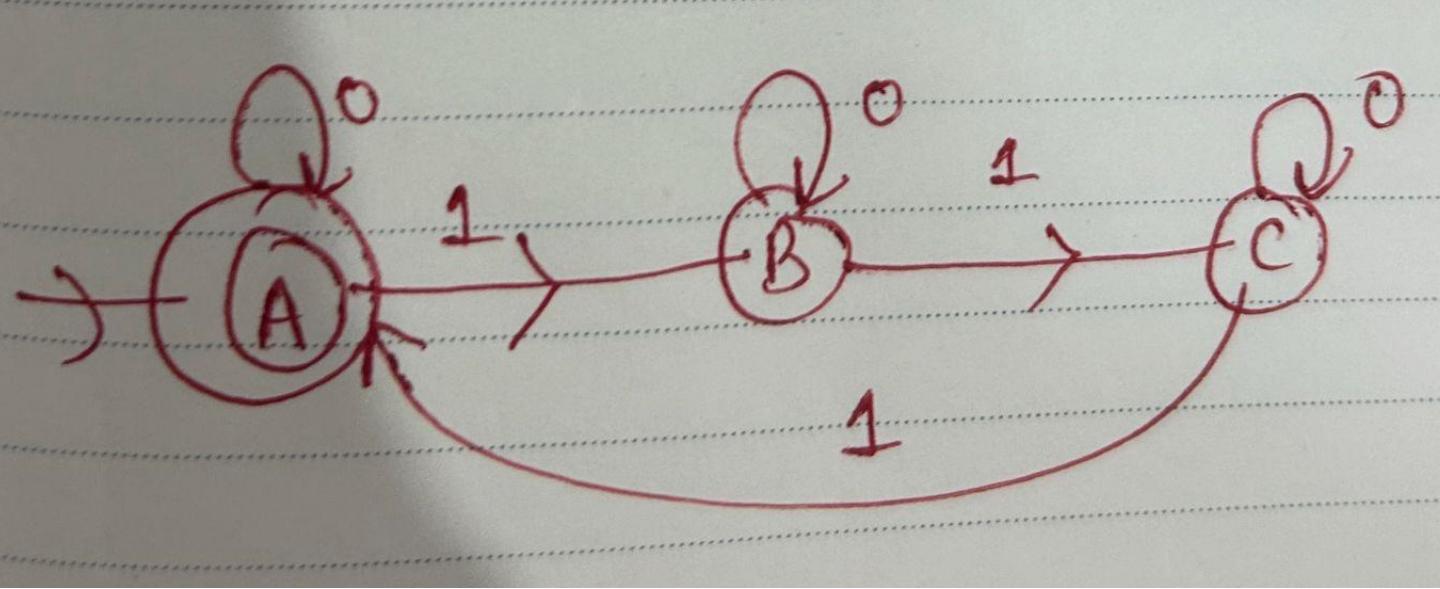
1. Write state equations by looking at the incoming transitions
2. Add epsilon to the start state equation
3. Perform any combination of substitution rearrangement and apply **ARDEN'S THEOREM** until we are able to get the final states equation in terms of alphabets.



1. Write state equations by looking at the incoming transitions
2. Add epsilon to the start state equation
3. Perform any combination of substitution rearrangement and apply **ARDEN'S THEOREM** until we are able to get the final states equation in terms of alphabets.

Dat





$$(0 + 1 \cdot 0^* \cdot 1 \cdot 0^* \cdot 1)^*$$

# Context Free Grammar

Context Free Grammar is formal grammar, the syntax or structure of a formal language can be described using context-free grammar (CFG), a type of formal grammar. The grammar has four tuples:  $(V_n, \Sigma, P, S)$ .

NOTE:  $\{(V_n, \Sigma, P, S)\}$  also denoted as  $(V_n, T, P, S)$ . Where  $T (\Sigma)$  is a set of terminals}

Where,  $V_n$ : is a non empty set of Variables or Non Terminals,  $V$  is finite.

- $\Sigma$  is a finite nonempty set whose elements are called terminals,  $V_n \cap \Sigma = \emptyset$ .
- $S$  is a special variable (i.e., an element of  $V_n (S \in V_n)$ ) called the start symbol. Like every automaton has exactly one initial state, similarly every grammar has exactly one start symbol.
- $P$  is a finite set whose elements are  $\alpha \rightarrow \beta$ . where  $\alpha$  and  $\beta$  are strings on  $V_n \cup \Sigma$ .  $\alpha$  has at least one symbol from  $V_n$ , the element of  $P$  are called productions or production rules or rewriting rules.  $\{\Sigma \cup V_n\}^*$  some writer refers it as total alphabet

## Points to remember

- Reverse substitution is not permitted. For example, if  $S \rightarrow AB$  is a production, then we can replace  $S$  by  $AB$  but we cannot replace  $AB$  by  $S$ .
- No inversion operation is permitted. For example, if  $S \rightarrow AB$  is a production, it is not necessary that  $AB \rightarrow S$  is a production.

**Consider the following grammar & and identify its language (CFG TO CFL)**

**Consider the following grammar and identify its language?**

$$S \rightarrow aAb$$

$$A \rightarrow aB / b$$

$$B \rightarrow c$$

Consider the following grammar & and identify its language (CFG TO CFL)

**S** -> **A****B**/**B****b**

**A** -> **b**/**c**

**B** -> **d**

Consider the following grammar & and identify its language (CFG TO CFL)

$$S \rightarrow aSb/\epsilon$$

Consider the following grammar & and identify its language (CFG TO CFL)

$S \rightarrow aA / abS$

$A \rightarrow bS / b$

**Ques. Consider the following grammar & and identify its language (CFG TO CFL)**

**S -> aAB**

**A -> aA / ∈**

**B->b**

$$L = \{a^m b \mid m \geq 1\}$$

**Ques. Consider the following grammar & and identify its language (CFG TO CFL)**

**S -> AB**

**A -> aA / ∈**

**B -> bB / b**

$L = \{a^n b^m \mid n \geq 0, m > 0\}$

Consider the following grammar & and identify its language (CFG TO CFL)

$$S \rightarrow aSa / bSb / \epsilon$$

$$L = \{\omega \in \{a,b\}^* \mid \omega = \omega^R\}$$

## Convert CFL to CFG

1.  $a^n b^n, n \geq 1$
2.  $a^n b^{n+2}, n \geq 0$

1.  $S \rightarrow aSb$   
 $S \rightarrow ab$
1.  $S \rightarrow aSb \mid bb$

## Convert CFL to CFG

3.  $a^{2n}b^n, n \geq 0$

4.  $a^{2n+3}b^n, n \geq 0$

3.  $S \rightarrow aaSb$

$S \rightarrow / \epsilon$

4.  $S \rightarrow aaSb$

$S \rightarrow aaa$

## Convert CFL to CFG

$$L = \{a^m b^n \mid m > n, n \geq 0\}$$

$$S \rightarrow AS_1$$
$$S_1 \rightarrow aS_1b \mid \lambda$$
$$A \rightarrow aA \mid a$$

## Convert CFL to CFG

$$L = \{a^m b^m c^n \mid m, n \geq 0\}$$

$$S \rightarrow S_1 C$$
$$S_1 \rightarrow aS_1b \mid \epsilon$$
$$C \rightarrow cC \mid \epsilon$$

## Convert CFL to CFG

$$L = \{w \in \{a, b\}^* \mid \#a(w) = \#b(w)\}$$

where  $\#a(w)$  denotes the number of  $a$ 's in the string  $w$ , and  $\#b(w)$  denotes the number of  $b$ 's in the string  $w$ .

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

Write CFG that generates a language which accepts all palindromic strings of even length over the alphabet {a,b}

Write CFG that generates a language which accepts all palindromic strings of odd length over the alphabet {a,b}

Write CFG that generates a language which accepts all palindromic strings over the alphabet {a,b}

# REGULAR GRAMMARS

Regular grammar generates regular language.

- Regular grammar can be of two types either left linear or right linear.
- Left regular grammar, support two types of production

$$A \rightarrow a/Ba$$

$$\begin{array}{ll} A, B \in V_n & |A| = |B| = 1 \\ a \in \Sigma^* & \end{array}$$

- Right regular grammar

$$A \rightarrow a/aB$$

$$\begin{array}{ll} A, B \in V_n & |A| = |B| = 1 \\ a \in \Sigma^* & \end{array}$$

- however, if left-linear rules and right-linear rules are combined, the language need no longer be regular.

# Right and Left linear Regular Grammars

$$A \rightarrow bB$$
$$B \rightarrow \epsilon \mid aB \mid bB$$
$$S \rightarrow Aa$$
$$A \rightarrow ab$$

$S \rightarrow Aa$

$A \rightarrow ab\gamma\epsilon$

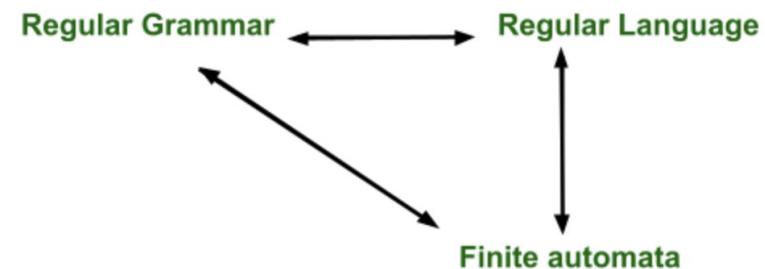
# Conversion of RLG to LLG for language L

**Step 1:** Reverse the FA for language L

**Step 2:** Write the RLG for it.

**Step 3:** Reverse the right linear grammar (RHS).

Ex: FA for accepting strings that start with b

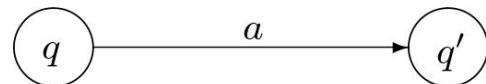


All have the same power and can be converted to other

# FA TO CFG CONVERSION

**FA  $\rightarrow$  CFG: algorithm.** Let us show how to transform a finite automaton (FA) into a context-free grammar (CFG).

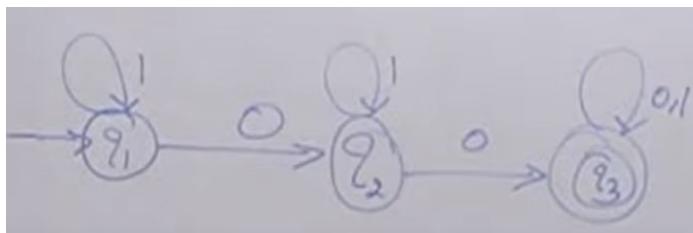
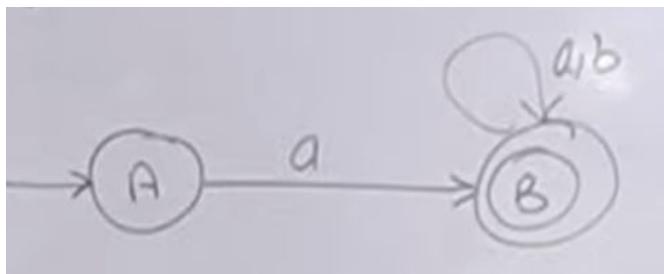
- To each state  $q$  of the FA, introduce a new variable  $Q$ .
- The variable corresponding to the starting state will be the starting variable of the new CFG.
- For each transition of the finite automaton



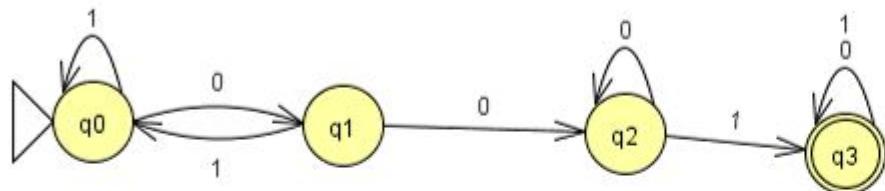
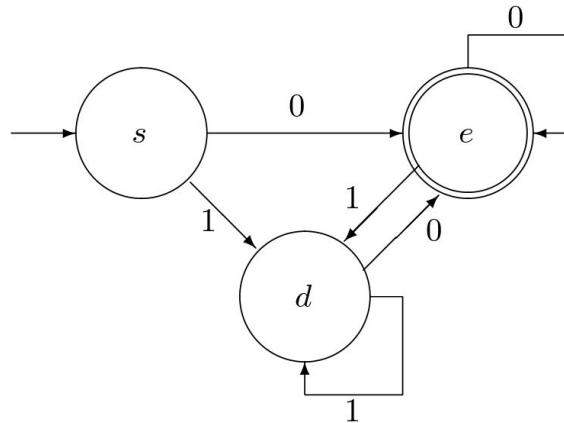
we add a rule  $Q \rightarrow aQ'$ .

- For each final state  $f$  of the FA, we add a rule  $F \rightarrow \varepsilon$ .

## EXAMPLES



## EXAMPLES









**S->01S/1**

**S->011S/01**

$$S \rightarrow aS \mid bA \mid b$$
$$A \rightarrow aA \mid bS \mid a$$

S -> 0A/1B/0/1

A -> 0S/1B/1

B -> 0A/1S

$$S \rightarrow aS \mid bS \mid aA$$
$$A \rightarrow bB$$
$$B \rightarrow aC$$
$$C \rightarrow a$$



S-> 001S/10A

A->101A/0/1

S -> 0S/1A/1

A -> 0A/1A/0/1



## Closure Properties of Regular Languages

These properties describe how regular languages behave under certain operations. Regular languages are closed under these operations, meaning performing these operations on regular languages results in a regular language:

1. **Union:** If you take the union of two regular languages, the result is a regular language.
2. **Intersection:** If you intersect two regular languages, the result is a regular language.
3. **Complement:** If you take the complement of a regular language, the result is a regular language.
4. **Difference:** If you take the difference between two regular languages, the result is a regular language.
5. **Concatenation:** If you concatenate two regular languages, the result is a regular language.
6. **Kleene Star:** If you apply the Kleene star operation to a regular language (repeating the language zero or more times), the result is a regular language.
7. **Reversal:** If you reverse the strings of a regular language, the result is a regular language.
8. **Homomorphism:** Applying a homomorphism (a substitution of symbols) to a regular language results in a regular language.



# PDA to CFG Conversion

# CFG to PDA Conversion

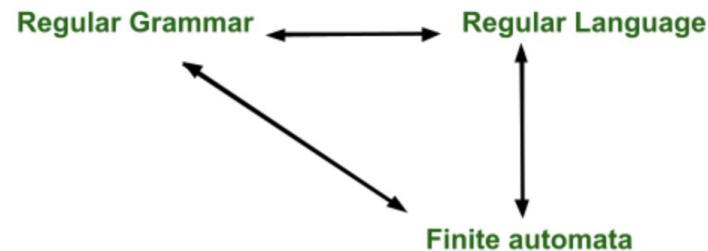
# Conversion of RLG to LLG for language L

**Step 1:** Reverse the FA for language L

**Step 2:** Write the RLG for it.

**Step 3:** Reverse the right linear grammar.

Example: FA for accepting strings that start with b



# Simplification of CFG

The process of **Removal of Unit productions, Null productions and Useless Symbols** is known as **Simplification of CFG**.

## Removal of Null or Empty productions

- The production of the form  $A \rightarrow \epsilon$  is known as null production or empty production, here we try to remove them by replacing equivalent derivation.

$S \rightarrow AbB$   
 $A \rightarrow a / \epsilon$   
 $B \rightarrow b / \epsilon$

$S \rightarrow A b B / b B / A b / b$   
 $S \rightarrow a$   
 $S \rightarrow b$

$S \rightarrow a S b / \epsilon$  $S' \rightarrow S / \epsilon$   
 $S \rightarrow a S b / ab$  $S \rightarrow ABA$  $A \rightarrow 0A | \epsilon$  $B \rightarrow 1B | \epsilon$  $S' \rightarrow S / \epsilon$   
 $S \rightarrow BA / A / AB / B / AA$   
 $A \rightarrow 0A / 0$   
 $B \rightarrow 1B / 1$  $S \rightarrow AB$  $A \rightarrow a / \epsilon$  $B \rightarrow b / \epsilon$  $S' \rightarrow S / \epsilon$   
 $S \rightarrow AB / B / A$   
 $A \rightarrow a$   
 $B \rightarrow b$

$S \rightarrow 0X \mid 1Y \mid 01$

$X \rightarrow 0S \mid 00$

$Y \rightarrow 0S \mid 00 \mid 1$

## Removal of unit productions

- The production of the form  $A \rightarrow B$  where  $A, B \in V_n$ ,  $|A| = |B| = 1$ , is known as unit production.

$S \rightarrow Aa$

$A \rightarrow a \mid B$

$B \rightarrow d$

Step 1: To remove  $A \rightarrow B$ , add production  $A \rightarrow x$  to the grammar rule whenever  $B \rightarrow x$  occurs in the grammar.

Step 2: Delete  $A \rightarrow B$  from the grammar.

Step 3: Repeat the first and the second steps until all the unit productions are removed.

# Remove Unit Productions from:

$S \rightarrow 0X \mid 1Y \mid Z$   
 $X \rightarrow 0S \mid 00$   
 $Y \rightarrow 1 \mid X$   
 $Z \rightarrow 01$

## Step 1: Remove Unit Productions

Unit productions are of the form  $A \rightarrow B$ , where both  $A$  and  $B$  are non-terminals.

Here, we have:

- $S \rightarrow Z$  ( $Z$  can be replaced)
- $Y \rightarrow X$  ( $X$  can be replaced)

So, the grammar becomes:

$S \rightarrow 0X \mid 1Y \mid 01$   
 $X \rightarrow 0S \mid 00$   
 $Y \rightarrow 0S \mid 00 \mid 1$

## Step 2: Replace Unit Productions

- Replace  $S \rightarrow Z$  with  $S \rightarrow 01$ , since  $Z \rightarrow 01$ .
- Replace  $Y \rightarrow X$  with  $Y \rightarrow 0S|00$ , since  $X \rightarrow 0S|00$ .

# Remove Unit Productions from:

$S \rightarrow aA\ b$   
 $A \rightarrow B\ / \ a$   
 $B \rightarrow C\ / \ b$   
 $C \rightarrow D\ / \ c$   
 $D \rightarrow d$

## Step 1: Identify Unit Productions

The unit productions are:

- $A \rightarrow B$
- $B \rightarrow C$
- $C \rightarrow D$

## Step 2: Replace Unit Productions

- Since  $D \rightarrow d$ , replace  $C \rightarrow D$  with  $C \rightarrow d$ , so now  $C \rightarrow d|c$ .
- Since  $B \rightarrow C$ , replace  $B \rightarrow C$  with  $B \rightarrow d|c$ , so now  $B \rightarrow d|c|b$ .
- Since  $A \rightarrow B$ , replace  $A \rightarrow B$  with  $A \rightarrow d|c|b$ , so now  $A \rightarrow a|d|c|b$ .

So, the grammar becomes:

$S \rightarrow a\ A\ b$   
 $A \rightarrow a\ | \ d\ | \ c\ | \ b$

## Removal of useless symbols

- The variables which are not involved in the derivation of any string is known as useless symbol
- Select the Variable that cannot be reached from the start symbol of the grammar and remove them along with their all production.
- Select variable that are reachable from the start symbol but which does not derive any terminal, remove them along with their productions

$S \rightarrow aAB$

$A \rightarrow a$     $A \rightarrow$

$B \rightarrow b$

$C \rightarrow d$

$S \rightarrow aA/aB$

$A \rightarrow b$

# Remove Useless Productions from:

$$\begin{aligned}S &\rightarrow aAB \mid bA \mid aC \\A &\rightarrow aB \mid b \\B &\rightarrow aC \mid d\end{aligned}$$

## Step 1.1: Find Symbols That Can Derive Terminal Strings

- $B \rightarrow aC$  and  $B \rightarrow d$ . Here,  $B \rightarrow d$  is valid, but we need to check  $C$ .
- $C$  does not appear on the right-hand side anywhere else, meaning it has no production.  $C$  is useless.
- If  $C$  is useless, then:
  - $B \rightarrow aC$  is useless, but  $B \rightarrow d$  is valid.
  - $S \rightarrow aC$  is also useless.

Thus, we remove all rules involving  $C$ .

## Grammar after removing useless $C$

$$\begin{aligned}S &\rightarrow aAB \mid bA \\A &\rightarrow aB \mid b \\B &\rightarrow d\end{aligned}$$

## Step 1.2: Check Reachability

Every symbol can be reached from  $S$ , and every non-terminal produces terminal strings

## Remove Useless Productions from:

$T \rightarrow xxY \mid xbX \mid xxT$

$X \rightarrow xX$

$Y \rightarrow xy \mid y$

$Z \rightarrow xz$

**Chomsky Normal Form.** A grammar where every production is either of the form  $A \rightarrow BC$  or  $A \rightarrow c$  (where  $A, B, C$  are arbitrary variables and  $c$  an arbitrary symbol).

A context free grammar (CFG) is in Chomsky Normal Form (CNF) if all production rules satisfy one of the following conditions:

- A non-terminal generating a terminal (e.g.;  $X \rightarrow x$ )
- A non-terminal generating two non-terminals (e.g.;  $X \rightarrow YZ$ )
- **Exception:** Start symbol generating  $\epsilon$  is allowed (e.g.;  $S \rightarrow \epsilon$ )

# Conversion of CFG to CNF

**Step 1.** Eliminate null, unit and useless productions.

**Step 2.** Eliminate terminals from RHS if they exist with other terminals or non-terminals. e.g., production rule  $X \rightarrow xY$  can be decomposed as:

$X \rightarrow ZY$

$Z \rightarrow x$

**Step 3.** Eliminate RHS with more than two non-terminals.

e.g., production rule  $X \rightarrow XYZ$  can be decomposed as:

$X \rightarrow PZ$

$P \rightarrow XY$

# Convert the following CFG to CNF

$$1. \ S \rightarrow aSb \mid ab$$

## Step 1: Introduce New Variables for Terminals

Since CNF does not allow mixed terminals and non-terminals, we introduce new variables

$$A \rightarrow a$$

$$B \rightarrow b$$

Now, rewrite the productions using these:

$S \rightarrow aSb \mid ab$  becomes:  $S \rightarrow ASB \mid AB$

## Step 2: Convert to CNF Format

The rule  $S \rightarrow ASB$  is not in CNF because the right-hand side has three symbols.

We introduce a new variable:  $C \rightarrow SB$

Now, rewrite  $S \rightarrow ASB$  as:

$$S \rightarrow AC$$

$$C \rightarrow SB$$

## Final CNF:

$$1. \ S \rightarrow AC$$

$$2. \ C \rightarrow SB$$

$$3. \ S \rightarrow AB$$

$$4. \ A \rightarrow a$$

$$5. \ B \rightarrow b$$

# EXAMPLES

1.

$S \rightarrow aAb / bB$

$A \rightarrow a / b$

$B \rightarrow b$

2.

Consider the CFG:

$S \rightarrow aXbX$

$X \rightarrow aY \mid bY \mid \epsilon$

$Y \rightarrow X \mid c$

## Greibach Normal Form (GNF)

- A non-terminal that generates a terminal. For instance,  $X \rightarrow x$ .
- A start symbol that generates  $\epsilon$ . For instance,  $S \rightarrow \epsilon$ .
- A non-terminal that generates a terminal followed by any number of non-terminals. For instance,  $S \rightarrow xXSY$ .

$A \rightarrow a\alpha$

$A \in V_n$

$a \in \Sigma$

$\alpha \in V_n^*$

## EXAMPLES

**S → aSb / ab**

**S → AY**

**X → x | SX**

**Y → y**

**A → x**

**S → xY**

**X → x | xY X**

**Y → y**

## EXAMPLES

$S \rightarrow aAb / bB$

$A \rightarrow a / b$

$B \rightarrow b$

# Pushdown Automata

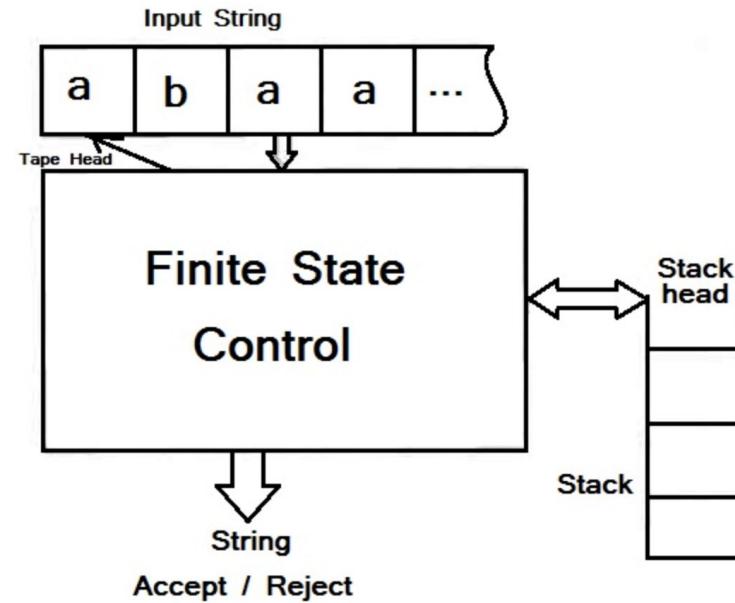
A Pushdown Automata (PDA) is a way to implement context-free Grammar

1. It is more powerful than FSM.
2. FSM has very limited memory but PDA has more memory.
3. PDA= Finite State Machine + Stack

# Block diagram of pushdown automata

A Pushdown Automata has three Components:

1. An input tape
2. A finite Control Unit.
3. A stack with infinite size.



# Formal Definition

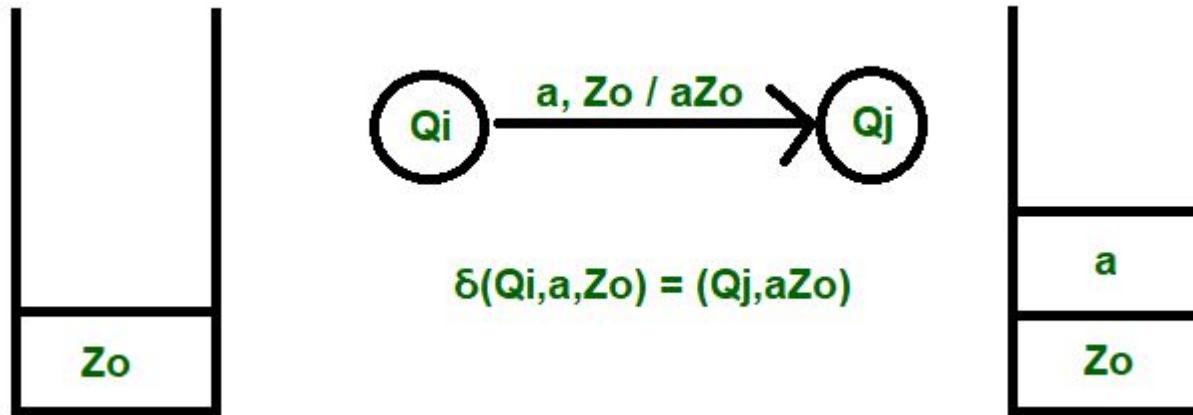
A Pushdown Automata (PDA) can be defined as a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, I, F)$ :

- $Q$  is the set of states
- $\Sigma$  is the set of input symbols
- $\Gamma$  is the set of pushdown symbols (which can be pushed and popped from stack)
- $q_0$  is the initial state
- $Z$  is the initial pushdown symbol (which is initially present in stack)
- $F$  is the set of final states
- $\delta$  is a transition function which maps  $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$  into  $Q \times \Gamma^*$ . In a given state, PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack.

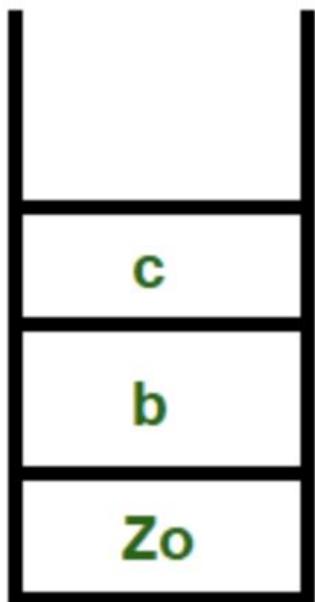
# Push,Pop and NOOP/Skip Operations on PDA

## Representation of States

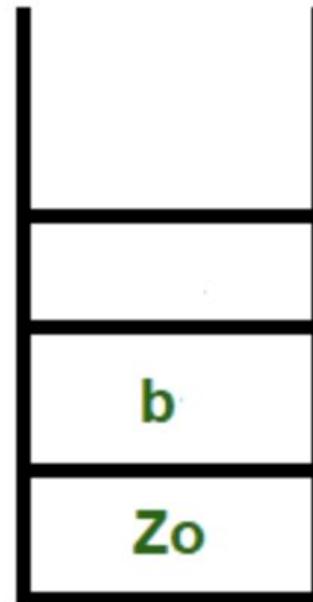
(1) PUSH - one symbol can be inserted into the stack at one time.  
 $\delta(q_i, a, z_0) = (q_j, az_0)$



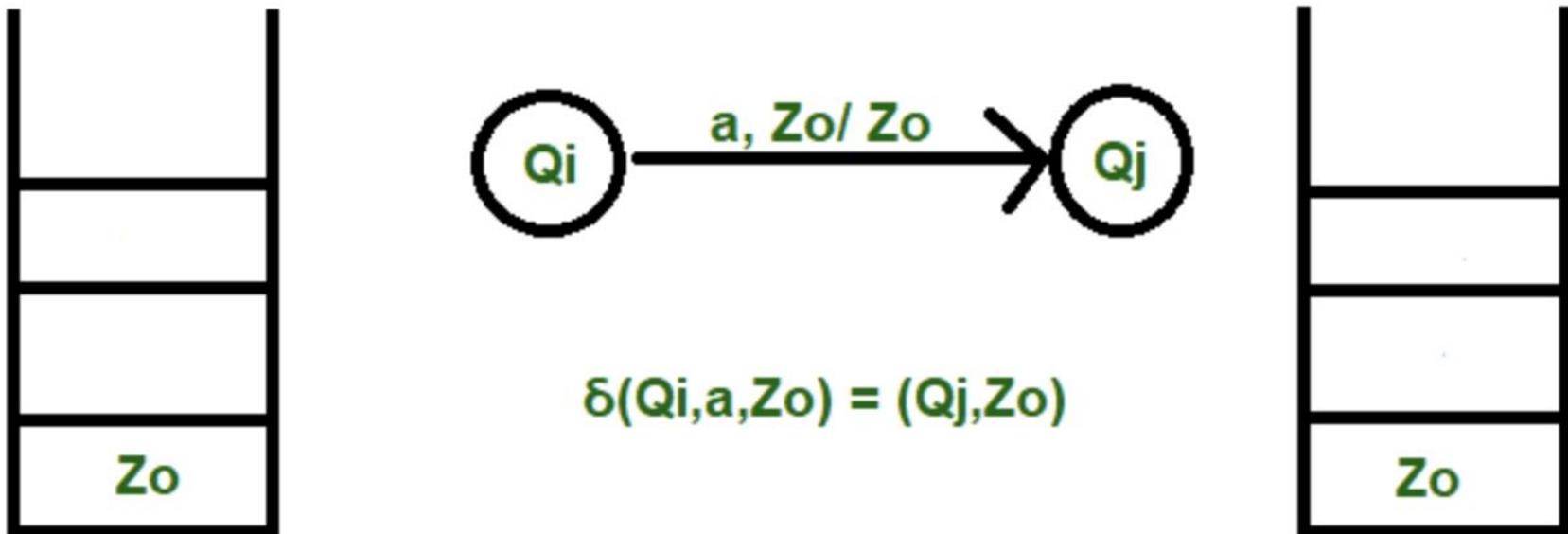
(2) POP - one symbol can be deleted from the stack at one time.



$$\delta(Q_i, a, c) = (Q_j, \epsilon)$$

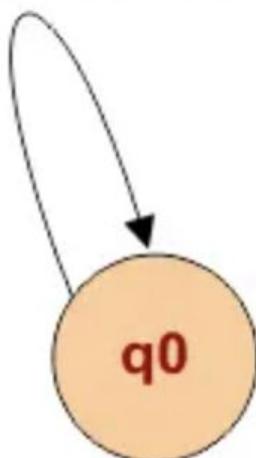


(3) SKIP - IT means no stack operation, status of the stack will remain same, before a after the operation



$$\delta(q_0, 0, Z_0) = (q_0, 0Z_0)$$

State Before Transition      Input Value      Top of Stack Before Transition      State After Transition      Present Value in Stack after Transition



Pushdown Automata

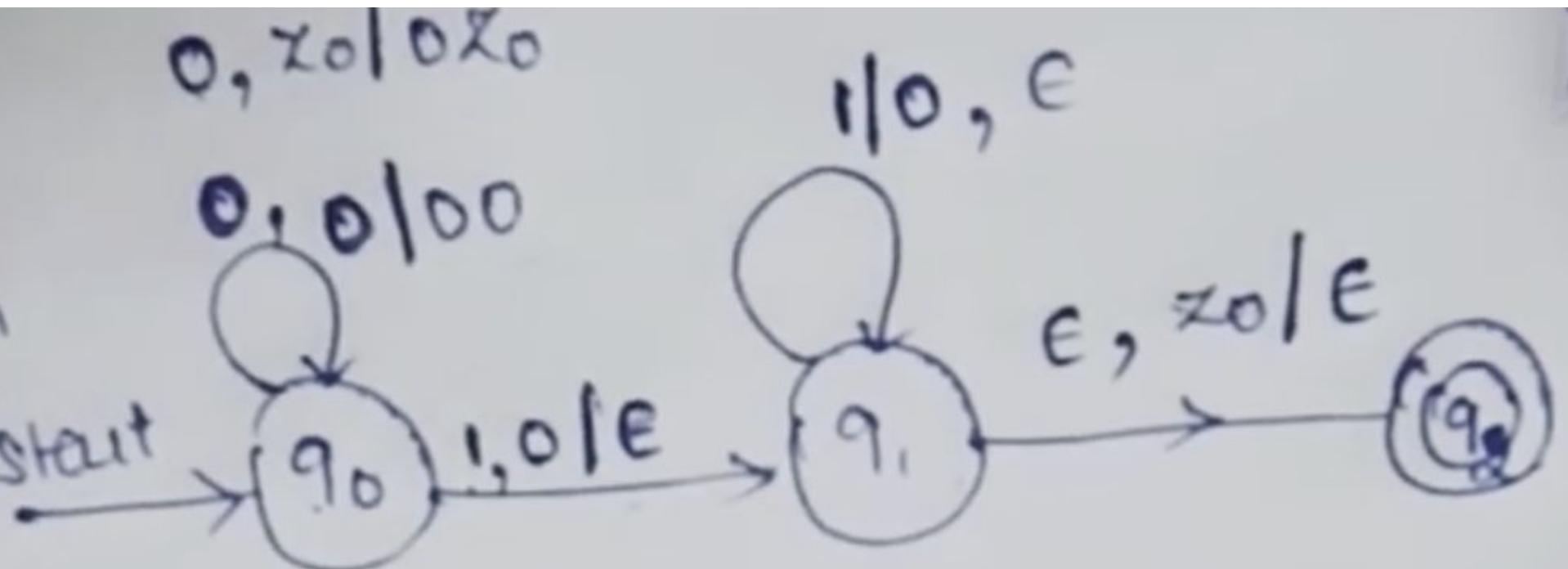
# Construct a PDA for the language $\{ 0^n 1^n \mid n \geq 1 \}$

To construct a PDA accepting by empty stack for the language  $L = \{0^n 1^n \mid n \geq 1\}$ , we can follow these steps:

1. Push a special symbol  $Z_0$  (not in the input alphabet) to the stack to mark the bottom of the stack.
2. Read the input symbol 0 and push it to the stack.
3. Repeat step 2 until all 0s are read.
4. Read the input symbol 1 and pop the topmost symbol (which should be 0).
5. Repeat step 4 until all 1s are read.
6. Check if the stack is empty/contains only  $Z_0$  at this point. If yes, accept the input.

Construct a PDA for the language  $\{ 0^n 1^n \mid n \geq 1 \}$

Transition Diagram



# Construct a PDA for the language $\{ 0^n 1^n \mid n \geq 1 \}$

Check the IP string:  
 $w = 0011$

Instantaneous Description (ID)

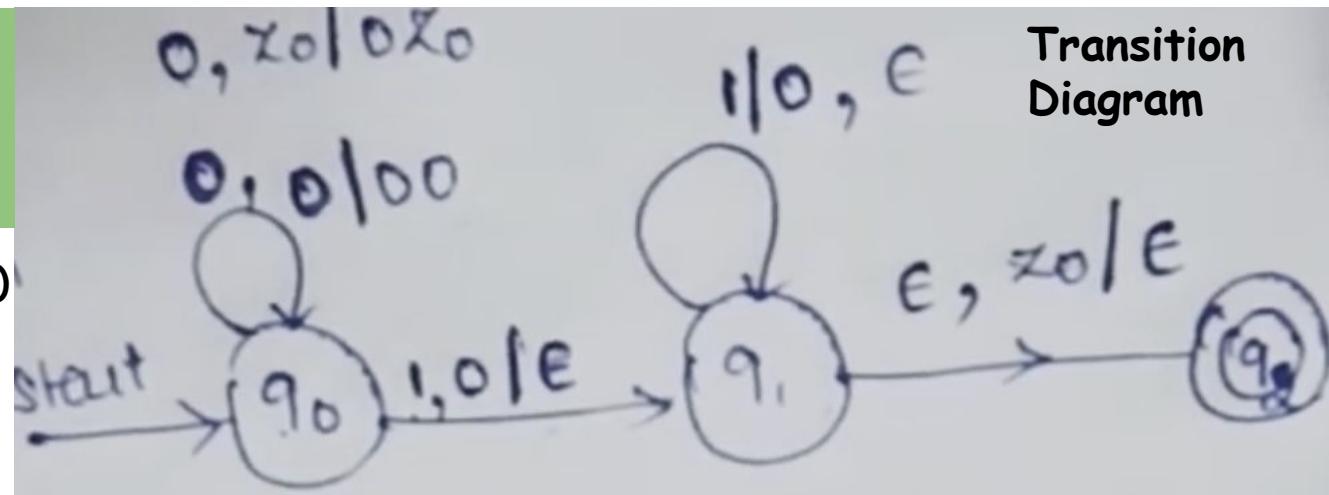
$$\delta(q_0, 0, z_0) = (q_0, 0z_0)$$

$$\delta(q_0, 0, 0) = (q_0, 00)$$

$$\delta(q_0, 1, 0) = (q_1, \epsilon)$$

$$\delta(q_1, 1, 0) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$$



Transition function  $\delta$  of a (PDA)

$$\vdash (q_0, 11, 00z_0)$$

$$\vdash (q_1, 1, 0z_0)$$

$$(q_0, w, z_0) \vdash (q_0, 0011, z_0)$$

$$\vdash (q_1, \epsilon, z_0)$$

$$\vdash (q_0, 011, 0z_0)$$

$$\vdash (q_2, \epsilon)$$

Construct a PDA for the language  $\{ a^n b^n \mid n \geq 1 \}$

Construct a PDA for the language  $\{ a^n b^n \mid n \geq 1 \}$

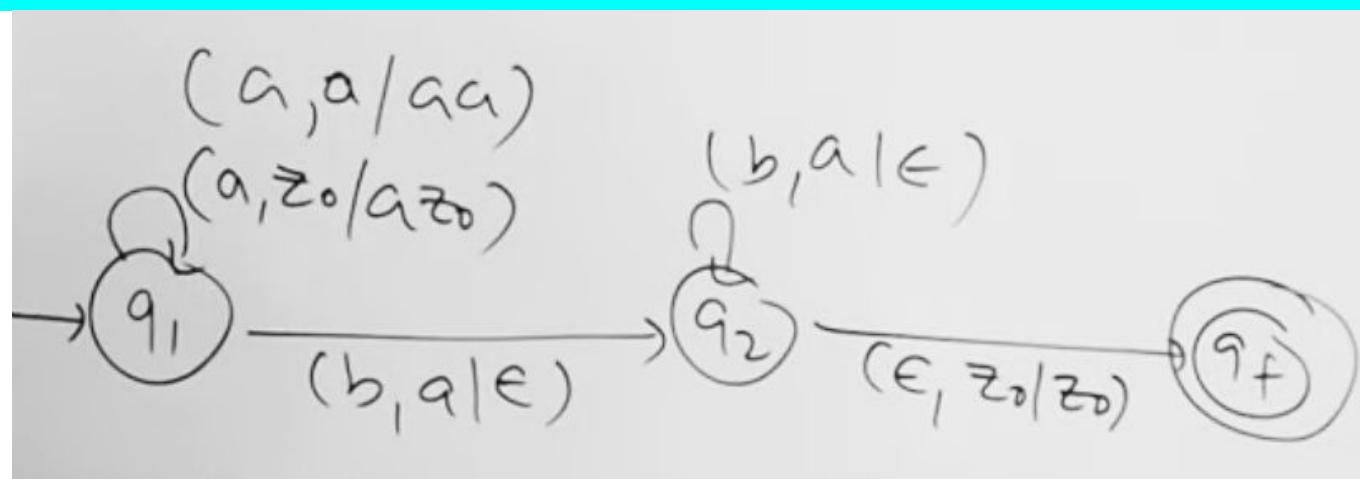
$$\delta(q_0, a, z_0) = (q_0, az_0)$$

$$\delta(q_0, a, a) = (q_0, aa)$$

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$$



$$(q_0, w, z_0) \vdash (q_0, aabb, z_0) \quad \vdash (q_1, b, az_0)$$

Check the  
IP string:  
 $w=aabb$

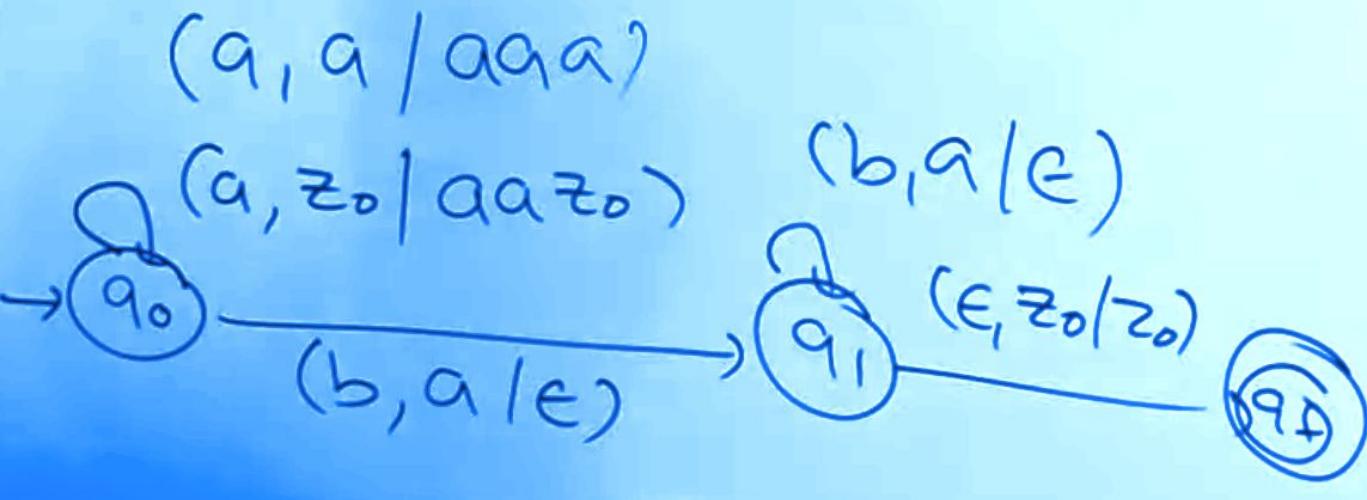
$$\vdash (q_0, abb, az_0)$$

$$\vdash (q_1, \epsilon, z_0)$$

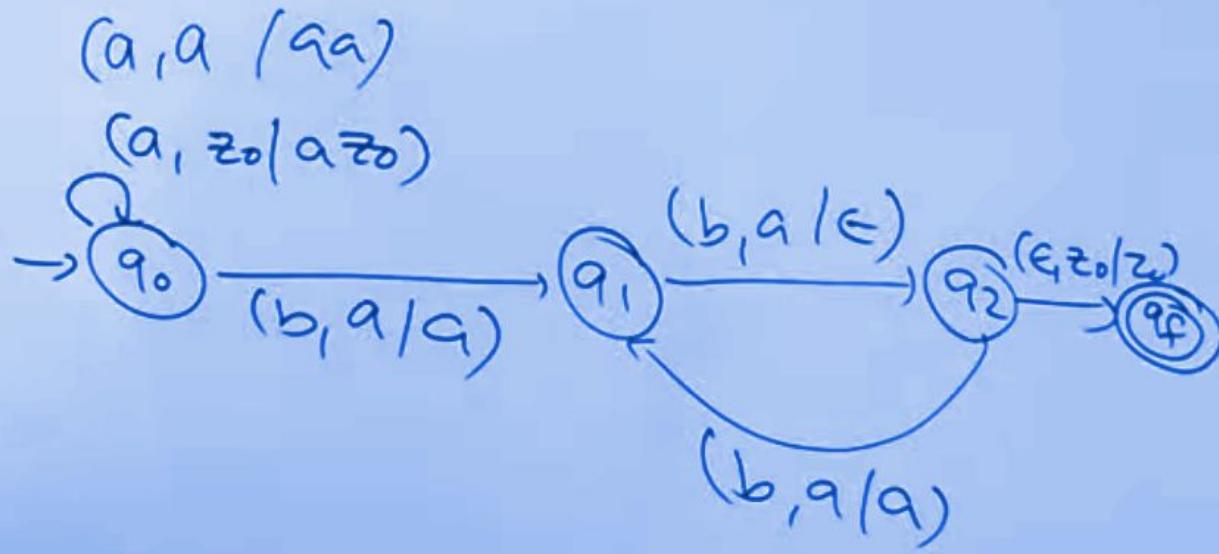
$$\vdash (q_0, bb, aa z_0)$$

$$\vdash (q_2, \epsilon)$$

Construct a PDA for the language  $\{ a^n b^{2n} \mid n \geq 1 \}$

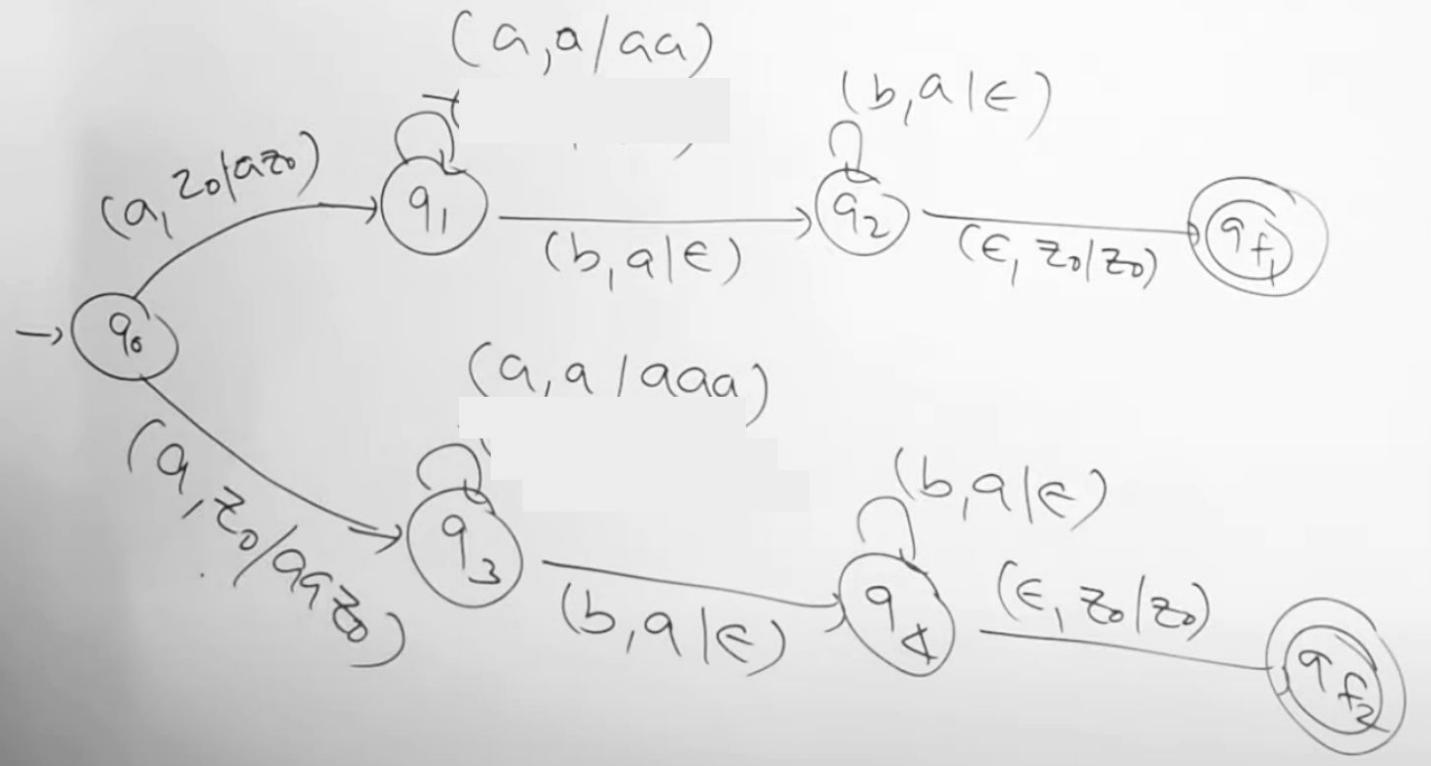


Construct a PDA for the language  $\{ a^n b^{2n} \mid n \geq 1 \}$



Construct a PDA for the language  $\{a^n b^n \cup a^n b^{2n} \mid n \geq 1\}$

Construct a PDA for the language  $\{a^n b^n \cup a^n b^{2n} \mid n \geq 1\}$



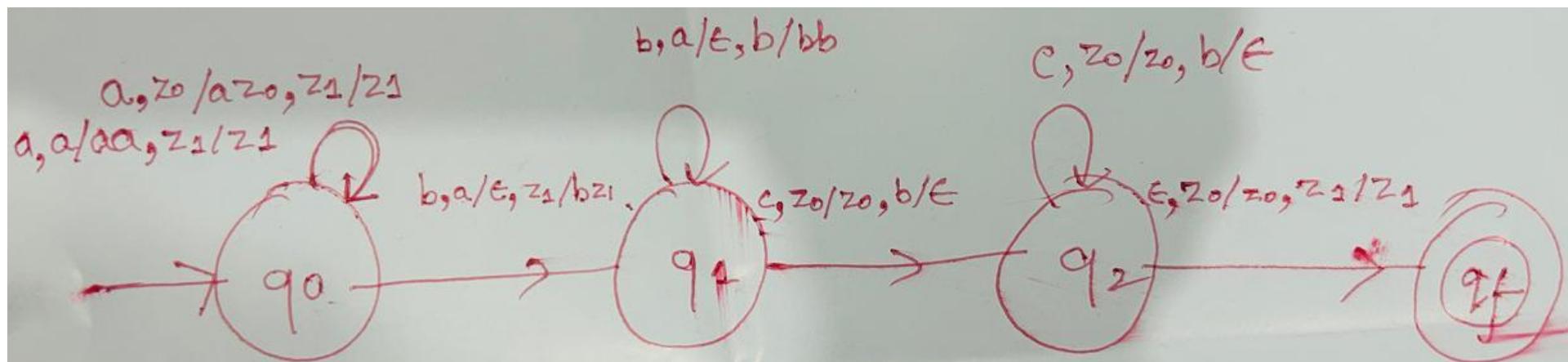
# Two stack PDA for $L = a^n b^n c^n \mid n \geq 1$

**Push 'a' into Stack 1:** For every 'a' in the input, push it onto the first stack.

**Pop 'a' and Push 'b' into Stack 2:** When 'b' appears, pop 'a' from Stack 1 and push 'b' onto Stack 2.

**Pop 'b' for each 'c':** When 'c' appears, pop 'b' from Stack 2.

**Accept if Both Stacks are Empty:** If the input is fully read and both stacks are empty, the string is accepted.



# Two stack PDA

$$L = a^n b^n c^n d^n \mid n \geq 1$$

**Push 'a' into Stack 1:** For each 'a' in the input, push it onto Stack 1.

**Pop 'a' and Push 'b' into Stack 2:** When 'b' appears, pop 'a' from Stack 1 and push 'b' onto Stack 2.

**Pop 'b' and Push 'c' into Stack 1:** When 'c' appears, pop 'b' from Stack 2 and push 'c' onto Stack 1.

**Pop 'c' for each 'd':** When 'd' appears, pop 'c' from Stack 1.

**Accept if Both Stacks are Empty:** If the input is fully read and both stacks are empty, the string is accepted.

# PDA for $wcw^R w \in (a,b)^*$

STACK Transition Function

$$\delta(q_0, a, Z) = (q_0, aZ)$$

$$\delta(q_0, a, a) = (q_0, aa)$$

$$\delta(q_0, b, Z) = (q_0, bZ)$$

$$\delta(q_0, b, b) = (q_0, bb)$$

$$\delta(q_0, a, b) = (q_0, ab)$$

$$\delta(q_0, b, a) = (q_0, ba)$$

// Decision step

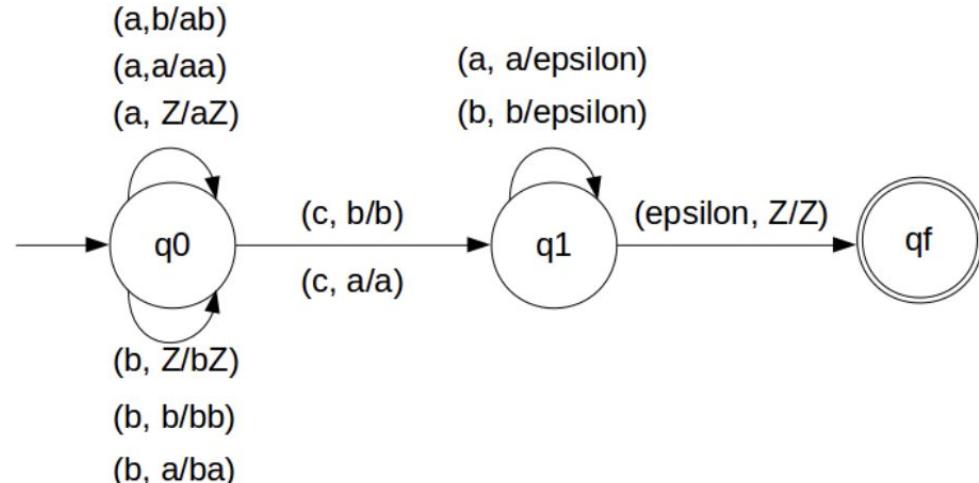
$$\delta(q_0, c, a) = (q_1, a)$$

$$\delta(q_0, c, b) = (q_1, b)$$

$$\delta(q_1, b, b) = (q_1, \epsilon)$$

$$\delta(q_1, a, a) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, Z) = (q_f, Z)$$



# Example : Process the input string: "abcbba" for the above DPDA

Scan string from left to right

First input is 'a' and follow the rule:

on input 'a' and STACK alphabet Z, push the two 'a's into STACK as : (a,Z/aZ) and state will be q0

Second input is 'b' and so follow the rule:

on input 'b' and STACK alphabet 'a', push the 'b' into STACK as : (b,a/ba) and state will be q0

Third input is 'b' and so follow the rule:

on input 'b' and STACK alphabet 'b', push the 'b' into STACK as : (b,b/bb) and state will be q0

Fourth input is 'c' and so follow the rule:

on input 'c' and STACK alphabet 'a' or 'b' and state q0, do nothing as : (c,b/b) and state will be q1

Fifth input is 'b' and so follow the rule:

on input 'b' and STACK alphabet 'b' (state is q1), pop one 'b' from STACK as : (b,b/ε) and state will be q1

Sixth input is 'b' and so follow the rule:

on input 'b' and STACK alphabet 'b' (state is q1), pop one 'b' from STACK as : (b,b/ε) and state will be q1

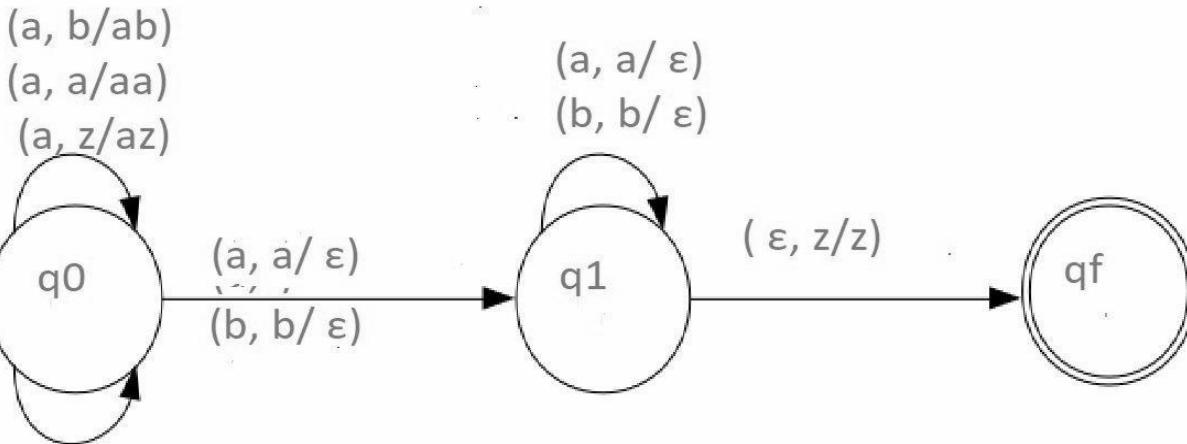
Seventh input is 'a' and so follow the rule:

on input 'a' and STACK alphabet 'a' and state q1, pop one 'a' from STACK as : (a,a/ε) and state will remain q1

We reached end of the string, so follow the rule:

on input ε and STACK alphabet Z, go to final state(qf) as : (ε, Z/Z)

# NPDA for $wwR \ w \in (\alpha, \beta)^*$ (EVEN PALINDROME)



Eg:  $w=ab$

# DPDA Vs NPDA

It is less powerful than  
NPDA.

Example:-

It is possible to convert  
every DPDA to a  
corresponding NPDA.

The language accepted by  
DPDA is a subset of the  
language accepted by  
NDPA.

It is more powerful than  
DPDA.

It is not possible to convert  
every NPDA to a  
corresponding DPDA.

The language accepted by  
NPDA is not a subset of the  
language accepted by  
DPDA.

There is only one state  
transition from one state to  
another state for an input  
symbol.

There may or maynot be  
more than one state  
transition from one state to  
another state for same  
input symbol.

# Pumping Lemma for CFLs

1. **Purpose:** Used to prove that a language is **not** context-free by showing that no matter how a string is divided, it cannot be pumped while staying in the language.
2. **Existence of a Constant p:** There exists a pumping length  $p$  (depends on the language) such that any string  $w$  with  $|w| \geq p$  can be split as:  $w = uvwxy$
3. **Contradiction Approach:** To prove  $L$  is **not** context-free:
  - Assume  $L$  is context-free.
  - Pick a string  $w$  with  $|w| \geq p$   $uv^iwx^iy \notin L$
  - Show that for every possible split, at least one pumped version
  - This contradiction proves  $L$  is **not** context-free.

# Pumping Lemma for CFLs

Let  $L$  be a context-free language, then there is a constant  $n$  which depends only on language  $L$ , such that there exists a string  $z \in L$  and  $|z| \geq n$ , where

$$z = uvwxy$$

## Conditions

1.  $|vwx| \leq n$
2.  $|vx| \geq 1$
3. For all  $i \geq 0$ ,  $uv^iwx^i y \in L$

## Example 1

$L = \{a^n b^n c^n \mid n \geq 0\}$  is not a CFL.

By assumption, the Pumping Lemma for CFLs must hold.

Let:

$$w = a^3 b^3 c^3 = aaabbcc$$

where  $n = 3$ .

We assume a split such that:

$$w = (a)(aa)(b)(bb)(ccc)$$

$uv^iwx^iy \notin L$  Where: v=aa, x=bb, and u,w,y are other segments

Let  $i = 2$  (pump up once). This results in:

$$uv^2wx^2y = a^5b^5c^2$$

which does not belong to  $L$  because the number of  $as$ ,  $bs$ , and  $cs$  is no longer equal.

Since the pumped string is not in  $L$ , the Pumping Lemma fails, proving that  $L$  is not a context-free language.

Thus,  $L = \{a^n b^n c^n\}$  is not a CFL.

## Example2

Show that  $L = \{a^i b^j \mid j = i^2\}$  is not a context-free language.

## Example2

Show that  $L = \{a^i b^j \mid j = i^2\}$  is not a context-free language.

Suppose, for contradiction, that  $L$  is a context-free language.

$$L = \{ab, aabb, aaab^9, \dots\}$$

Let  $z = aabb$

Now, divide  $z$  into 5 parts such that  $z = uvwxy$ , satisfying:

- $|uvw| \leq n$
- $|vx| \geq 1$

Let:

$$z = \underline{aabbb}$$

u v w x y

Where:  $u = a, v = a, w = b, x = bb, y = b$ .

Since  $|vx| \geq 1$  and  $|vx| \leq 4$ , we select  $|vx| = 2$ .

If we pump  $i = 2$ , then the resulting string is:

$$uv^2wx^2y = aa^2bbb^2b$$

$= aaab^6$ , which does not belong to  $L$  because for  $i = 3$ ,  $j$  should be 9, but here it is 6.

Thus, the Pumping Lemma fails, proving that  $L$  is not a context-free language.

# **Closure properties of Context-Free Language**

Context-Free languages are **closed** under:

- a. Union
- b. Concatenation
- c. Kleene Closure

Not closed under:

- a. Intersection
- b. Complementation

# Union

**Union:** Let  $L_1$  &  $L_2$  be CFLs.

$L_1 \cup L_2$  is also a CFL.

$$L_1 = \{a^n \mid n \geq 0\}$$

$$L_2 = \{b^n \mid n \geq 0\}$$

$$L_1 = \{\epsilon, a, aa, \dots\}$$

$$L_2 = \{\epsilon, b, bb, \dots\}$$

Grammar rules:

$$S_1 \rightarrow aS_1 \mid \epsilon$$

$$S_2 \rightarrow bS_2 \mid \epsilon$$

$$L_3 = L_1 \cup L_2$$

$$L_3 = \{\epsilon, a, aa, b, bb, \dots\}$$

Final grammar:

$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow aS_1 \mid \epsilon$$

$$S_2 \rightarrow bS_2 \mid \epsilon$$

## Union Example 2:

Given:

$$L_1 = \{a^n b^n \mid n \geq 1\} = \{ab, aabb, aaabbb, aaaabbbb, \dots\}$$

$$L_2 = \{c^n d^n \mid n \geq 1\} = \{cd, ccdd, cccddd, ccccdddd, \dots\}$$

Since  $L_3 = L_1 \cup L_2$ , the language  $L_3$  contains all strings that belong to either  $L_1$  or  $L_2$ .

$$L_3 = L_1 \cup L_2 = \{ab, aabb, aaabbb, aaaabbbb, \dots\} \cup \{cd, ccdd, cccddd, ccccdddd, \dots\}$$

So, some example strings in  $L_3$  are:

$$L_3 = \{ab, aabb, aaabbb, aaaabbbb, \dots, cd, ccdd, cccddd, ccccdddd, \dots\}$$

$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow aS_1b \mid \epsilon$$

$$S_2 \rightarrow cS_2d \mid \epsilon$$

# Concatenation

Given CFLs:

- $L_1 = \{a^n \mid n \geq 0\} = \{\epsilon, a, aa, aaa, \dots\}$
- $L_2 = \{b^n \mid n \geq 0\} = \{\epsilon, b, bb, bbb, \dots\}$

Concatenation:

- $L_3 = L_1 \cdot L_2$  consists of all strings where a sequence of 'a's is followed by a sequence of 'b's.
- $L_3 = \{a^m b^n \mid m, n \geq 0\} = \{\epsilon, b, bb, a, ab, abb, aa, aab, aabb, \dots\}$

Grammar Rules:

1.  $S_1 \rightarrow aS_1 \mid \epsilon$  (generates  $L_1$ )
2.  $S_2 \rightarrow bS_2 \mid \epsilon$  (generates  $L_2$ )
3.  $S \rightarrow S_1S_2$  (concatenates  $L_1$  and  $L_2$ )

Final Grammar:

$$S \rightarrow S_1S_2$$

$$S_1 \rightarrow aS_1 \mid \epsilon$$

$$S_2 \rightarrow bS_2 \mid \epsilon$$

## Concatenation Example 2

Languages:

$$L_1 = \{a^n b^n \mid n \geq 1\} \Rightarrow S_1 \rightarrow aS_1b \mid \epsilon$$

$$L_2 = \{c^n d^n \mid n \geq 1\} \Rightarrow S_2 \rightarrow cS_2d \mid \epsilon$$

The concatenation of  $L_1$  and  $L_2$  is:

$$L_3 = L_1 \cdot L_2 = \{a^m b^m c^n d^n \mid m \geq 1, n \geq 1\}$$

So, the language is:

$$L_3 = \{a^m b^m c^n d^n \mid m \geq 1, n \geq 1\}$$

$$S \rightarrow S_1 S_2$$

$$S_1 \rightarrow aS_1b \mid \epsilon$$

$$S_2 \rightarrow cS_2d \mid \epsilon$$

# Kleene Closure

$$L = \{a^n b^n \mid n \geq 0\},$$

$L_1 = L^*$  is also a CFL

$$S \rightarrow SS_1 \mid \epsilon$$

$$S_1 \rightarrow aS_1b \mid \epsilon$$

-

# Intersection

$$L_1 = a^n b^n c^m \mid n, m \geq 0$$

$$L_2 = a^m b^n c^n \mid m, n \geq 0$$

Intersection:

$$L_1 \cap L_2 = a^n b^n c^n \mid n \geq 0$$

Strings in  $L_1$ :

$\epsilon, ab, abc, aabb, aabbc, aabbcc, aaaabbb, a^3b^3c, a^3b^3c^2, a^3b^3c^3, \dots$



Strings in  $L_2$ :

$\epsilon, bc, abc, abcc, abbcc, aabbcc, b^3c^3, ab^3c^3, a^2b^2c^2, a^2b^3c^3, a^3b^3c^3, \dots$



The intersection  $L_1 \cap L_2$  gives  $a^n b^n c^n$  where  $n \geq 0$ . This language is known to be a non-context-free language (non-CFL). Since  $L_1$  and  $L_2$  are context-free languages (CFLs) but their intersection is not a CFL, this shows that the intersection of two CFLs is **not necessarily a CFL**. Therefore, by this counterexample, we prove that context-free languages (CFLs) are not closed under intersection.

# Complement

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

$$= \overline{\text{CFL} \cup \text{CFL}}$$

$$= \overline{\text{CFL}} = \text{CFL}$$

Which is invalid,  
therefore , not closed  
under complement

# LL(k) Grammar

If k=1, then we have, **LL(1)** Grammar

- **1st L indicates** → Reading input string from left to right.
- **2nd L indicates** → Leftmost derivation
- **1 indicates** → Looking ahead terminal symbol in the input string.
- Used for Top Down Parsing

Example: Consider the Grammar:

& the string,

$$S \rightarrow aA \mid bB$$

w=aaabd

$$A \rightarrow aB \mid cB$$

Determine

Whether the

$$B \rightarrow bC \mid aC$$

given grammar

$$C \rightarrow bD$$

is LL(1) or not.

$$D \rightarrow d$$

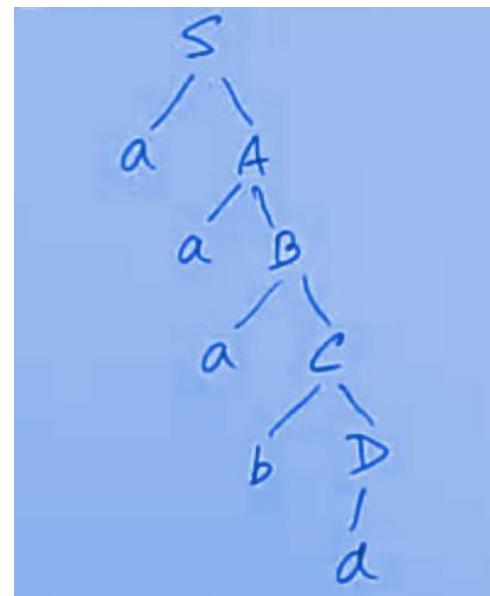
Implicit Parse Tree

$$S \Rightarrow aA \Rightarrow aaB \Rightarrow aaac$$

$$\Rightarrow aaabD \Rightarrow aaabd$$

$$\text{So } S \xrightarrow{*} aaabd$$

Explicit Parse Tree



**Example2:** Consider the following grammar and the string w=abd. Determine whether the given grammar is LL(1) or not

$$S \rightarrow abB \mid aaA$$

$$B \rightarrow d$$

$$A \rightarrow c \mid d$$

$$S \Rightarrow abB \Rightarrow abd$$

# LL(k) Grammar

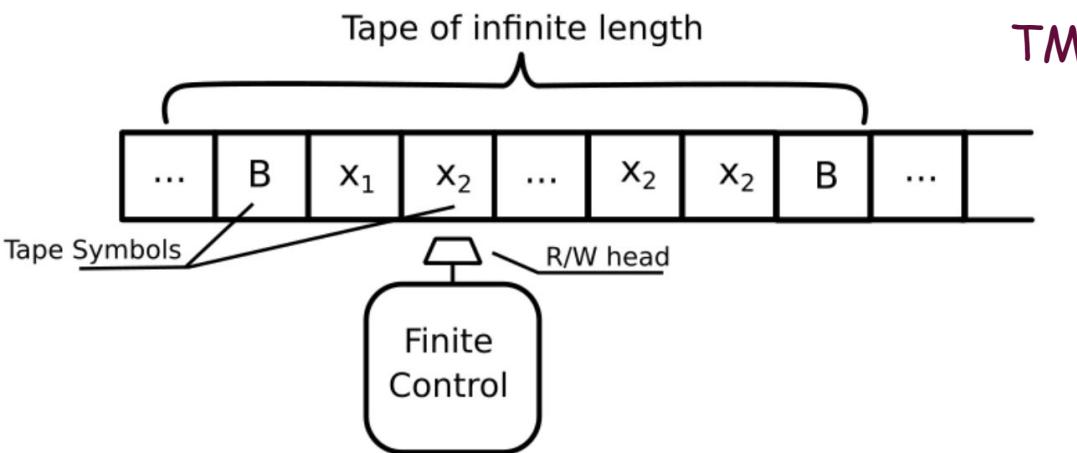
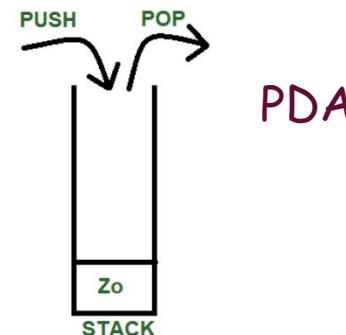
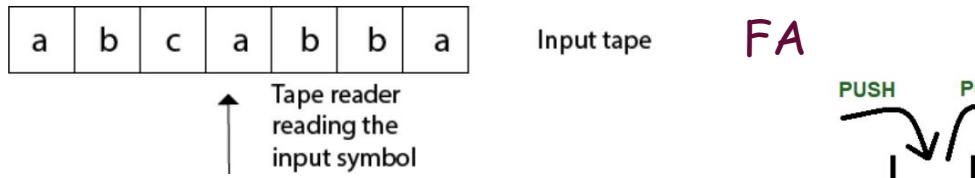
$\text{LL}(k) \subset \text{LL}(k+1)$

So  $\text{LL}(1) \Rightarrow \text{LL}(2) \Rightarrow \text{LL}(3) \Rightarrow \dots \Rightarrow \text{LL}(k)$

But if  $\text{LL}'(k) \Rightarrow \dots \Rightarrow \text{LL}'(3) \Rightarrow \text{LL}'(2) \Rightarrow \text{LL}'(1)$

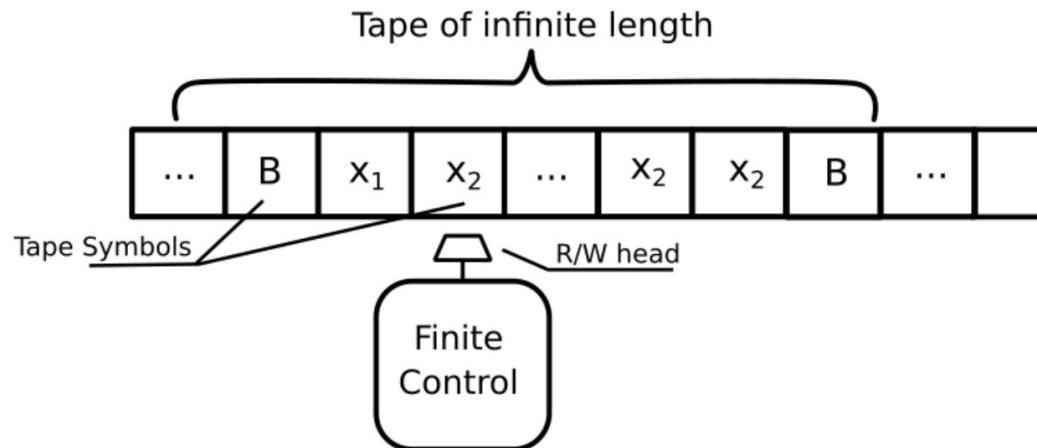
LL(k)	LL(k+1)	Inference
True	True	True
True	False	False
False	True	True
False	False	True

# Introduction to Turing Machine



# TMs are able to do any of the following Operations on Tape

- 1) Read a symbol from the tape.
- 2) Write a symbol to the tape.
- 3) Move the tape head, one step LEFT.
- 4) Move the tape head, one step RIGHT.



# Components of Turing Machine

## TAPE:

A 2-way infinite tape, divided into cells. Each cell is capable of holding only one tape symbol.

**Purpose:** Acts as the machine's memory, allowing it to read from and write to the tape.

## TAPE HEAD:

A read/write head that moves along the tape. It can read the symbol in the current cell, write a new symbol, and move left or right.

**Purpose:** Facilitates interaction with the tape by reading symbols and updating them.

## FINITE CONTROL UNIT:

The finite control unit is the part of the Turing Machine that contains the transition function and manages the state transitions.

**Purpose:** It acts as the decision-making component that directs the machine's actions based on its current state and the input symbol it encounters

# Formal Definition

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

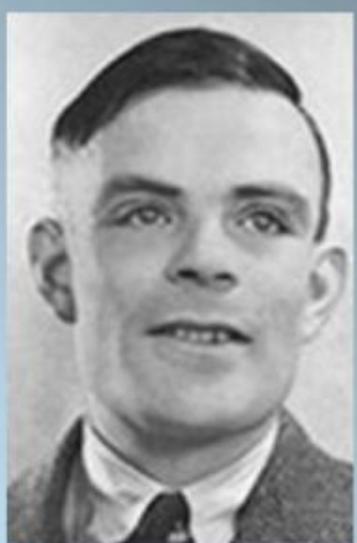
- $Q$ : The finite set of **states**
- $\Sigma$ : The finite set of **input symbols**.
- $\Gamma$ : The complete set of **tape symbols**,  $\Sigma$  is always a subset of  $\Gamma$ .
- $\delta$ : The **transition function**. Defines the next state assumed by the machine, the symbol to be written in the tape, and the direction which the head will moves.  $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ 

The transition  $\delta(q_0, a) = (q_1, X, R)$ , means that at  $q_0$  state if 'a' occurs then replace it with 'X', change the state to  $q_1$  and move to right.
- $q_0$ : The **start state**, a member of  $Q$ , where the finite control is found initially.
- $B$ : the **blank symbol**. This symbol is in  $\Gamma$  but not in  $\Sigma$ .
- $F$ : the set of final or **accepting states**, a subset of  $Q$ .

# Modifications of Turing Machine

1. Multi-tape Turing Machine
2. Turing Machine with stay option d:  $Q \times \Gamma \rightarrow Q \times \Gamma \times (L/R/S)$
3. Multi-dimensional Turing Machine d:  $Q \times \Gamma \rightarrow Q \times \Gamma \times (L/R/U/D)$
4. Turing Machine with one way infinite tape (Semi infinite tape)
5. Multi Read/Write head points
6. Multi-head Turing Machine
7. Offline TM: have a restriction that input can't be changed (2 tapes)
8. Jumping TM: where the tape head can jump to non-adjacent positions on the tape
9. Non erasing TM: where input can not be converted to blank
10. Always writing TM
11. UTM
12. Non-Deterministic Turing Machine
13. TM without Writing Capacity (FA)
14. TM with Tape used as Stack
15. TM with finite tape
16. TM with Input Size Tape (LBA)

# *Church's Thesis*



Alan Turing  
(1912 – 1954)



Alonzo Church  
(1903-1995)

# Church's Thesis

**According to the Church-Turing thesis:**

- Anything that can be done on any existing digital computer can also be done by a Turing machine/ Everything that can be computed, can be computed by a Turing machine / What could naturally be called as an effective procedure, can be realized by a turing machine.
- This thesis cannot be proven but is believed on the basis of the amount of evidence that supports it. No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a Turing machine program cannot be written.
- Computer scientists have devised several alternative models of computation, but none of them is more powerful than the Turing machine model.

**"Any real-world computation can be translated into an equivalent Turing machine computation."**

# Design a Turing machine for $a^n b^n \mid n \geq 1$

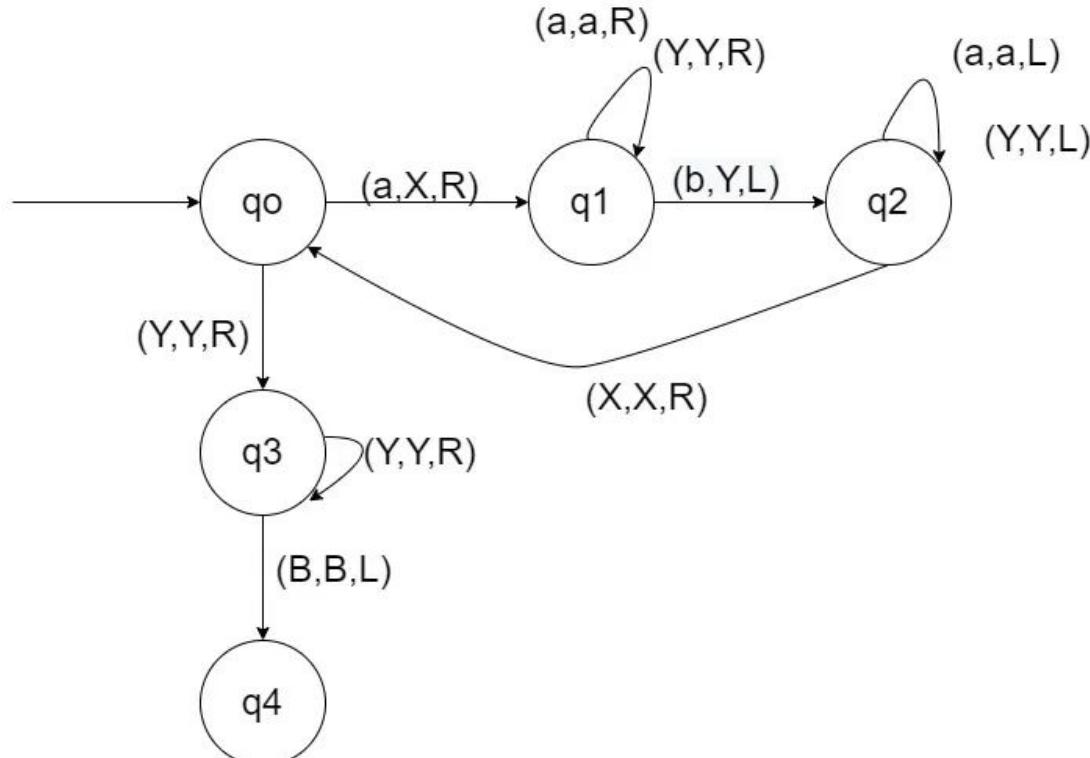
B	a	a	b	b	B
---	---	---	---	---	---

B	X	a	b	b	B
---	---	---	---	---	---

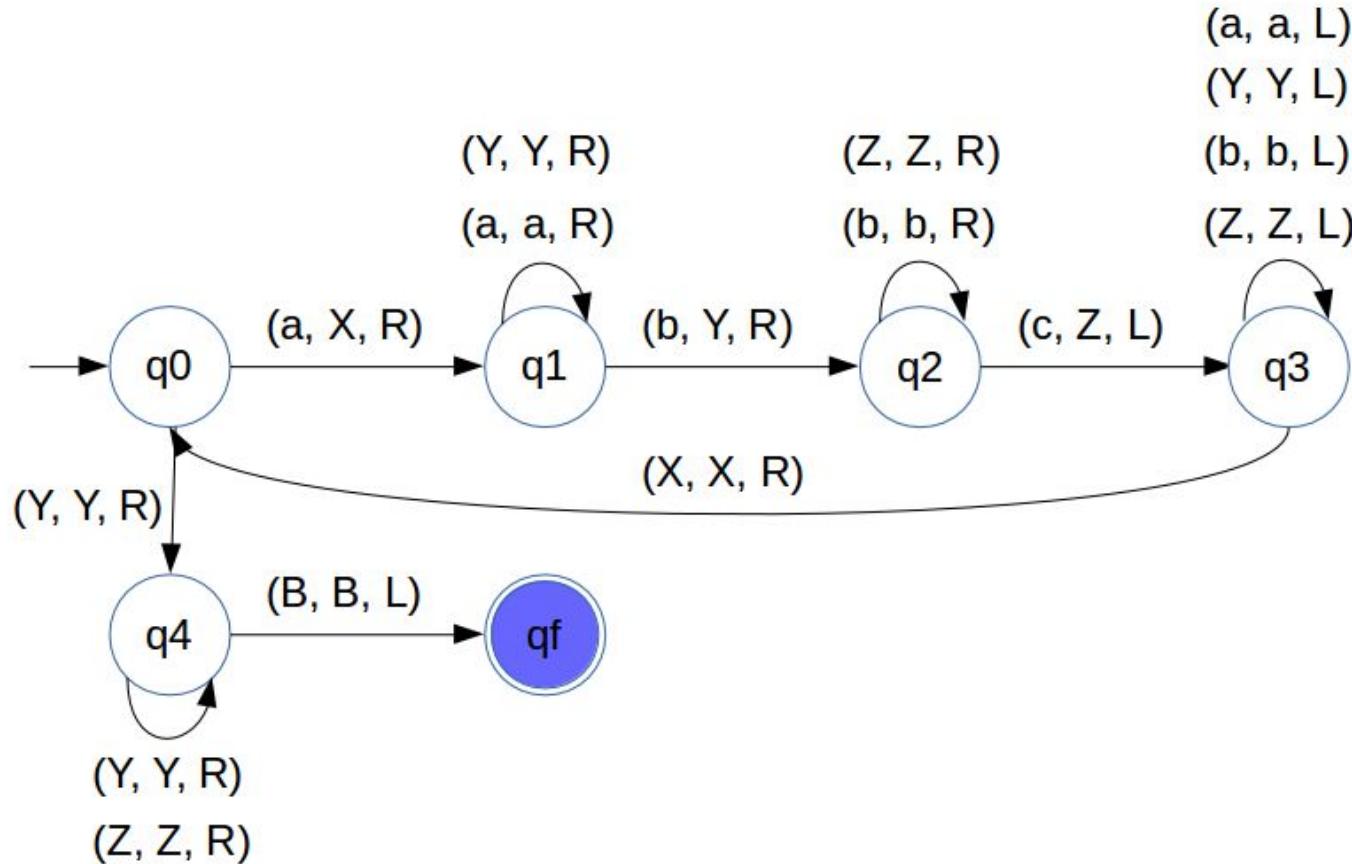
B	X	a	Y	b	B
---	---	---	---	---	---

B	X	X	Y	b	B
---	---	---	---	---	---

B	X	X	Y	Y	B
---	---	---	---	---	---



# Design a Turing machine for $a^n b^n c^n \mid n \geq 1$



## Turing machine for 2's Complement

Let M be the Turing Machine for above AIM, hence it can be define as  $M(Q, \Sigma, \Gamma, \delta, q_0, B, F)$  where

$Q$ : set of states:  $\{q_0, q_1, q_2\}$

$\Sigma$ : set of input symbols:  $\{0, 1\}$

$\Gamma$ : Set of Tape symbols:  $\{0, 1, B\}$

$q_0$ : initial state ( $q_0$ )

$B$ : Blank Symbol ( $B$ )

$F$ : set of Final states:  $\{\}$  [Note: Here, set of final states is null as here we have to design turing machine as enumerator.]

$\delta$ : Transition Function:

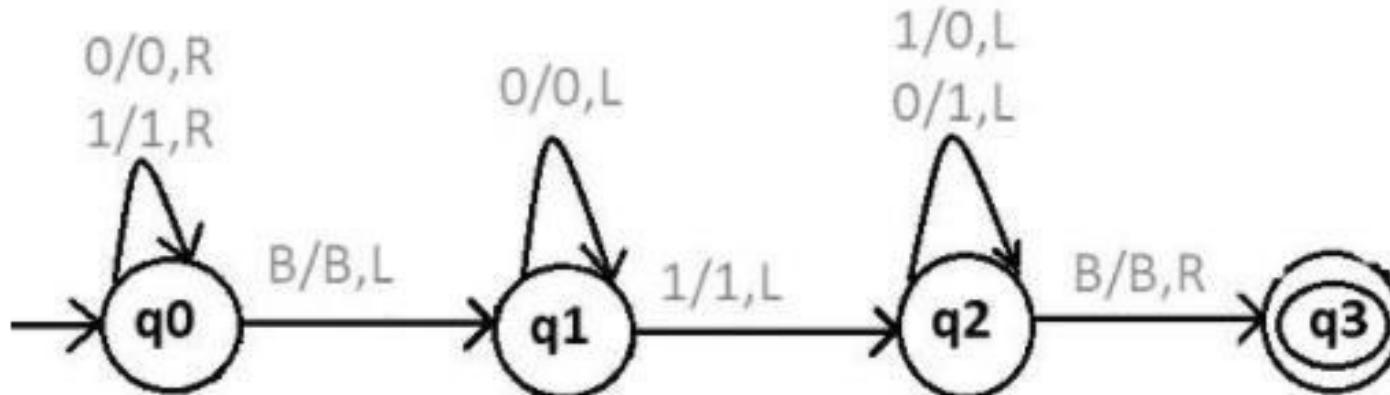
## Turing machine for 2's Complement

1. Move the input head at right direction, till it reaches to B. If B encounter then change state from  $q_0$  to  $q_1$ .
2. Now, start reading input symbol right to left one by one.
3. If the input symbol is 0, move the input head left and do the same for all 0 till 1 encounter.
4. If the input symbol is 1, move the input head left and change its state from  $q_1$  to  $q_2$ .
5. Now, if the input symbol is 0, make it 1 and if input symbol is 1 make it 0 till reach to the starting point.

## Transition Table

State	Input		
-	0	1	B
q0	(q0, 0, R)	(q0, 1, R)	(q1, B, L)
q1	(q1, 0, L)	(q2, 1, L)	
q2	(q2, 1, L)	(q2, 0, L)	(q3, B, R)
q3			

## Transition Diagram



Turing machine for  $L = \{w c w \mid w \in \{0, 1\}^*\}$ ?

# Turing machine for addition of two number in unary

# Turing machine for converting unary number into binary number

# Universal Turing Machine (UTM)

**Motivation:** We need a single m/c which can solve all type of problems

- ❖ A Universal Turing Machine is a reprogrammable Turing machine capable of simulating the behavior of any other Turing machine, effectively serving as a programmable computer that can execute any computable function.
- ❖ A standard Turing machine can simulate only a single computation problem, so if we want to solve another computation problem, then another Turing machine is required to be constructed, but in case of a UTM, the same UTM can be used to solve any computation problem.
- ❖ Power of a UTM is the same as the power of a standard TM as both solve the same set of computational problems. (also a digital computer)
- ❖ **Turing complete** means a system can perform any computation that a Turing machine can, given enough time and memory.

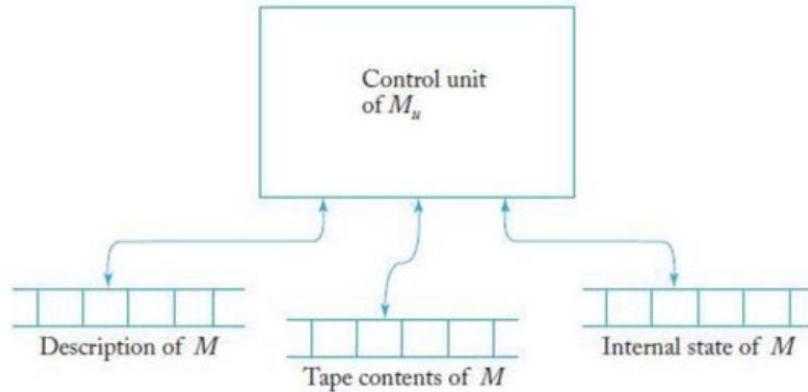
# Universal Turing Machine (UTM)

Input to UTM is description of TM () and string w.

It consists of 3 tapes:

1. 1st tape stores the binary encoding of TM.
2. 2nd tape stores string w.
3. 3rd tape stores the state of TM.

Action: Accept, Reject, Loop



# Linear Bounded Automaton (LBA)

An LBA is a restricted type of Turing Machine where the tape is bounded by the length of the input.  
LBAs are precisely the machines that recognize context-sensitive languages.  
LBA is more powerful than PDA but less powerful than unrestricted Turing Machines.

A linear bounded automaton can be defined as an 8-tuple  $(Q, X, \Sigma, q_0, M_L, M_R, \delta, F)$  where –

**Q** is a finite set of states

**X** is the tape alphabet

**$\Sigma$**  is the input alphabet

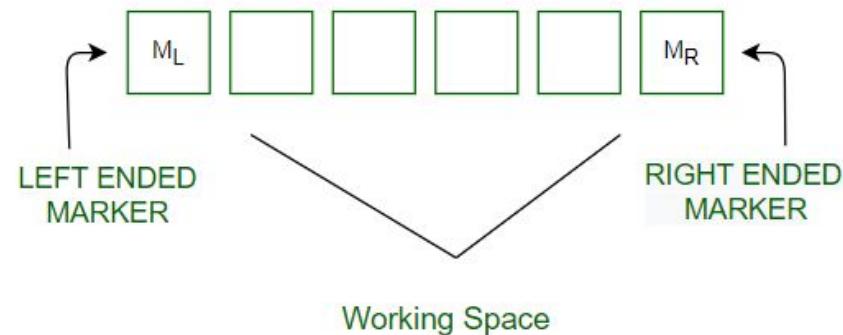
**$q_0$**  is the initial state

**$M_L$**  is the left end marker

**$M_R$**  is the right end marker where  $M_R \neq M_L$

**$\delta$**  is a transition function

**F** is the set of final states



# **Recursive & Recursive Enumerable Language**

## **Recursive Language (REC) {Turing Dec.}**

A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language. The TM will always halt in this case. REC languages are also called as Turing decidable languages. Recursive languages are not closed under following operations • Kleene closure • Homomorphism • Substitution

## **Recursive Enumerable (RE) or Type -0 Language {Turing Rec.}**

RE languages or type-0 languages are generated by type-0 grammars. An RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. RE languages are also called as Turing recognizable languages. RE are not closed under following operations • Complement • Set Difference

# Halting Problem

**Halt** : The state of Turing machine, where no transition is defined or required is known as Halt.

After taking an input string, there are three possibilities for Turing Machine, i.e. it may go to:

- **Final halt**
- **Non- Final halt**
- **Infinite loop**

In case of Final Halt, Machine halts on final state, and hence it shows that string is accepted. In Non-Final Halt, Machine halts on non- final state, and hence the string is rejected. If the machine goes to infinite loop after taking an input string, then we can't say whether the string is Accepted/Rejected.

# Halting Problem

The Halting Problem in Automata theory is a problem that asks whether a given computer program (Turing machine) will eventually stop running (halt) or continue to run forever when provided with a specific input.

**Problem:** Determine if a program halts on a given input.

The answer is no we cannot design a generalized algorithm which can appropriately say that given a program will ever halt or not?

The only way is to run the program and check whether it halts or not.

# What is the Halting Problem of a TM?

This is the classic **undecidable problem**:

**Given** a Turing Machine  $M$  and an input string  $w$ ,  
**Does**  $M$  eventually **halt** on  $w$ ?

Halting means:

- TM stops (accepts or rejects).
- Not looping forever.

Alan Turing proved: **No algorithm can decide** whether any TM halts on all inputs.

That's what we call **undecidable**.

# Turing Machine as Computer of Integer Functions

- An integer function is a function that accepts integers as input and produces an integer as output.
- Let an integer function be  $f(x,y) = x+y$ . We have to solve this function using Turing Machine. so , input tape must represent the input arguments.
- Let the number of '0' be used to represent the argument. But there must be a separator to differentiate the number of '0' for each argument. In Turing machine, blank space is used to separate the arguments on the tape.

# Undecidability

## # Chomsky Hierarchy

Class of language for which Turing machine will never halt for rejecting strings

Classes of language for which Turing machine will always halt in any case.

Type – 0

Recursive Enumerable Set

Type – 0

Recursive Set

Type – 1

Context Sensitive Grammar

Type – 2

Context Free Grammar

Type – 3

Regular Grammar

# Undecidability Problems

Undecidable problems are decision problems where no algorithm can be constructed to always provide a correct "yes" or "no"

Eg. Fermat's Last theorem is undecidable, which states that no three positive integers x, y, and z can satisfy the following equation for any integer value of n > 2.

$$x^n + y^n = z^n$$

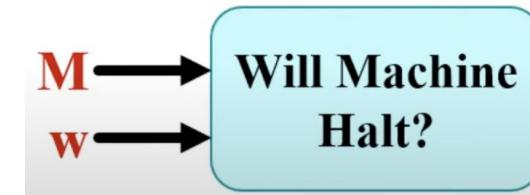
## 1. The Halting Problem:

**Question:**

Given a program and an input, will the program eventually halt (stop running) or run forever?

**Why it's undecidable:**

It's impossible to create an algorithm that can always correctly determine if any arbitrary program will halt. Alan Turing proved this in 1936, demonstrating the existence of undecidable problems.



## 2. The Post Correspondence Problem (PCP):

**Question:**

Given a list of pairs of strings (dominoes), can you arrange them in a way such that concatenating the top strings equals the concatenation of the bottom strings?

**Why it's undecidable:**

There's no algorithm that can always determine if such an arrangement exists.

# Post Correspondence Problem (PCP)

It is a popular undecidable problem that was introduced by Emil Leon Post in 1946.

The problem involves two lists of words over a common alphabet.

The challenge is to find a sequence of indices where the concatenation of the words in the first list, in that sequence, is equal to the concatenation of the words in the second list in the same sequence

Def 2: we have N number of **Dominos** (tiles). The aim is to arrange tiles in such order that string made by Numerators is same as string made by Denominators.

# Post Correspondence Problem

Q. Find whether the lists  $M = (ab, bab, bbaaa)$  and  $N = (a, ba, bab)$  have a post correspondence solution? For  $x_2 \ x_1 \ x_3 \Delta \ y_2 \ y_1 \ y_3$

Q.  $A = \{1, 110, 0111\} \cdot B = \{111, 001, 11\}$

## Post Correspondence Problem

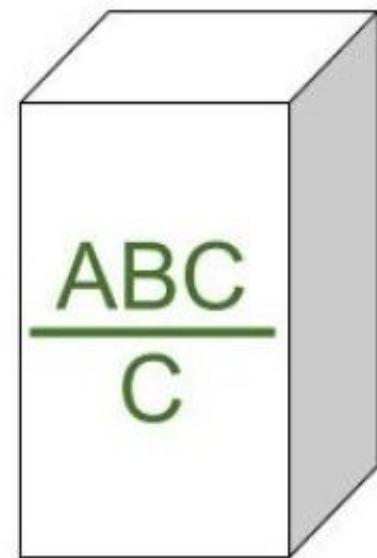
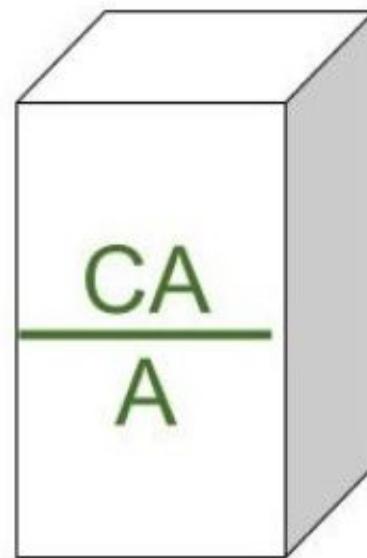
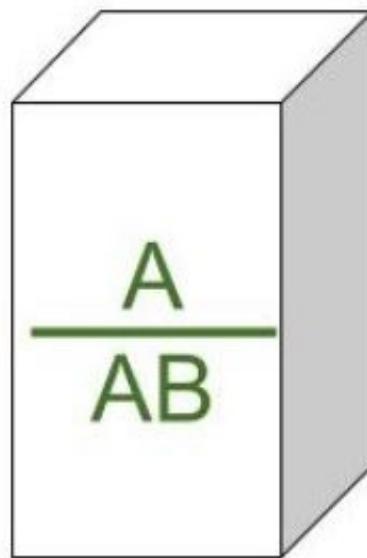
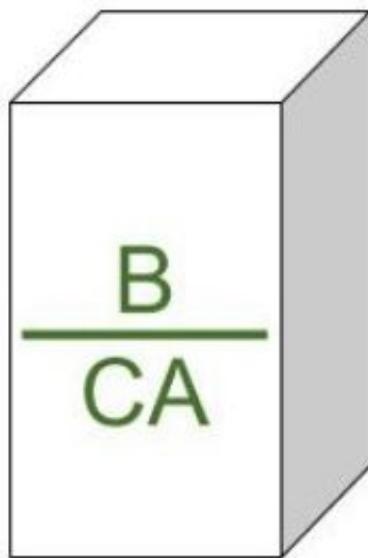
Find whether the lists  $M = (\text{abb}, \text{aa}, \text{aaa})$  and  $N = (\text{bba}, \text{aaaa}, \text{aa})$  have a post correspondence solution.

# PCP Examples

	A	B
1	1	111
2	10111	10
3	10	0

2113

# PCP Examples



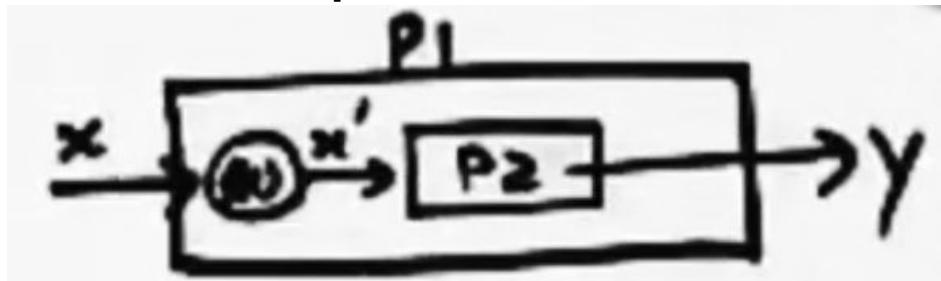
1	100	1
2	0	100
3	1	00

**Check does PCP with two lists X= (0101, 000111, 001, 10, 01, 00) and Y= (0101000, 11, 1001, 100, 10, 0) have a solution?**

# Reducibility

Reducibility means to **transfer knowledge** about one problem to another

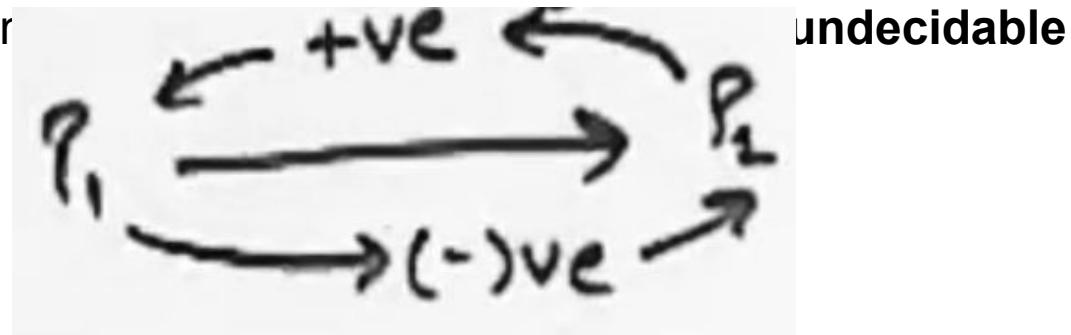
Formal definition: If there exists a **computable function f** such that:  $x \in P_1 \Leftrightarrow f(x) \in P_2$



Then,  $P_1 \leq P_2$  or  $P_1 \leq \square P_2$ , ( $P_1$  is reducible to  $P_2$ )

If we know  **$P_2$  is decidable**, and  **$P_1$  reduces to  $P_2$** , then  **$P_1$  is also decidable**.

If we know  **$P_1$  is undecidable**, ar



Let **+ve  $\rightarrow$  Decidable**

**-ve  $\rightarrow$  Undecidable**, them

# Reducibility

If  $P_1$  is decidable, then  $P_2$  \_\_\_\_\_?

If  $P_1$  is undecidable, then  $P_2$  \_\_\_\_\_?

If  $P_2$  is decidable, then  $P_1$  \_\_\_\_\_?

If  $P_2$  is undecidable, then  $P_1$  \_\_\_\_\_?

Q1. Given  $P_1 \leq P_2$  and  $P_2 \leq P_3$

- a)  $P_1$  is decidable  
 $P_3$  is undecidable, then,

$P_2$  is \_\_\_\_\_?

Q2. Given  $L_1 \leq L_2$  and  $L_2 \leq L_3$

- 1) If  $L_2$  is decidable then  $L_1$  is \_\_\_\_\_ and  $L_3$  is \_\_\_\_\_
- 2) If  $L_2$  is undecidable then  $L_1$  is \_\_\_\_\_ and  $L_3$  is \_\_\_\_\_

# Undecidability of the Post Correspondence Problem

## PROOF:

**Approach:** Take a problem that is already proven to be undecidable and try to convert it to PCP. If we can successfully convert it to an equivalent PCP then we prove that PCP is also undecidable.

**ACCEPTANCE (Halting) PROBLEM OF A TURING MACHINE** (*This is an undecidable problem*)

↓

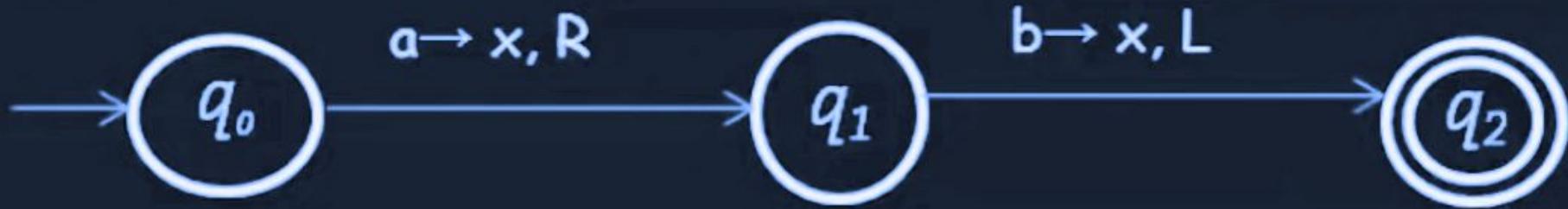
Convert this to PCP (called Modified PCP - MPCP)

So, to prove that **PCP is undecidable**, we want to **reduce** the TM halting problem to PCP.

So we take:

- A Turing Machine **M** and an input string **w** = aba

# Undecidability of the Post Correspondence Problem



Given:

- TM States:  $q_0, q_1, q_2$  (final)
- Input:  $w = aba$
- Alphabet:
  - Input symbols:  $\Sigma = \{a, b\}$
  - Tape symbols:  $\Gamma = \{a, b, x, B\}$  (includes blank  $B$  and symbol  $x$  used for marking)

# Undecidability of the Post Correspondence Problem

We have a Domino  $[#/#]$ , and the given TM Transitions are :

1. From  $q_0$ , if  $a \rightarrow$  write  $x$ , move **Right**, go to  $q_1$

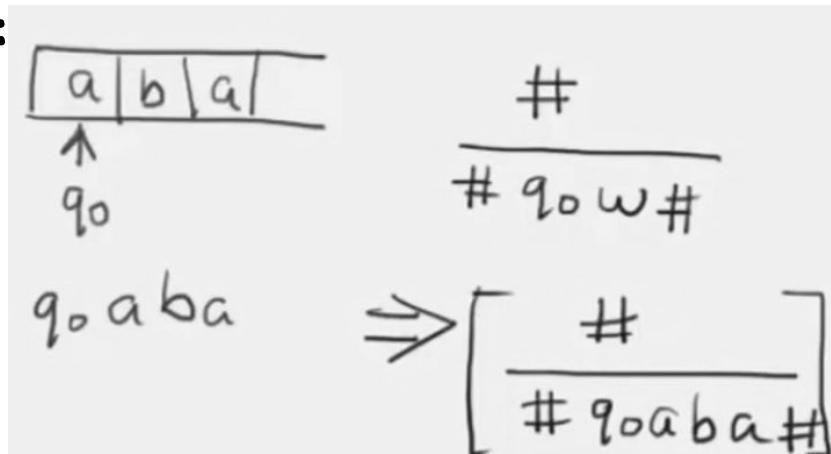
Notation:  $q_0 \text{ -- } a \rightarrow x, R \text{ --> } q_1$

2. From  $q_1$ , if  $b \rightarrow$  write  $x$ , move **Left**, go to  $q_2$

Notation:  $q_1 \text{ -- } b \rightarrow x, L \text{ --> } q_2$

Now we need to follow the following steps to convert the given Turing m/c into a format that can be simulated using **Post Correspondence Problem (PCP)**.

**Step 1:**



This becomes the **starting tile** in the PCP setup

# Undecidability of the Post Correspondence Problem

Step 2:

$$\therefore S(q_0, a) = (q_1, x, R)$$

$$\frac{\boxed{a}}{\begin{array}{c} \uparrow \\ q_0 \end{array}} = \frac{\boxed{x}}{\begin{array}{c} \uparrow \\ q_1 \end{array}}$$

$$q_0 a = x q_1 \Rightarrow \left[ \frac{q_0 a}{x q_1} \right]$$

# Undecidability of the Post Correspondence Problem

Step 3:

$$f(q_1, b) = (q_2, x, L)$$

$$\begin{array}{|c|c|}\hline Y & b \\ \hline\end{array} = \begin{array}{|c|c|}\hline Y & x \\ \hline\end{array}$$

$$Y \in \Gamma$$

$$\Gamma = \{a, b, x, B\}$$

$$Y q_1 b = q_2 Y x$$

$$\left[ \frac{aq_1b}{q_2ax} \right], \left[ \frac{bq_1b}{q_2bx} \right], \left[ \frac{xq_1b}{q_2xx} \right], \left[ \frac{Bq_1b}{q_2Bx} \right]$$

# Undecidability of the Post Correspondence Problem

Step 4: For all possible Tape symbols

$$\Gamma = \{a, b, x, B\}$$

$$\left[\frac{a}{a}\right], \left[\frac{b}{b}\right], \left[\frac{x}{x}\right], \left[\frac{B}{B}\right]$$

Step 5: For all possible tape symbols after reaching the accepting state

Accept  $q_2$

$$\left[\frac{aq_2}{q_2}\right], \left[\frac{q_2a}{q_2}\right], \left[\frac{bq_2}{q_2}\right], \left[\frac{q_2b}{q_2}\right], \left[\frac{xq_2}{q_2}\right], \left[\frac{q_2x}{q_2}\right], \left[\frac{Bq_2}{q_2}\right], \left[\frac{\bar{q}_2B}{q_2}\right]$$

# Undecidability of the Post Correspondence Problem

Step 6:

Step 6 : 
$$\begin{bmatrix} \# \\ \# \end{bmatrix} \quad \begin{bmatrix} \# \\ B\# \end{bmatrix}$$

Step 7 : 
$$\begin{bmatrix} q_2 \# \# \\ \# \end{bmatrix}$$

# Undecidability of the Post Correspondence Problem

Solution:

$$\left[ \begin{array}{c} \# \\ \# q_0 a b a \# \end{array} \right] \left[ \begin{array}{c} q_0 a \\ \times q_1 \end{array} \right] \left[ \begin{array}{c} b \\ b \end{array} \right] \left[ \begin{array}{c} a \\ a \end{array} \right] \left[ \begin{array}{c} \# \\ \# \end{array} \right]$$

From Step 1, 2 ,4 and 6

$$\left[ \begin{array}{c} \# \\ \# \times q_1 b a \# \end{array} \right] \left[ \begin{array}{c} \times q_1 b \\ q_2 \times x \end{array} \right] \left[ \begin{array}{c} a \\ a \end{array} \right] \left[ \begin{array}{c} \# \\ \# \end{array} \right]$$

From Step 3, 4 and 6

$$\left[ \begin{array}{c} \# \\ \# q_2 x x a \# \end{array} \right] \left[ \begin{array}{c} q_2 x \\ q_2 \end{array} \right] \left[ \begin{array}{c} x \\ x \end{array} \right] \left[ \begin{array}{c} a \\ a \end{array} \right] \left[ \begin{array}{c} \# \\ \# \end{array} \right]$$

From Step 5,4 and 6

$$\left[ \begin{array}{c} \# \\ \# q_2 x a \# \end{array} \right] \left[ \begin{array}{c} q_2 x \\ q_2 \end{array} \right] \left[ \begin{array}{c} a \\ a \end{array} \right] \left[ \begin{array}{c} \# \\ \# \end{array} \right]$$

From Step 5,4 and 6

$$\left[ \begin{array}{c} \# \\ \# q_2 a \# \end{array} \right] \left[ \begin{array}{c} q_2 a \\ q_2 \end{array} \right] \left[ \begin{array}{c} \# \\ \# \end{array} \right]$$

From Step 5 and 6

From Step 7

$$\left[ \begin{array}{c} \# \\ \# q_2 \# \end{array} \right] \left[ \begin{array}{c} q_2 \# \# \\ \# \end{array} \right]$$



So,we could reduce are acceptance problem of TM TO our PCP problem, hence PCP is also undecidable

# Recursive Function Theory

Not all functions, even those with precise definitions, can be computed by Turing Machines.

Example: Busy Beaver Functions

If you can define a given function using initial primitive recursive functions, then it is a Turing computable function/primitive recursive function.

So, some initial functions are taken as primitive recursive functions.

There are three initial functions:

1. Zero function (Z)

It is defined as  $Z(x) = 0$

- $Z(5) = 0$
- $Z(70) = 0$

2. Successor function (S)

It is defined as  $S(x) = x + 1$

- $S(6) = 6 + 1 = 7$
- $S(70) = 70 + 1 = 71$

3. Projection function (P)

$$p_i^n(x_1, x_2, \dots, x_i, \dots, x_n) = x_i$$

- $p_1^4(3, 7, 8, 9) = 3$
- $p_2^3(2, 9, 12) = 9$
- $p_4^6(2, 5, 12, 15, 18, 8) = 15$

# Recursive Function Theory

## Composition of Function

A composite function is defined when one function is substituted into another function.

- **Example:** Let  $f(x) = 3x + 2$  and  $g(x) = x + 5$ 
  - $f(g(x)) = f(x + 5) = 3(x + 5) + 2 = 3x + 17$
  - $g(f(x)) = g(3x + 2) = 3x + 2 + 5 = 3x + 7$

# Recursive Function Theory

## Recursive Function

A recursive function is a function that calls itself during its execution. Eg:factorial

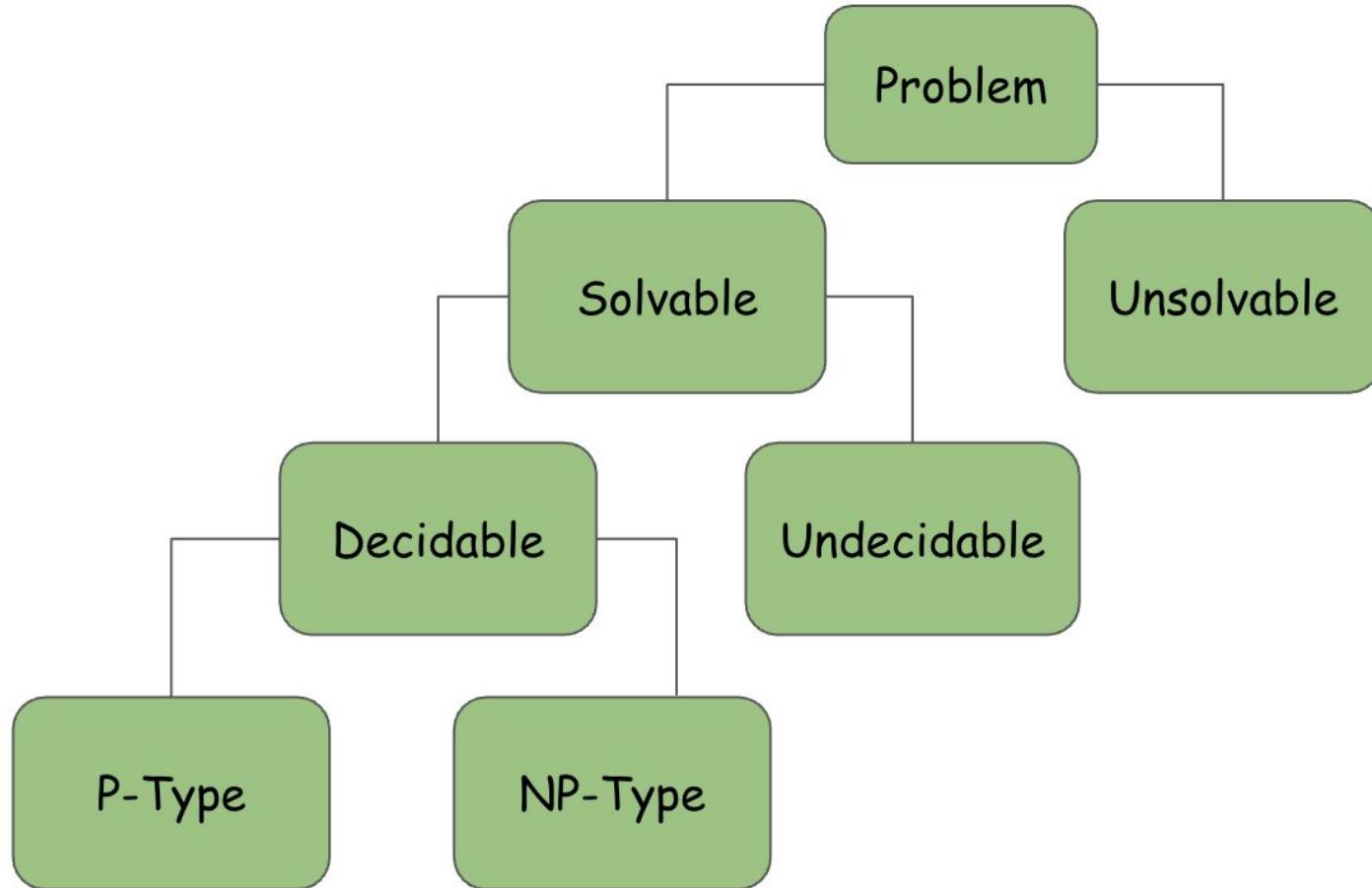
## Partial Recursive Function

A function is called a partial recursive function if it is defined for some of its arguments. They may not produce a result for every possible input. In other words, they can be undefined for some inputs.

Example: Subtraction of two positive numbers m and n,  $f(m, n) = m-n$ ; if  $m \geq n$ , otherwise it's not defined.

**Primitive Recursive Functions:** These are a subset of recursive functions that are defined using only basic initial functions (such as zero, successor, and projection functions) and can be formed through a finite combination of composition and recursion. They are always total functions, meaning they produce a result for every possible input. (We can create recursive func using the basic initial func)

# Complexity Theory



# Complexity Theory

**Solvable** - A problem is said to be solvable if either we can solve it or if we can prove that the problem cannot be solved

**Unsolvable** - A problem is said to be unsolvable if neither we can solve it, nor we can proof that the problem can not be solved

**Decidable**- Decidable problem is one for which there exists an algorithm that can determine the answer (yes or no) for every input in a finite amount of time. In terms of formal languages, a language is **decidable** if there exists a **halting Turing machine** that always decides membership in the language. Such a language is also called a **recursive language**

**Undecidable**- Undecidable problems are those for which no algorithm can determine the correct answer (yes or no) for all possible inputs. In other words, there is no Turing machine that always **halts** with a correct decision for every input. In terms of formal languages, a language is **undecidable** if there does **not exist a halting Turing machine** that decides membership in the language. Such languages are called **non-recursive languages**.

# Complexity Classes

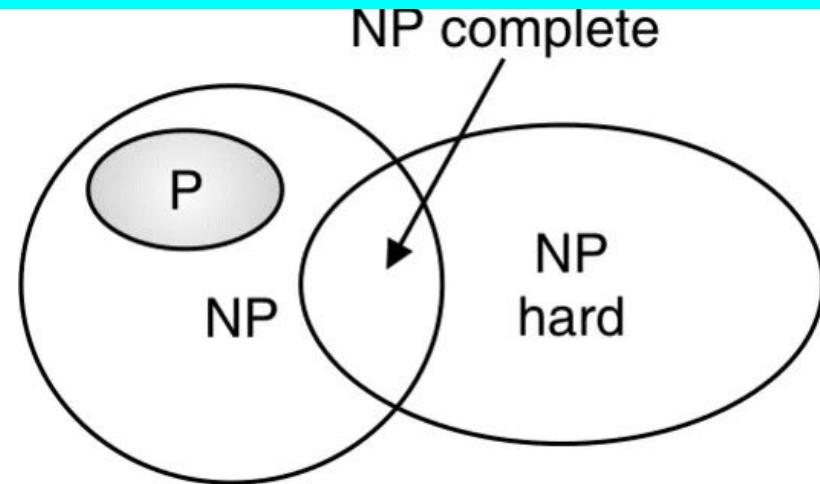
## Polynomial Time Algorithms

- Linear Search ( $O(n)$ )
- Binary Search ( $O(\log n)$ )
- Insertion Sort ( $O(n^2)$ )
- Merge Sort ( $O(n \log n)$ )

## Non-Polynomial Time (Exponential Time)

### Algorithms

- 0/1 Knapsack ( $2^n$ )
- Traveling Salesman ( $2^n$ )
- Graph Coloring ( $2^n$ )
- Sudoku ( $2^n$ )
- Scheduling ( $2^n$ )



Ex: If  $n = 10$ ,

then  $(10)^2 = 100$  and  $2^{10} = 1024$

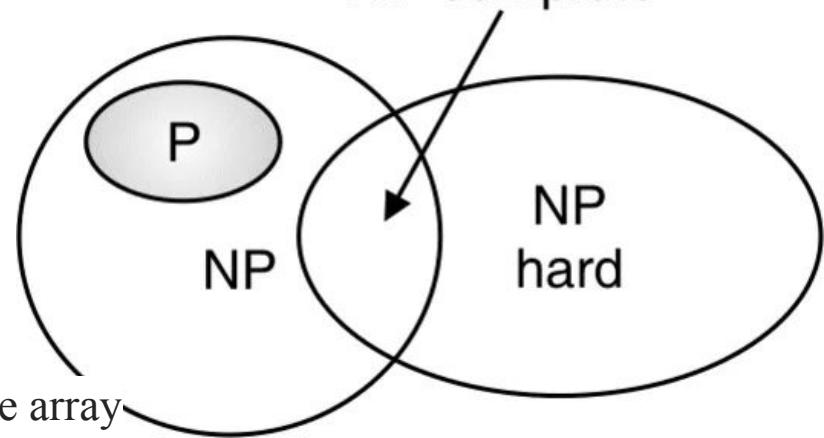
# P-Class Problems

- P (polynomial time) problems are a set of problems that can be solved in polynomial time by deterministic algorithms. A deterministic algorithm is an algorithm that always follows a fixed sequence of steps and produces the same output for the same input every time.
- P is also known as PTIME or DTIME complexity class.
- P problems are a set of all decision problems which can be solved in polynomial time using the deterministic Turing machine.
- They are simple to solve, easy to verify and take computationally acceptable time for solving any instance of the problem. Such problems are also known as “*tractable*”.

**NP complete**

## Examples of P Problems:

- Insertion sort
- Merge sort
- Linear search
- Matrix multiplication
- Finding minimum and maximum elements from the array

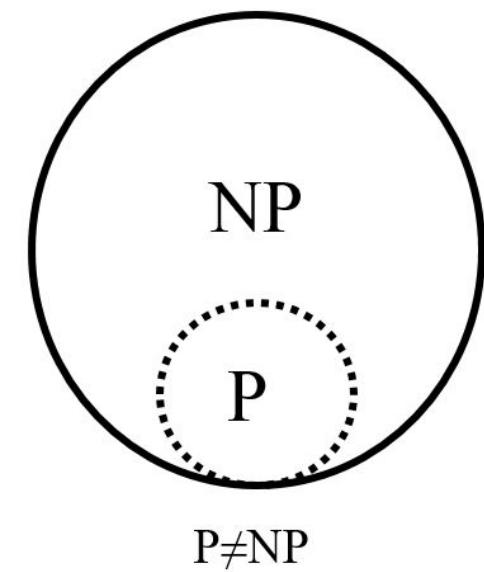


# NP-Class Problems

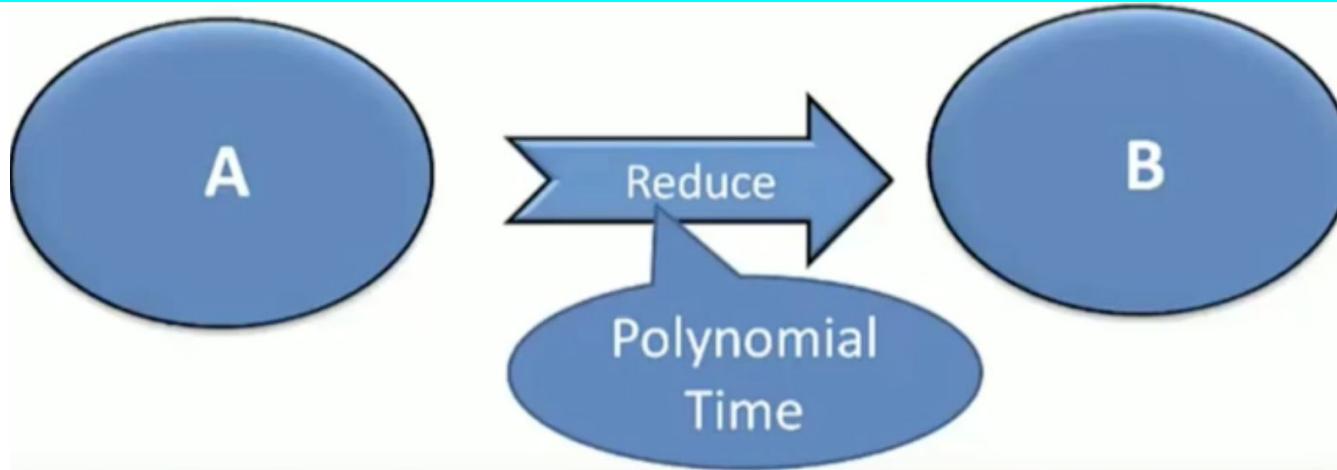
- NP (Non-deterministic polynomial time) problems are a set of problems that can not be solved in polynomial time, but given the solution, it can be verified in polynomial time.
- NP includes all problems of P, i.e.  $P \subseteq NP$ .

**Examples of NP problems:** Knapsack problem ( $O(2^n)$ ), Travelling salesman problem ( $O(n!)$ ), Tower of Hanoi ( $O(2^n - 1)$ ), Hamiltonian cycle ( $O(n!)$ )

P Problems	NP Problems
P problems are a set of problems that can be solved in polynomial time by deterministic algorithms.	NP problems are problems that can be solved in nondeterministic polynomial time.
P problems can be solved and verified in polynomial time.	The solution to NP problems cannot always be obtained in polynomial time, but if a solution is given, it can be verified in polynomial time.
P problems are a subset of NP problems.	NP problems are a superset of P problems.
<b>Example:</b> Selection Sort, Linear Search	<b>Example:</b> TSP (Traveling Salesman Problem), Knapsack Problem



# Polynomial time Reduction



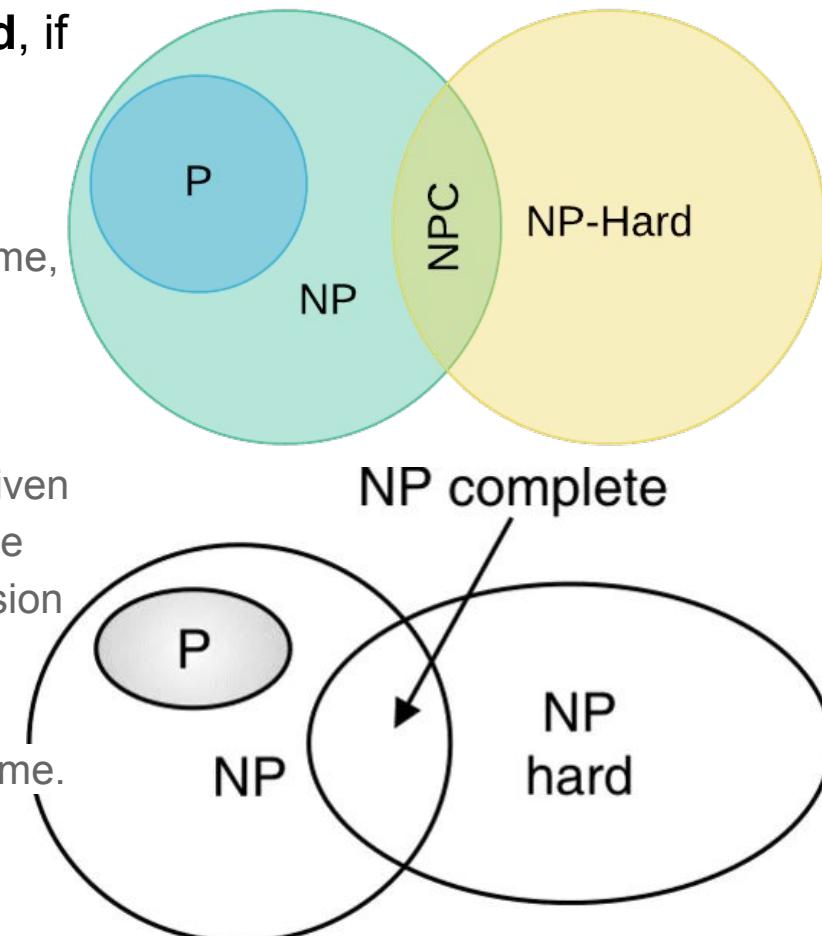
Let A and B are two problems then problem A reduces to problem B iff there is a way to solve A by deterministic algorithm that solve B in polynomial time.

If A is reducible to B we denote it by  $A \leq_p B$

# NP-Hard & NPC (NP-Complete) Problems

**NP-Hard:** A decision problem  $p$  is called **NP-hard**, if every problem in NP can be reduced to  $p$  in **polynomial time**.

- If we can solve any NP-hard problem in polynomial time, we would be able to solve all the problems in NP in polynomial time.
- A well-known example of the NP-hard problem is the Halting problem. The halting problem is stated as, “Given an algorithm and set of inputs, will it run forever ?” The answer to this question is Yes or No, so this is a decision problem.
- There does not exist any known algorithm which can decide the answer for any given input in polynomial time. So halting problem is an NP-hard problem.



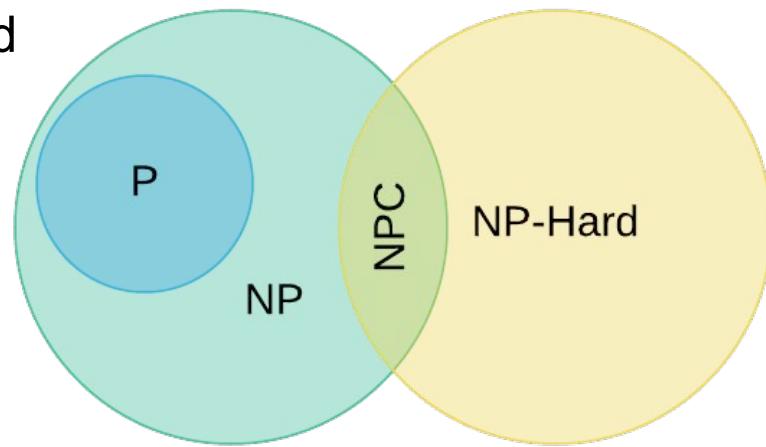
# NP-Hard & NPC (NP-Complete) Problems

**NPC (NP-Complete):** A decision problem A is called NP-complete if it has the following two properties :

- It belongs to class NP.
- It is NP-Hard

Examples of the well-known NP-complete problems are :

- Boolean satisfiability problem.
- Knapsack problem.
- Hamiltonian path problem.
- Travelling salesman problem.
- Subset sum problem.
- Vertex covers the problem.
- Graph colouring problem.
- Clique problem.



# Space Complexity

**The Power of Space:** Space seems to be more powerful than time because space can be reused.

## What is Space Complexity of a Turing Machine?

**Measuring Space:** A Turing Machine  $M$  runs in space  $S(n)$  if, for all inputs of length  $n$ ,  $M$  uses at most  $S(n)$  cells in total on its work tapes.

**Space Complexity** of a Turing machine = The **maximum number of tape cells** it scans or uses **during computation** on any input of size  $n$ .

It is a **function  $f(n)$** , where  $n$  is the length of the input string.

**Formal Definition:** Let  $M$  be a Turing machine, and let  $f(n)$  be a function, We say that  $M$  runs in **space  $f(n)$**  if : For every input of length  $n$ , the Turing machine **never scans more than  $f(n)$  tape cells** on any computation branch (for non-deterministic machines, on any branch)

# Space Complexity

The Turing machine **never scans more than  $f(n)$  tape cells** on any computation branch (for non-deterministic machines, on any branch), i.e

- No matter **which path** (or set of choices) the NDTM follows,
- It **never uses more than  $f(n)$  space (tape cells)** along that path.

So even if there are many possible branches, **each one individually must obey the space limit  $f(n)$ .**

# Space Complexity Classes

## 1. DSPACE( $f(n)$ )

- Problems solvable by a **Deterministic Turing Machine (DTM)** using at most  $f(n)$  space.
- Example: **DSPACE( $n$ )** means space grows linearly with input, for instance, SAT problem can be decided in linear space

## 2. NSPACE( $f(n)$ )

- Problems solvable by a **Non-deterministic Turing Machine (NTM)** using at most  $f(n)$  space **on any branch** of computation.
- Example: **NSPACE( $n^2$ )** = all problems solvable using **quadratic space**, non-deterministically

# Space Complexity Classes

**3. PSPACE:** Class of all problems decidable by a **deterministic TM using polynomial space.**

$$\text{PSPACE} = \bigcup \text{DSPACE}(n^k), \text{ for all } k \in \mathbb{N}$$

- Includes problems solvable in space:  $O(n)$ ,  $O(n^2)$ ,  $O(n^3)$ , ...
- Contains **P**, **NP**, and many more
- Example problems: Quantified Boolean Formula (QBF)

**4. NPSPACE:** Class of problems decidable by a **non-deterministic TM using polynomial space.**

$$\text{NPSPACE} = \bigcup \text{NSPACE}(n^k), \text{ for all } k \in \mathbb{N}$$

Note: By Savitch's Theorem, **PSPACE = NPSPACE**, As it proves that non-deterministic space doesn't give more power than deterministic space for polynomial bounds.

# Boolean Satisfiability Problem (SAT Problem)

This problem determines whether there exists a set of Boolean variable assignments that satisfy a given Boolean formula.

**Objective:** We aim to find values for Boolean

variables (TRUE, FALSE, or 1, 0)

such that the Boolean formula

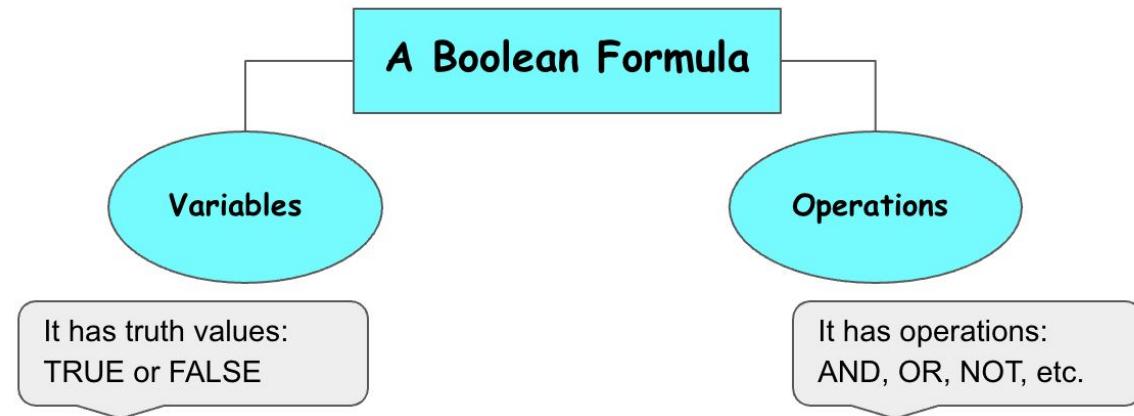
evaluates to TRUE (or 1)

**Input:** A Boolean Formula

**Output:** Find for which combination

of the variables (in the form of TRUE and FALSE),

the Boolean formula becomes TRUE.



P	Q	$P \wedge Q$	$\sim(P \wedge Q)$
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T

The Boolean formula  $\sim(P \wedge Q)$  is satisfiable for

P=F and Q= F

P=F and Q=T

P=T and Q= F

The Boolean formula  $P \wedge \neg P$  is NOT satisfiable

$P$	$\neg P$	$P \wedge \neg P$
T	F	F
F	T	F

For any value of  $P$ ,  $P \wedge \neg P$  is not TRUE

## **Steps for proving NP-Complete:**

**Step 1:** Prove that B is in NP

**Step 2:** Select an NP-Complete Language A.

**Step 3:** Construct a function f that maps members of A to members of B.

**Step 4:** Show that x is in A iff  $f(x)$  is in B.

**Step 5:** Show that f can be computed in polynomial time.

# NDTM $\leq_P$ SAT

- 1)  $Q^t q \rightarrow \text{current state}$
- 2)  $H^{t_{ij}} \rightarrow \text{tape \& current cell}$
- 3)  $T^{t_{ijs}} \rightarrow \text{current symbol}$

## Conditions

- 1) Single state at a time

$$Q^t q \rightarrow \neg Qq'{}^t \equiv \neg Q^t q \vee \neg Qq'{}^t$$

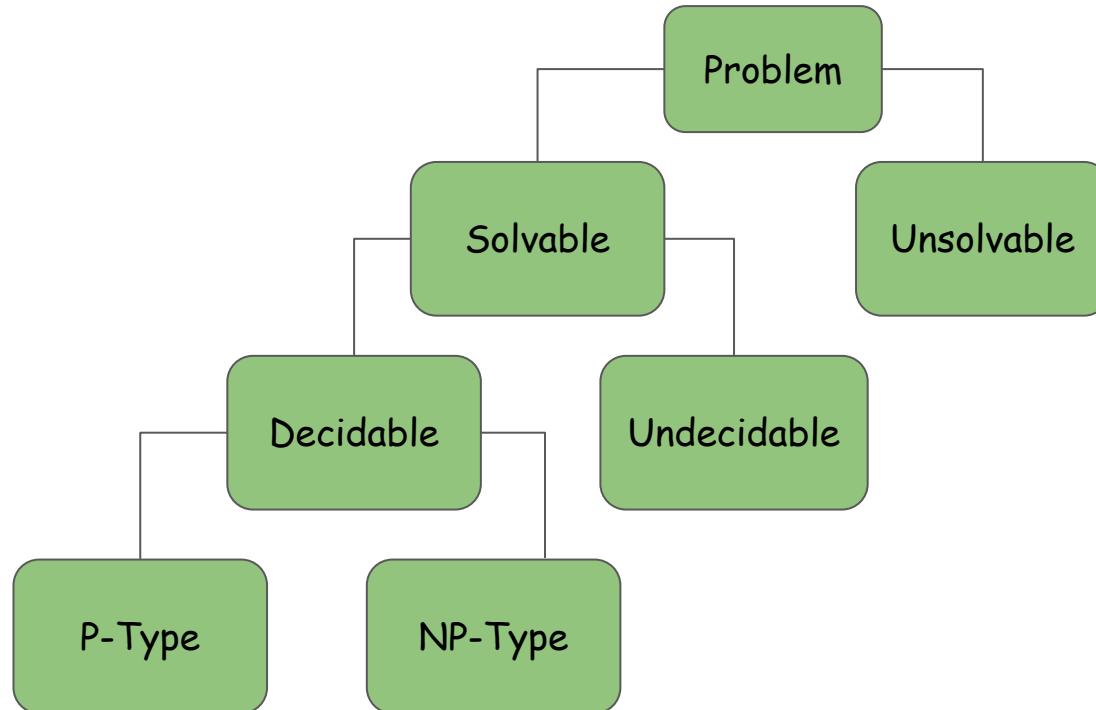
- 2) In a single cell at a time

$$H^{t_{ij}} \rightarrow \neg H^{t_{ij}'} \equiv \neg H^{t_{ij}} \vee \neg H^{t_{ij}'}$$

- 3) Single symbol

$$T^{t_{ijs}} \rightarrow \neg T^{t_{ijs}'} \equiv \neg T^{t_{ijs}} \vee \neg T^{t_{ijs}'}$$

# Complexity Theory



# Ackermann Function

The Ackermann function is a two-parameter function that takes non-negative integers as inputs and produces a non-negative integer as its result. The **Ackermann function is total**. The function is defined as follows:

$$\begin{aligned} A(m,n) = & \{n+1, && \text{if } m=0 \\ & A(m-1,1) && \text{if } m>0 \text{ and } n=0 \\ & A(m-1,A(m,n-1)) && \text{if } m>0 \text{ and } n>0 \end{aligned}$$

**UNIT 1: DFA, NFA, EQ., MINMZTN**

**IMPORTANT TOPICS**

**UNIT 2: RE, ARDEN'S, PL**

**UNIT 3: CFG, CFG SMPLFCTN,  
AMBIGUOUS, RLG, RLG,  
CHOMSKY,CNF,GNF**

**UNIT 4: NPDA**

**DPDA**

**Pumping Lemma for CFL  
Closure Properties of CFL  
Decision Problems of CFL**

**UNIT 5: TM, UTM, LBA, PCP**