Theory of computation

Unit - I

Automata and Regular Expression

Theory of computation (Toc)

* It is a branch of computer science which deals with how efficiently the problems are solved on the particular model of computation with the help of algorithm

* It is classified into 3 types namely

1) automata theory and language
2) Computation theory
3) complexity

Automata theory of language:

It deals with definition and population on various mathematical model of computation

Mathematical model's

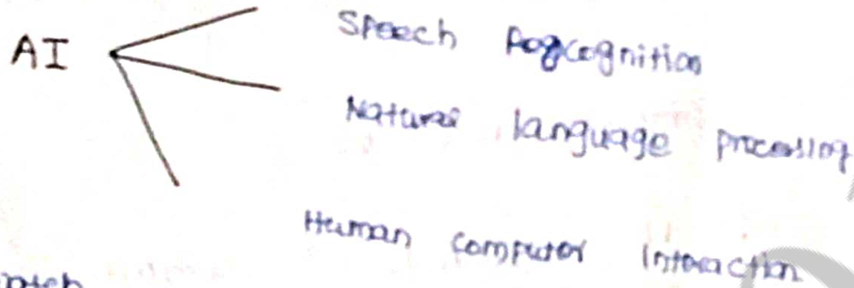
* Turning machine

* finite Automata.

* Push Down Automata

Main Purpose of TOC:

Develop mathematical model after computation for real world computers



AI

Speech Recognition

Natural language processing

Human computer interaction

Eg: Watch with timer

Basic Definition

1) Symbol:

Symbol is a character

Eg:

a, b, c, .. z

0, 1, 2 ... 9

+, -, x, ÷ ... Special character

Alphabet:

An Alphabet is a finite - non-empty set of symbol and it is denoted by Σ

E.g

$\Sigma = \{0,1\}$ - set of binary alphabet's

$\Sigma = \{a, b, c \ldots z\}$ set of lowercase alphabets

String / Word

It is a finite set of sequence of symbol's Choosen from some alphabet's

E.g

a) 01110110 is a String from Σ= {0,1}

b) a, aa, aaa are all String over the alphabet Σ= {a}

Empty String (or) Null String : ε

An Empty String is the String from with zero occurence of symbol's (ε)

Length of String:

Let w be the String then the of the String is the number of symbol's composing the String. and it's denoted by |w|

Eg:

1) If W= abcd the |w|= 4

2) If W= 010011000 the |w| = 9

3) |ε| =0

Symbols → Alphabet → String

Automata: Study of abstract computing device

Why?

→ Complexity

→ To implement our brain function (finite automata

Application:

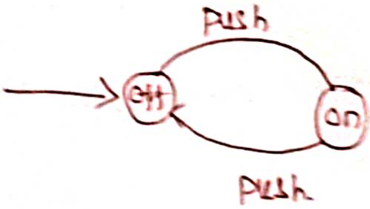
* Software designing	* web pages
* Digital circuits	* Robotics

Limitations:

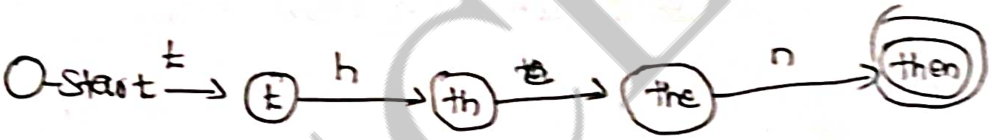
* It can Recognised only simple language

* FA can be designed only for decision making problems.

(i) Finite automation for an on/off switch
- Digital system



(ii) Lexical analysis - Recognising a String "then"



O – Initial State

→ – transition

◎ final state (or) accepted state

Finite Automata:

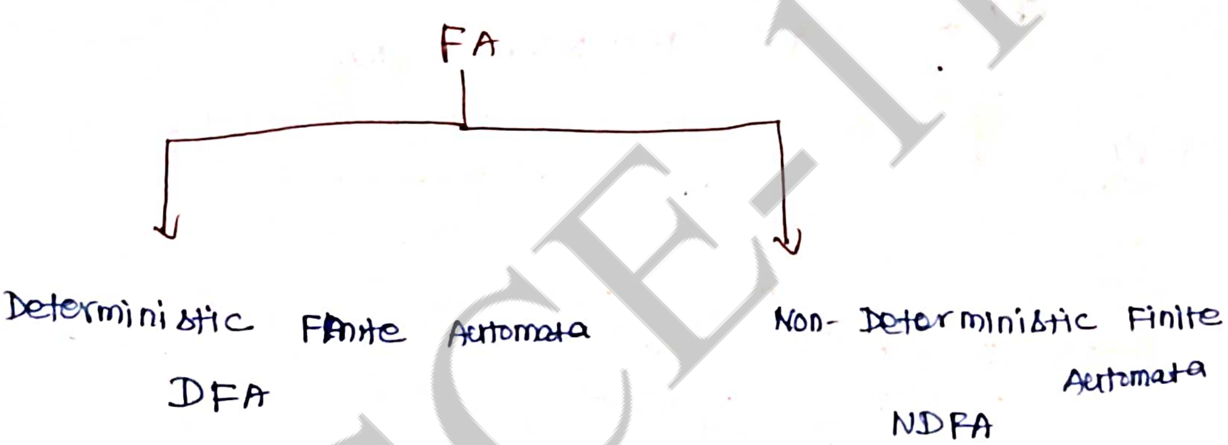
* Introduction:

It is a self / operating machine / It is a system / which / obtains transforms

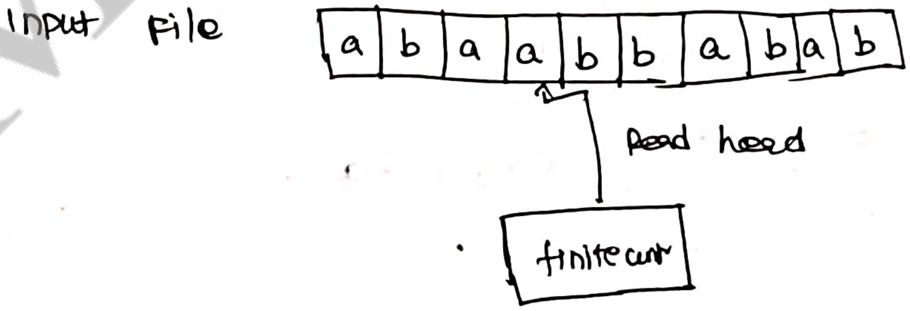
The finite Automation F(A) is a mathematical model of a system. With discrete inputs and outputs a finite number of memory configuration called states and a set of transitions from state to state that occurs on input symbols from alphabet Σ.

Language:

A set of strings taken from an alphabet is called a language.


FA
Deterministic Finite Automata
DFA
Non- Deterministic Finite Automata
NDFA

Basic DFA



DFA is a language Recogniser that has

(i) An input file - containing an input string

(ii) A finite control - a device that can be in a finite number of states

(iii) A reader — a sequential Reading device

(iv) A program

Deterministic finite Automata

A deterministic finite automation (DFA) is a five-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where.

Q - finite non-empty set of states

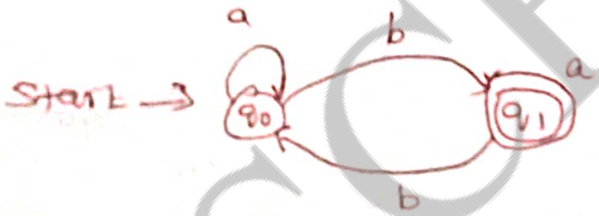
Σ - is a finite set of input symbols

q_0 - initial states (or) start states $q_0 \in Q$

F - Final state (or) accepting states

$\delta : Q \times \Sigma \rightarrow Q$ transition function

Example



$Q = \{q_0, q_1\}$

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

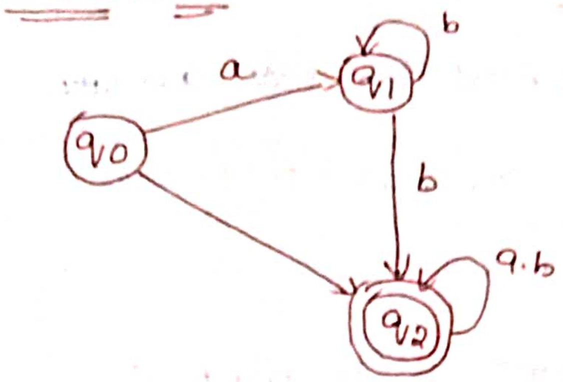
q_0 - Initial state

$F = q_0$

$\delta(q_0, a) = q_0$

$\delta(q_0, b) = q_1$

Example 2



Q= q0, q1, q2

Σ = {a.b}

q0 — initial state

q2 — final state

$\delta(q_0,a) = q_1$, $(q_1,b) = q_2$

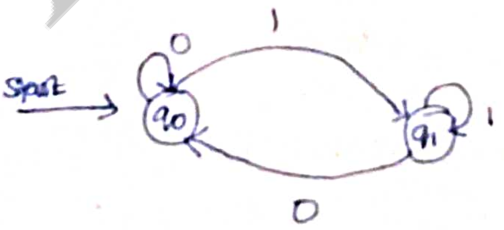
Transition Table

A transition table is a conventional tabular Representation of function like δ that takes 2 arguments and Returns a values.

δ	0	1
q0	q2	q0
q1	q1	q1
q2	q2	q1

Transition Diagram:

A transition diagram is a directed graph associated with the vertices of the graph corresponding to the state of finite automata



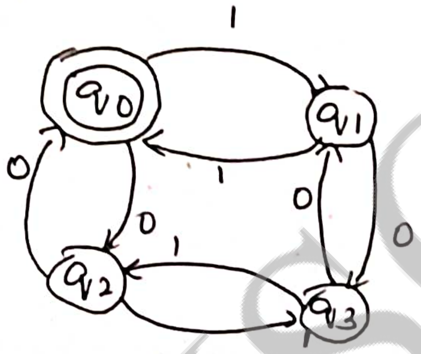
Language acceptance by FA

A string x is accepted by finite automata $M = (Q, \Sigma, \delta, q_0, F)$ only if $\delta(q_0, x) = P$ for some P in F

The language accepted by M which is denoted by L(M).

$$L(M) = \{ x / \delta(q_0, x) \text{ in } F \}$$

Check whether the i/p string 110101 is accepted by FA (or) not



Sol

State i/p	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

$\delta(q_0, 110101)$

$= \delta(\delta(q_0, 1), 10101)$

$= \delta(\delta(q_1, 1). 0101)$

$= \delta(\delta(q_0, 0), 101)$

$= \delta(\delta(q_2, 1), 01)$

$= \delta(\delta(q_3, 0), 1)$

accepted state $= \delta(\delta(q_1, 1)$

$= q_0$

∴ 110101 is in L(M)

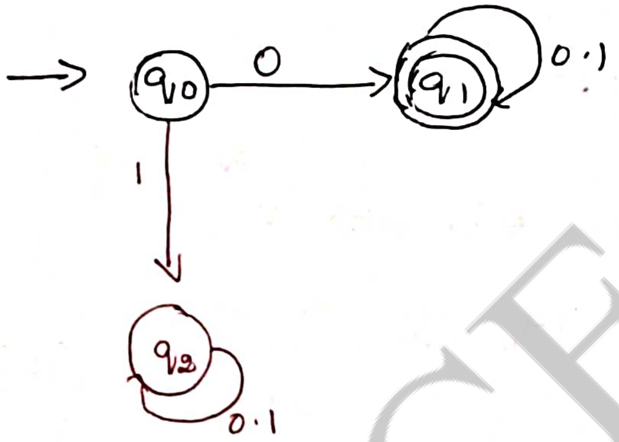
Deterministic Finite Automata (DFA)

The term deterministic refers to the fact that on each i/p there is one and only state to which the automation can have transition from its current state.

DFA:

1) L1 = set of all strings that start with 0

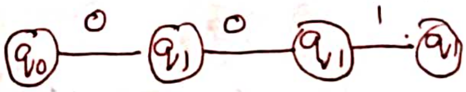
L1 = {0, 00, 01, 000, 001, 0110.}



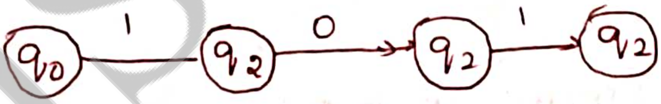
final state

Dead state / trap state

001

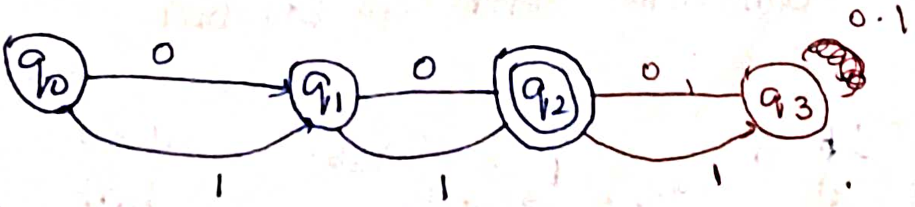


101

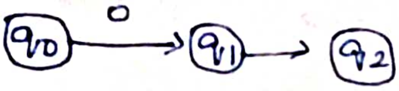


② Construct a DFA that accepts of all over {0,1} of length 2

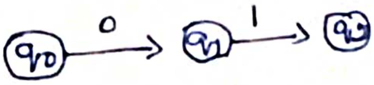
L= {00, 10, 01, 11}



.00



01



Non- Deterministic finite Automata

The non- deterministic finite Automata (NFA) is defined by a five tuple $(Q, \Sigma, q_0, \delta, F)$ Where

Q - finite non- empty set of States

Σ - Finite set of input alphabet

$q_0 \in Q$ - Start State, belongs to Q

$F \subseteq Q$ - Set of final States, Subset of Q

δ - mapping function $Q \times \Sigma$ to 2^Q

(2^Q is Power set of Q)

Extended Transition function ($\bar{\delta}$)

The function δ can be extended to a function $\bar{\delta}$ mapping $Q \times \Sigma$ to 2^Q such that

$\hat{\delta}(q, \epsilon) = \{q\}$

$$\hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q,w)} \delta(q,a) \quad \text{for each } w \in \Sigma^*,\ a \in \Sigma$$

Since $\hat{\delta}(q, a) = \delta(q, a)$ for an input symbol 'a' we may use δ in place of $\hat{\delta}$

Also

$$\delta(\{p_1, p_2, \dots, p_n\}, x) = \bigcup_{i=1}^{n} \delta(p_i, x)$$

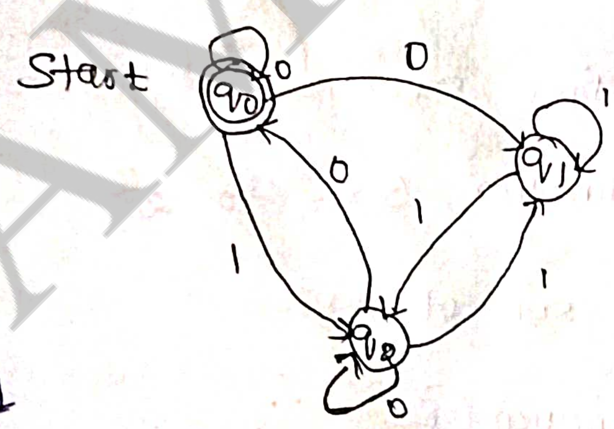
Language of a NFA

A language is accepted by M if there exists some state in both F and $\delta(q_0, x)$

i.e ~~E~~ $L(A) = \{ w : \delta(q_0, w) \cap F \neq \phi \}$

Example 1

For the NFA shown, check whether the input string 0100 is accepted or not



Sol

The transition table δ is

States	Inputs 0	1
q0	{q0, q1}	{q2}
q1	φ	{q1, q2}
q2	{q0, q2}	{q1}

Input String = 0100

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

~~δ(-δ(q0~~ ~~δ(δ(q0,~~

δ(~~q0,~~ 0100) = ~~δ(δ(q0,0), 100~~)

= ~~δ({q0,q1}, 100~~)

$\delta(q_0, 01) = \delta(\delta(q_0, 0), 1)$

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

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

$= \{q_2\} \cup \{q_1, q_2\}$

$= \{q_1, q_2\}$

$\delta(q_0, 010) = \delta(\delta(q_0, 01), 0)$

$= \delta(\{q_1, q_2\}, 0)$

$= \delta(q_1, 0) \cup \delta(q_2, 0)$

$= \{\phi\} \cup \{q_0, q_2\} = \{q_0, q_2\}$

$\delta(q_0, 0100) = \delta(\delta(q_0, 010), 0)$

$= \delta(\{q_0, q_2\}, 1)$

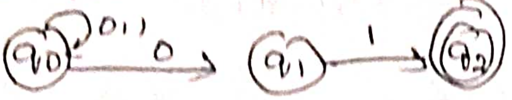
$= \delta(q_0, 1) \cup \delta(q_2, 1)$

$= q_2 \cup \{q_1\}$

$\delta(q_0, 0100) \cap F = \{q_2, q_1\} \cap \{q_0\} \neq \phi$. not accepted

(2) For NFA check whether the input string 001 is accepted (or) not

Sol



$\delta(q_0, 001) = \delta(\delta(q_0, 0), 01)$

$= \delta(\{q_0, q_1\}, 0, 1)$

$= \delta(\delta(\{q_0, q_1\}, 0), 1)$

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

$= \delta(\{q_0, q_1) \cup \{\phi\}, 1)$

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

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

$= \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

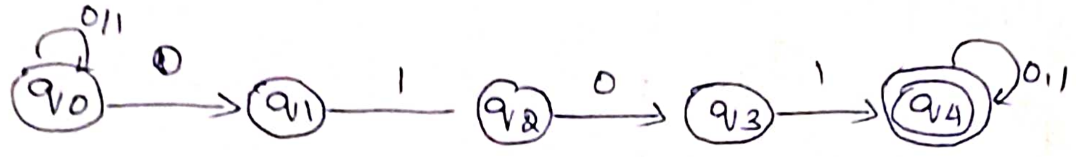
By Language of an NFA

$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \phi\}$

L(A) is the set of strings w in Σ^* such that $\hat{\delta}(q_0, w)$ contains atleast 1 accepting state

② Design a NFA to accept strings containing the substring 0101

Sol



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

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

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

$q_0 = \{q_0\}$

$F = \{q_4\}$

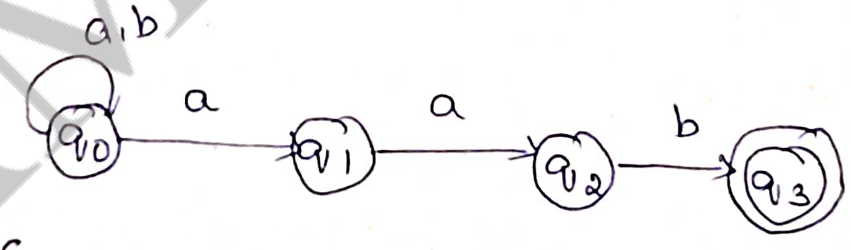
Transition Table

States	Inputs	
	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	ϕ	$\{q_2\}$
q_2	$\{q_3\}$	ϕ
q_3	ϕ	$\{q_4\}$
q_4	$\{q_4\}$	$\{q_4\}$

Construct a NFA that accepts

$L = \{x \in \{a, b\} \,/\, x$ ends with aab$\}$

Sol



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

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

$q_0 = \{q_0\}$

$F = \{q_3\}$

δ is defined by

$\delta(q_0, a) = \{q_0, q_1\}$

$\delta(q_0 . b) = \{q_0\}$

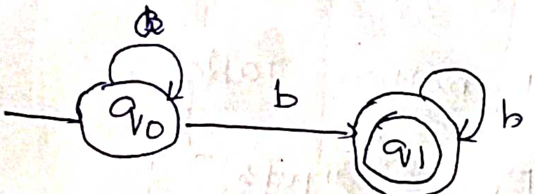
$\delta(q_1\ a) = \{q_2\}$

$\delta(q_2, b) = \{q_3\}$

Q/Σ	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	ϕ
q_2	ϕ	$\{q_3\}$
q_3	ϕ	q

③ Design a NFA for $L = \{x \in \{a.b\}^* \,/\, x$ contains any number of a's followed by atleast one b $\}$

Sol



NFA :

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

$Q = \{q_0, q_1\}$; $\Sigma = \{a, b\}$

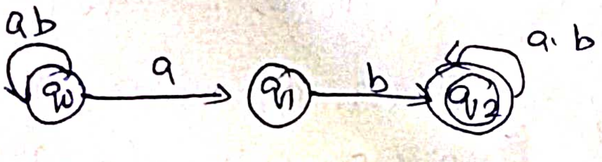
$q_0 = \{q_0\}$ $F = \{q_1\}$

δ

State	Input	
	a	b
q_0	$\{q_0\}$	$\{q_1\}$
q_1	ϕ	$\{q_1\}$

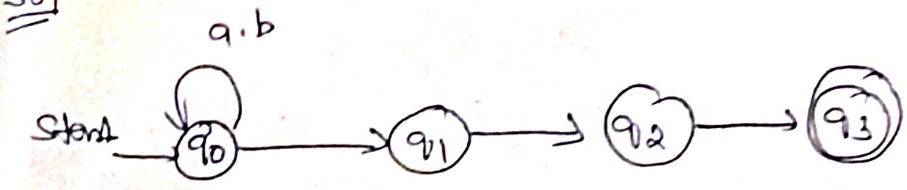
① Exercise

1) Construct a NFA over alphabet $\Sigma = \{a, b\}$ that accept string with substring ab



② Construct a NFA that accepts string which has 3rd symbol 'b' from Right

Sol



) Construct the DFA equivalent to the NFA $M = (\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_1\}$ and δ is defined as

States	Inputs	
	0	1
q_0	$\{q_0\ q_1\}$	$\{q_1\}$
q_1	ϕ	$\{q_0, q_1\}$

Sol

DFA = $M' = (Q', \{0,1\}, \delta' [q_0], F')$ accepting $L(M)$.

$Q' = 2^Q$ (all subset of $Q = \{q_0, q_1\}$) $\Rightarrow 2^2 = 4$

$= \{\phi, [q_0], [q_1], [q_0, q_1]\}$

$\delta'([q_0], 0) = [q_0, q_1]$

$\delta'([q_0], 1) = [q_1]$

$\delta'([q_1], 0) = \phi$

$\delta' [[q_1], 1] = [q_0, q_1]$

$\delta'([q_0, q_1], 0) = [q_0, q_1]$

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

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

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

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

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

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

$= \{q_0, q_1\} \cup \phi = \{q_0, q_1\}$

$\delta'([q_0,q_1],1) = [q_0,q_1]$ $\quad\quad$ $\delta(\{q_0,q_1\},1)$

$$\Rightarrow \delta(q_0,1) \cup \delta(q_1,1)$$

$$= \{q_1\} \cup \{q_0,q_1\}$$

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

DFA transition table is

States	Inputs	
	0	1
$[q_0]$	$[q_0,q_1]$	$[q_1]$
$[q_1]$	ϕ	$[q_0,q_1]$
$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_1]$

Equivalence of NFA and DFA

Theorem :

Let L be a set accepted by NFA. Then there exist's a DFA that accept L.

proof:

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA for language L

Then define DFA $M' = \{Q', \Sigma, \delta', q_0', F'\}$

The States of M' are all the subset of M'

The $Q' = 2^Q$

$F' =$ be the set of all the final states in M.

The element in Q' will be denoted by $[q_1, q_2 \cdot q_i]$ and the element in Q are denoted by $[q_1, q_2 \cdot q_n]$

The $[q_1, q_2 \cdot\cdot q_i]$ will be assumed as one state in Q' if on the NFA q_0 is initial state it is denoted in DFA as $q_0' = [q_0]$

Now we define

$$\delta'([q_1, q_2 \cdot\cdot q_i]\, a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \delta(q_i, a)$$

equivalently

$$\delta'\{[q_0, q_2 \cdot\cdot q_i] . a) = [p_1, p_2 \cdot\cdot p_j]$$

iff

$$\delta(\{q_1, q_2 \cdot q_i\}, a) = \{p_1, p_2 \cdot\cdot p_j\}$$

Proof By Induction

Input string x

$\delta'(q_0, x) = [q_1, q_2 \cdots q_i]$

Iff

$\delta(q_0, x) = \{q_1, q_2 \cdots q_i\}$

Basis Step:

The Result is trivial if string length is 0 i.e $|x|=0$

Since $q_0' = [q_0]$

Induction Step:

If we assume that the hypothesis is true for the string of length m (or) less the m Then if x is a string of length m+1, The function δ' should be written as

$\delta'(q_0', xa) = \delta'(\delta'(q_0, x), a)$

By the Induction hypothesis

$\delta'(q_0', x) = [p_1, p_2 \cdots p_j]$

Iff

$\delta(q_0, x) = \{p_1 p_2 \cdots p_j\}$

By definition of δ'

$\delta'([p_1, p_2 \cdots p_j], a) = [r_1, r_2 \cdots r_k]$

iff $\delta(\{p_1, p_2 \cdots p_j\}, a) = \{r_1, r_2 \cdots r_k\}$

Thus $\delta'(q_0', xa) = [r_1, r_2 \cdots r_k]$

iff

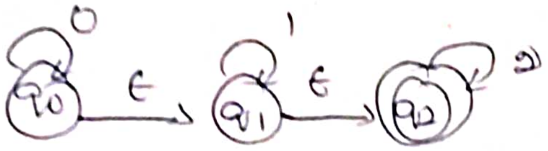
$\delta(q_0, xa) = \{r_1, r_2 \cdots r_k\}$

which establishes the inductive hypothesis

Thus $L(M) = L(M')$

Finite Automata with ε- Moves

It is possible in NFA that an NFA is allowed to make transition simoteneously without Receiving an i/p symbol. This move is called ε moves. This ε. Represent any number of times



State	input			
	ε	0	1	2
q0	q_ϕ	q0	φ	φ
q1	q2	φ	q1	φ
q2	φ	φ	φ	q2

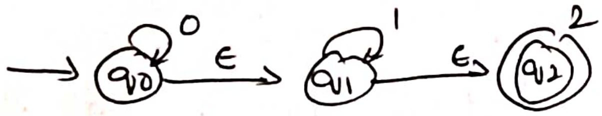
~~Epti~~ Epsilon (ε) - closure

If state p is in ε- closure (q), and there is a transition from state p to state r labled ε, then r is in ε- closure (q). More precisely If δ is the function of the ε- NFA involved and p is in ε closure (q) then, ε- closure (q) also contains all the state in δ(P. ε)

Naturally Let ε- closure, where p is a set of states then

$$\bigcup_{q\,in\,P} \epsilon\text{ - closure }(q)$$

① Find ε- closure



Find $\hat{\delta}(q_0, 01)$

Sol

ε- closure $(q_0) = \{q_0, q_1, q_2\}$

ε- closure $(q_1) = \{q_1, q_2\}$

ε- closure $(q_2) = \{q_2\}$

$\hat{\delta}(q_0, 01) =$ ε- closure $\delta(\hat{\delta}(q_0,1),1))$

$=$ ε - closure $\delta[\{q_0, q_1, q_2,0\},1)$.

$=$ ε- closure $\delta(\delta\{q_0,0\} \cup \delta(q_1,0) \cup \delta(q_2,0),1)$

$=$ ε- closure $(\delta(\{q_0, \phi, \cup\phi\}),1)$

$=$ ε- closure $(\delta(q_0,1))$.

$=$ ε- closure (ϕ).

$= \phi$

Language of ε- NFA

The language of an ε- NFA,

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

Theorem

If L is accepted by NFA with ε-transition Then L is accepted by an NFA without ε-transition

Sol proof:

~~If~~ ~~L is~~ ~~accepted~~ by ~~NFA~~

Let $M = (Q, \Sigma, \delta, q_0, F)$. be an NFA with ε-transition

construct M' Which is NFA without ε-transition.

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

$$F' = \begin{cases} F \cup \{q_0\} & \text{if } \varepsilon\text{-closure } q_0 \text{ contains a state of } F \\ F & \text{otherwise} \end{cases}$$

By Induction.
$$\begin{cases} \delta' \text{ \& } \hat{\delta} \text{ are same} \\ \delta \text{ \& } \hat{\delta} \text{ are different} \end{cases}$$

Let x be any string

$$\hat{\delta}(q_0, x) = \hat{\delta}(q_0, x)$$

This statement is not true if $x = \varepsilon$ because

$\delta'(q_0, \epsilon) = \{q_0\}$ & $\hat{\delta}(q_0, \epsilon) = \varepsilon\text{-closure}(q_0)$

Basic step:

$|x| = 1$ x is a symbol whose vale is a

$\delta'(q_0, a) = \hat{\delta}(q_0, a)$ (because by definition $\hat{\delta}$)

Induction step:

Let $x = wa$ where a is in Σ

$$\delta'(q_0, wa) = \delta'(\delta'(q_0, w), a)$$

$$= \delta'(\hat{\delta}(q_0, w), a)$$

$$= \delta'(p, a)$$ [Because by inductive hypothesis]

$\delta(q_0, w) = \hat{\delta}(q_0, w) = p$ (say)

Now we must show that

$$\delta'(p, a) = \hat{\delta}(q_0, wa)$$

But

$$\delta'(p, a) = \bigcup_{q \text{ in } p} \delta'(q, a) = \bigcup_{q \text{ in } p} \hat{\delta}(q, a)$$

$$= \hat{\delta}(\hat{\delta}(q_0, w), a)$$

$$= \hat{\delta}(q_0, wa)$$

$$= \hat{\delta}(q_0, x)$$

Hence $\delta'(q_0, x) = \hat{\delta}(q_0, x)$

Unit-II

Regular Expression and Language

Regular Langguge:

Languge that can be Represented using F.A / Regex

Regen – short (powerfue). → patten matching.

$\Sigma = \{a\}$ $L = \{a. aaa, aaaa \dots\}$

Regex	Language
ϵ	$L(\epsilon) = \{\epsilon\}$
ϕ	$L(\phi) = \{\}$
a	$L(a) = \{a\}$

E.g

$L = \{a. aa. aaa\} = a + aa + aaa$

$L = \{aa. ab, ba. bb\} = aa + ab + ba + bb$

Operation in Rex

① Union

$L_1 = \{a.b\}$

$L_2 = \{cd. cc\}$

$L_1 \cup L_2 = \{a, b, cd, cc\}$

Concatenation: (.)

$L_1 = \{a, b\}$ $L_2 = \{cd, cc\}$

$L \cdot M = \{acd, acc, bcd, bcc\}$

Kleen closure: L^*

$L = \{a\},$ $L^0 = \{\epsilon\}$ $L^1 = \{a\}$

$L^2 = \{aa\}$ $L^3 = \{aaa\}$

$$L^* = \bigcup_{i=1}^{\infty} L^i$$

$L^* = \{\epsilon, a, aa, aaa \dots\}$

(2) Example for Kleen closure

$L = \{a, b\}$

Sol

Find $L^* = ?$ $L^0 = \{\epsilon\}$

$L^1 = \{a, b\} = L$

00
10
01
11

$L^2 = \{aa, ab, ba, bb\}$

$L^3 = \{aaa, aab\ aba\ abb, baa, bab\ bba, bbb\}$

$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \dots$

$L^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\}$

Example 2

$L = \{a, ab\}$

$L^0 = \{a\}$, $L^1 = \{a.ab\}$ $L^2 = \{$ ~~aab~~, $aa.\ aab,\ aba,\ abaab\}$

$L^3 = \{$

Difference between * and +

$\Sigma = \{a\}$

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$

$= \{\epsilon, a.\ aa,\ aaa \ldots\}$

$\Sigma^+ = \Sigma^* - \Sigma^0$

$= \{a.aa,\ aaa \ldots\}$

Regex for finite language

SI. NO	Specification	Language	Regex
1	No String	$\{\}$	ϕ
2	Length 0	$\{\epsilon\}$	ϵ
3	Length 1	$\{a.b\}$	$a+b$
4	2	$\{aa.ab\ ba.bb\}$	$aa+ab+bb+bb$ $a(a+b)+b(a+b = (a+b)(a+b)$
5	3	$\{aaa.\ aab \ldots\}$	$(a+b)^3$
6)	Atmost 1	$\{\epsilon, a, b\}$	$\epsilon+a+b$

7) Atmost 2 $\{\epsilon\ ab\ aa.\ abb\} = (\epsilon+a+b)^2$

Precedence of Regex

$*$ – hight Precedence

S.I NO	specification	Language	Represent
1	Begin with 110	{11000, 1100 110011 · ·1011111 · · }	$110(0+1)^*$
②	Containg 1101	= {01101, 001101	$(0+1)^*\ 1101(0+1)^*$
③	exactly three 1's	= {111, 01110 111000}	$0^*10^*10^*1$
④	Ending with 110	= {110, 01110 · · }	$(0+1)^*\ 110$

⑤ Having single b

L= { b, ab, aab, aba }

$a^*\ b\ a^*$

⑥ Having atleast one b

$(a+b)^*\ b\ (a+b)^*$

⑦ Having bbb as substring

{ bbb abbbb}

$(a+b)^*\ bbb\ (a+b)^*$

⑧ Ending with ab

$(a+b)^*\ (ab)$

Begining with ba / or +

$ba(~~b~~a+b)^*$

Containing a

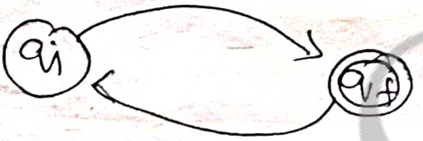
~~$(a+b)^*$~~ $a(a+b)^*$

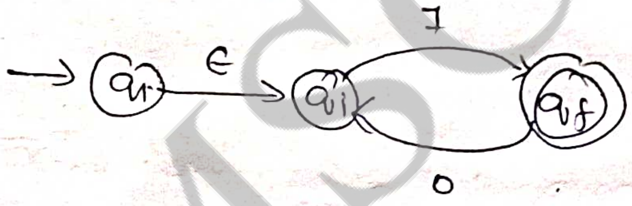
Begining (~~a+b~~)

Start & end with different symbol.

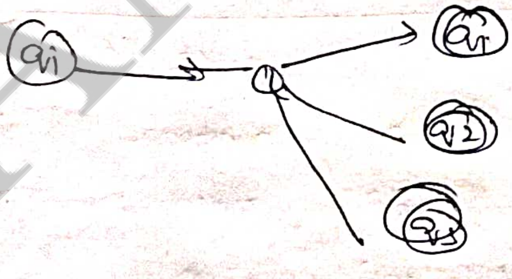
$a(a+b)^*b + b(a+b)^*a$

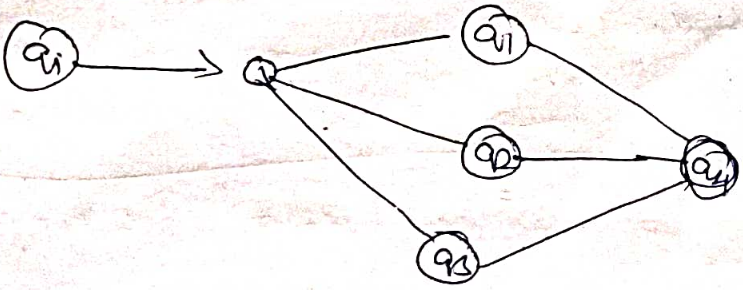
① No Incoming Edge For Initial state





② Only one final state must be present

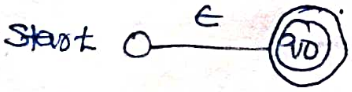




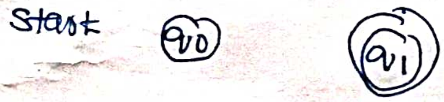
Conversion of Regular Expression to NFA with ε transition (Thompson's) construction

Basis:

1) RE = ε

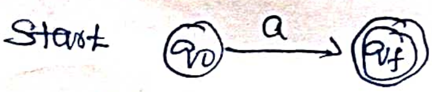


2) RE = φ



ε- NFA for φ

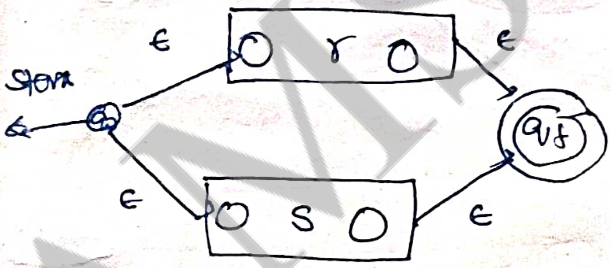
3) RE = a $\forall\, a \in \Sigma \Rightarrow a = \{a\}$



ε- NFA for a

Introduction:

RE = r+s



Presedence

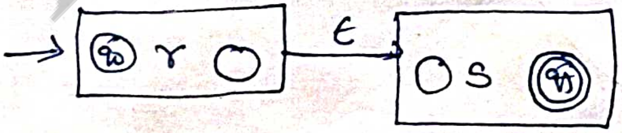
1) ( )

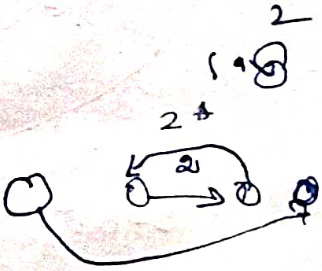
2) *

3) concatination

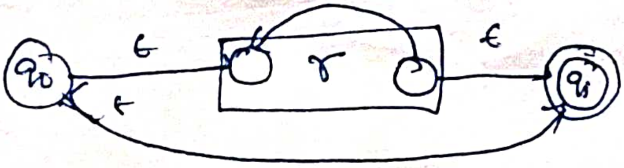
4) + (or) |

RE = rs





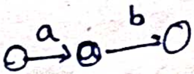
RE = r*



① construct the ε- NFA for the given regular Expression using Thompson's construction

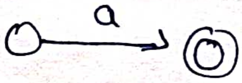
(a+b)*. ab

Sol

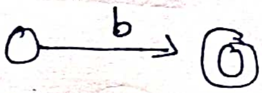


Step 1

a

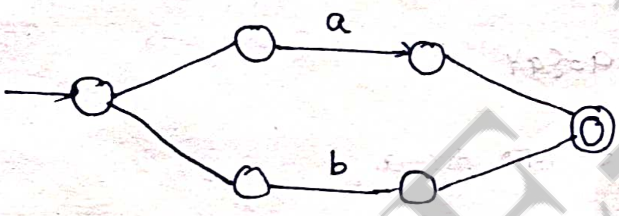


Step 2 b

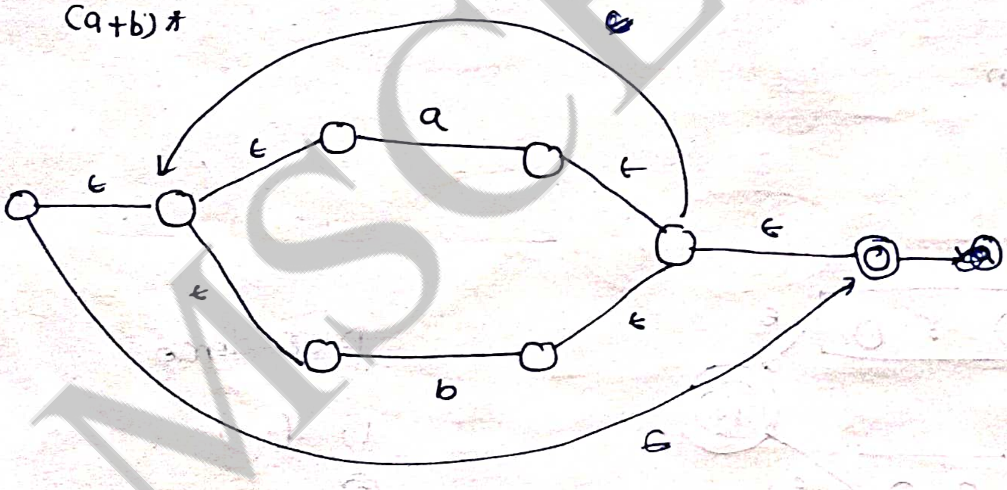


Step 3:

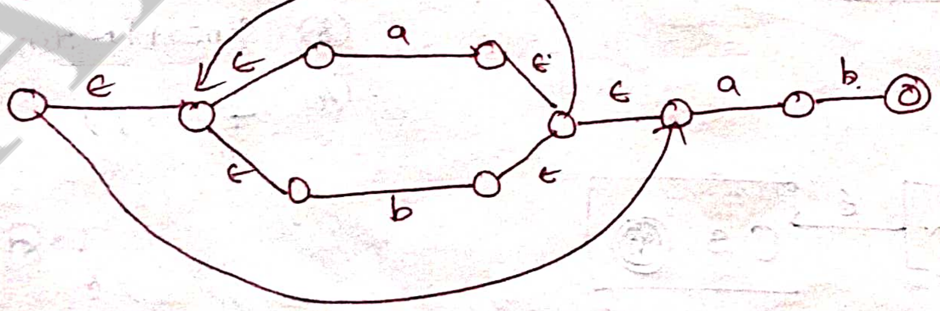
a+b



(a+b)*



(a+b)* ab

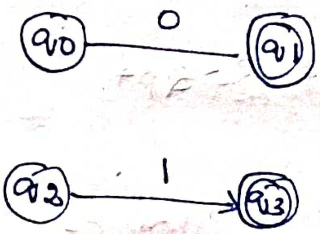


② b+ba*

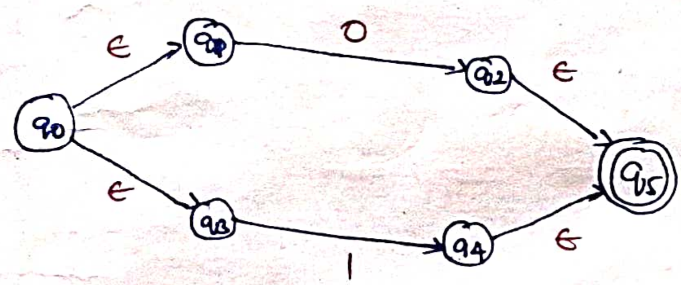
Convert the R.E to ε- NFA

1) 0+1

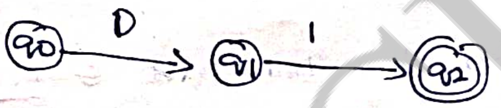
Sol



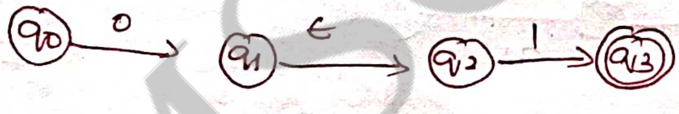
(0+1)



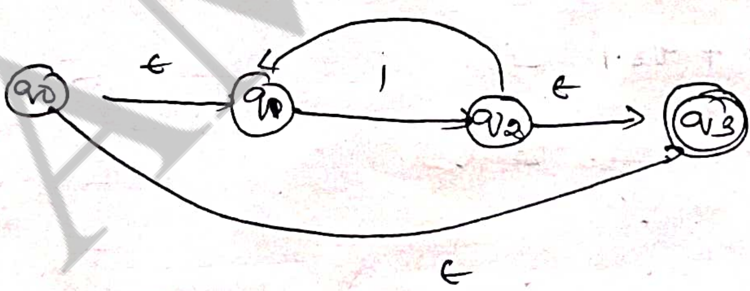
② 01



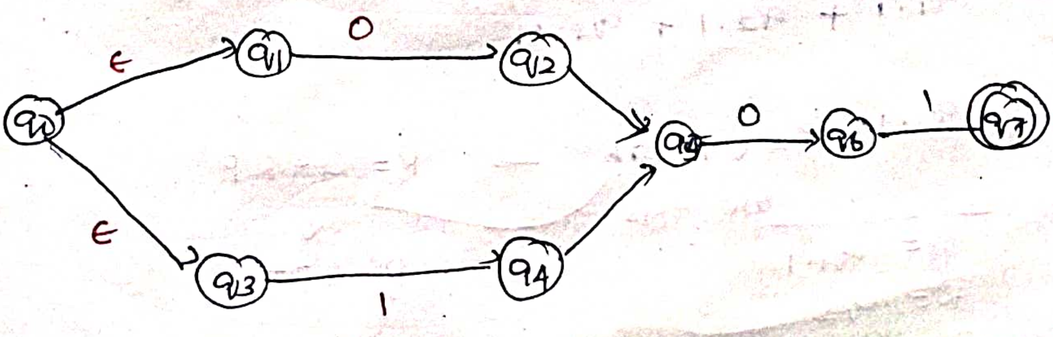
(or)



③ 1*



⑤ (0+1) 01

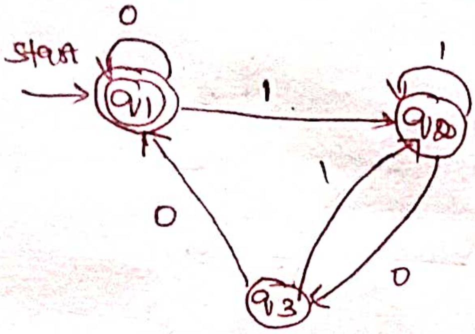


Arden's Theorem

① P & Q be two Regular expression over Σ If P does not contain ε, then the Equation in R

R= Q+RP has a solution (i.e) R=QP*

Construct Regular expression to the given FA (using Arden's theorem).



Step 1 :

(i) check Whether FA does not have ε-moves

(ii) It has only one start state

Step 2 :

Incoming of q1 as

$q_1 = q_0 \cdot 0 + q_0 3 + \epsilon$ — ①

$q_2 = q_1 \cdot 1 + q_2 \cdot 1 + q_3 \cdot 1$ — ②

$q_3 = q_2 \cdot 0$ — ③

③ in ②

$q_2 = q_1 \cdot 1 + q_2 \cdot 1 + q_2 \cdot 01$

$q_2 = q_1 \cdot 1 + q_2 (1+01)$ → $R = Q + RP$

we this

$\Downarrow$

$R = QP^*$

$Q = q_1 \cdot 1$

$R = q_2$

$P = (1+01)$

$q_2 = q_1 1(1+01)^*$ — ④

Now

$q_1 = q_1 \cdot 0 + q_3 \cdot 0 + \epsilon$ — ①

sub ③ in ①

$q_1 = q_1 \cdot 0 + q_2 \cdot 00 + \epsilon$ — ⑤

sub ④ in ⑤

$q_1 = q_1 \cdot 0 + q_1 \cdot 1(1+01)^* \cdot 00 + \epsilon$

Again apply arden ther

$R = Q + RP$

$\underbrace{q_1}_{R} = \underbrace{q_1}_{R} \underbrace{(0+1(1+01)^* 00)}_{P} + \underbrace{\epsilon}_{Q}$

$\boxed{\epsilon = 1}$

$q_1 = (0+1(1+01)^* 00)^*$

As q_1 is the only final state, the Regular Expression corresponding to given FA is

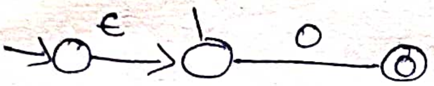
$RE = (0+1(1+01)^* 00)^*$

① construct a DFA with Reduced state equivalent to the Regular expression. R. E = 10+ (0+11)0*1

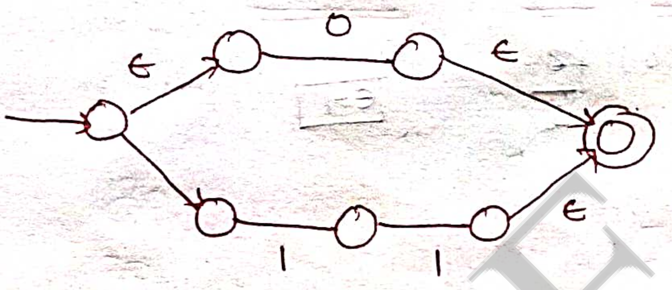
Sol

Step 1 (NFA with ε- transitions)

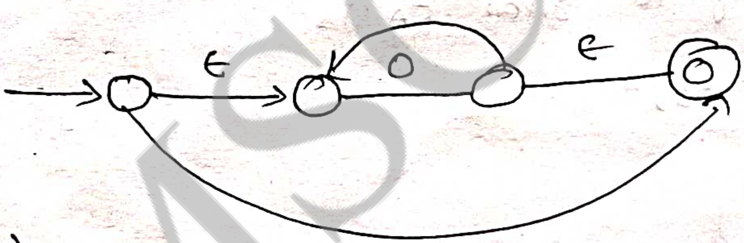
(i) The automaton for 10



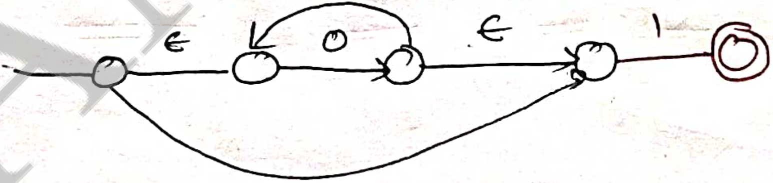
(ii) 0+11



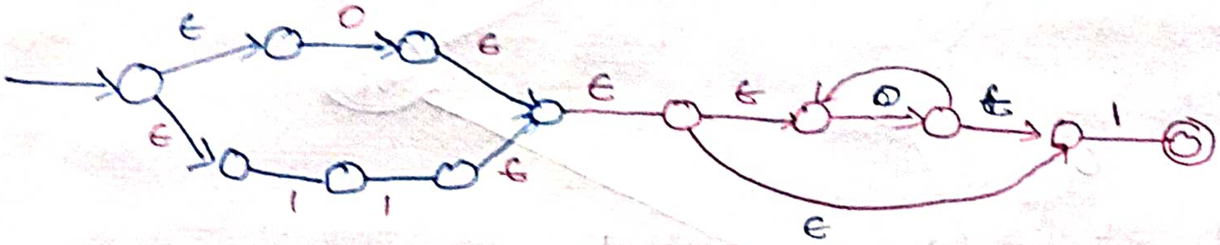
(iii) The automata for 0*



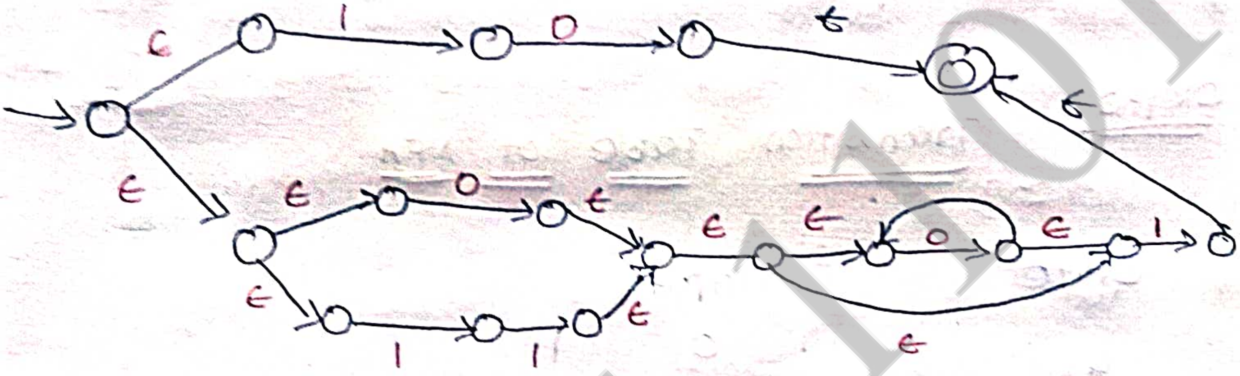
(iv) The automata for 0*1



(v) The automata for (0+11)0* 1

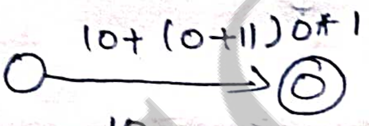


(vi) The automata for 10+ (0+11)0* 1

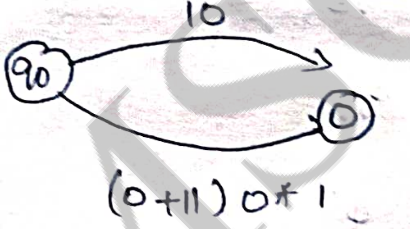


Step ② NFA without ε- moves

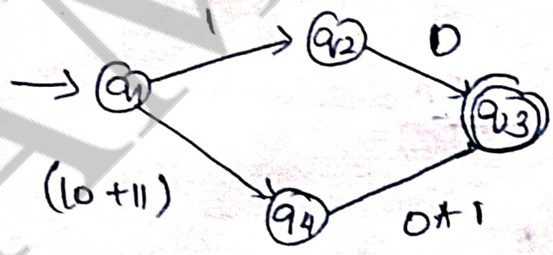
(i)



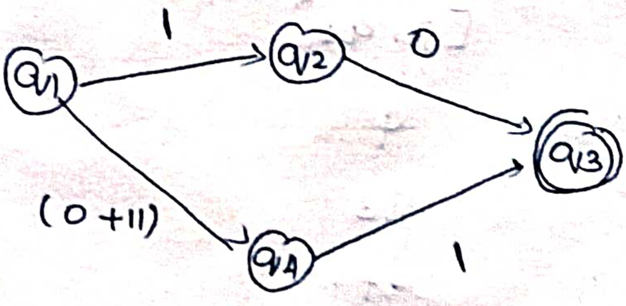
(ii)



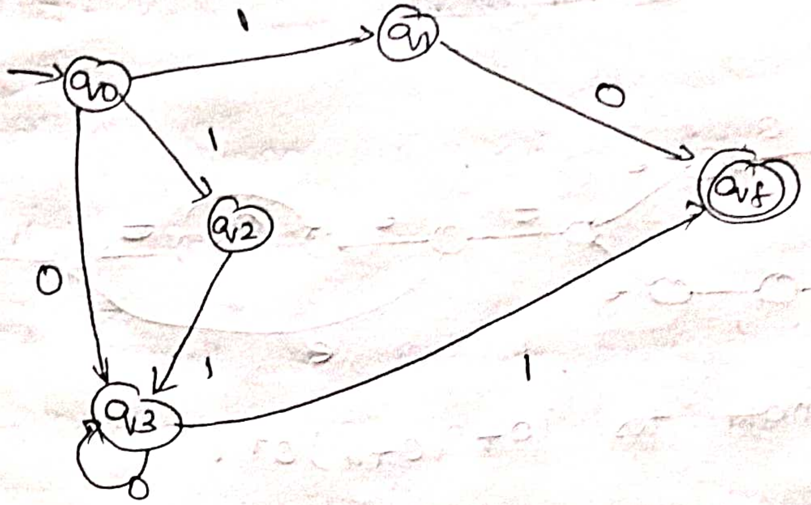
(iii)



(iv)



(v)



Step 3 :

Transition Table of NFA

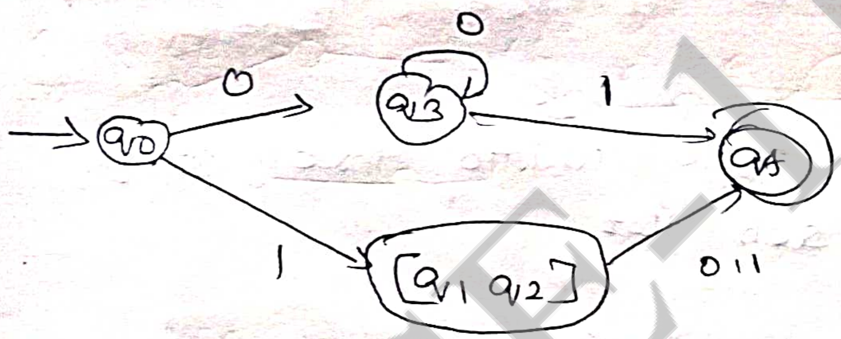
State	inputs	
	0	1
q0	q3	[q1, q2]
q1	qf	φ
q2	φ	q3
q3	q3	qf
qf	φ	φ

Transition table for DFA

State	0	1
[q0]	q3	[q1 q2]
q1	qf	φ
q2	φ	q3
q3	q3	qf
[q1 q3]	qf	qf
qf	φ	φ

State	0	1
q_0	q_3	$[q_1 q_2]$
q_3	q_3	q_f
$[q_1 q_2]$	q_f	q_f
q_f	ϕ	ϕ.

Transition Diagram of DFA



Proving Languages Not to be Regular (using Pum Lem

* Pumping lemma is used to prove that language is not Regular

* It cannot be used to prove that a language is Regular

* Let 'L' be a Regular Language then there exis a constant 'n' that for every string w in L

$|w| \geq n$.

* We can break w into three strings

$w = xyz$ such that

(i) $|y| > 0$ (or) $y \neq \epsilon$

(ii) $|xy| \leq n$.

(iii) for all $k \geq 0$ the string xy^kz is also in L.

Example 1

$L = \{a^n b^n \mid n \geq 1\}$ is not Regular using pumping Lemma.

$L = \{\epsilon, ab, aabb, aaabbb, aaaabbbb \ldots\}$

$w = aaabbb$ $\quad$ n= string length what ever you want

$|w| = 6 \geq n \Rightarrow |w| \geq 6 \quad 6 \geq 6$ true

Divide the string into three parts x, y z

$w = \underbrace{aa}_{x}\underbrace{ab}_{y}\underbrace{bb}_{z}$ $\quad$ $x = aa$
$y = ab$
$z = bb$

$|y| = ab$

(i) $|y| > 0$ $\quad |ab| > 0 \quad 2 > 0$ string length two true

(ii) $|xy| \leq n$ $\quad x = aa \quad y = ab \quad n = 6$

$|aaab| \leq 6 \quad 4 \leq 6 \quad aaab = 4$ string length

(iii) $xy^kz \quad k \geq 0$

$x = aa \quad y = ab \quad z = bb$

$xy^kz = \quad k = 0$

$aa(ab)^0(bb)$

$aabb \in L$ $\quad$ true

xy^kz, $K=1$

$\Rightarrow aa(ab)^1 bb$

$aaabbb \in L$ True String belongs to language

$\Rightarrow xy^kz$ $K=2$

$\Rightarrow aa(ab)^2 bb \Rightarrow aaababbb \notin L$ false

This is not valid string

$\therefore a^n b^n$ is not Regular Language.

16/9/2020

Unit-III Content Free Grammar and Languages

Grammar:

$G = (V, T, S, P)$

Where V = set of variables / Non-terminal symbols

T = set of terminal symbols

S = start symbol (S)

P = Production rule for T/N T symbols

Production Rule = $a \rightarrow \infty$ ($\because \underbrace{a, \alpha}$ (string) $\rightarrow V \cup T$)

Eg: $G = (\underbrace{\{S, A, B\}}_{V}, \underbrace{\{a, b\}}_{T}, S, \underbrace{\{S \rightarrow A.B,\ A \rightarrow a,\ B \rightarrow b\}}_{P.R})$

Grammar

Right linear gramar

↓

Production on Right side

$A \rightarrow XB$

$A \rightarrow X$

A, B → terminal symbols

X - Non-Terminal symbol

Left linear grammar

↓

Production on left side

$A \rightarrow BX$

$A \rightarrow X$.

Eg: $S \rightarrow ab\underline{S} / b$ → Right linear

$S \rightarrow \underline{S}bb / b$ → Left linear

$S \rightarrow abS$
$\rightarrow ab\underline{b}$

$S \rightarrow Sbb$
$\rightarrow \underline{b}bb$

Derivation
↓
set of all string
↓
Grammar
↓
Language

1) $G = (\{S, A\}, \{a, b\}, S, \{S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow e\})$

$S \rightarrow \underline{aA}b$ By $aA \rightarrow aaAb$

$\rightarrow a\underline{aA}bb$

$\rightarrow aa\underline{aA}bbb$ By $A \rightarrow e$

$\rightarrow aaaebbb$

$\rightarrow aaabbb$

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

$S \rightarrow \underline{A}B$ By $A \rightarrow a$

$\rightarrow a\underline{B}$ By $B \rightarrow b$

$\rightarrow ab$

3) $(\{S, A, B\}, \{a, b\}, S, \{S \to AB,\ A \to aA/a,\ B \to bB/b\})$

$S \to AB$

$\to aB$ (by $A \to a$)

$\to ab$ (by $B \to b$)

$S \to AB$

$\to aAB$ ($A \to aA$)

$\to aaB$ ($A \to a$)

$\to aab$ ($B \to b$)

$\to a^2b$

$S \to AB$

$\to aAB$ (by $A \to aA$)

$\to aAbB$ (By $B \to bB$)

$\to aabB$ (By $A \to a$)

$\to aabb$ (By $B \to b$)

$\to a^2b^2$

$S \to AB$

$\to AbB$ (by $B \to bB$)

$\to abB$ (by $A \to a$)

$\to abb$ (by $B \to b$))

$\to ab^2$

$L(G) = \{ab, a^2b^2, a^2b, ab^2 \dots\}$

$= \{a^m b^n \mid m \geq 0\ \&\ n \geq 0\}$

Context Free Grammar

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

Eg:- Language of Palindrome → Lpal

$\omega = \omega^R$

Eg: 0110, 11011, 101.

Basis : ϵ, 0, 1

Induction : ω Eg: $0\omega 0$, $1\omega 1$.

→ Recursive def

CFL → CFG

Eg: Palendrome rules

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0w0$
5. $P \rightarrow 1w1$

Context free grammar Example:

1) $a^n b^n$ (n should be equal for both a and b)

$G = \{(S, A), (a, b)\ (S \rightarrow aAb,\ A \rightarrow aAb \mid \epsilon)\}$

$S \rightarrow aAb$

$\rightarrow aaAbb$ (by $A \rightarrow aAb$)

$\rightarrow aaaAbbb$ (by $A \rightarrow aAb$)

$\rightarrow aaa\epsilon bbb$ (by $A \rightarrow \epsilon$)

$\rightarrow aaabbb$

$\rightarrow a^3 b^3$.

$L(G) = \{a^n b^n / n > 0\}$.

Parse tree

→ ordered root tree

→ semantic information of strings derived from CFG

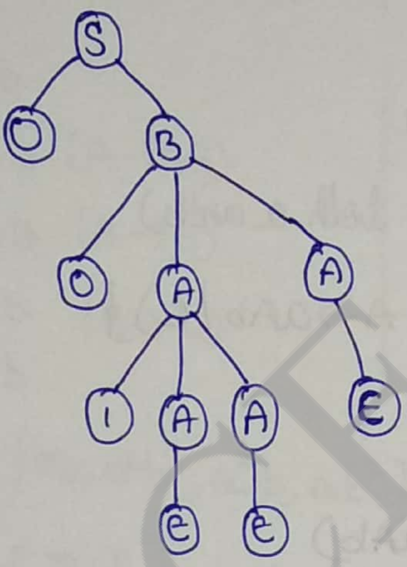
1) G = {V, T, P, S} where S → 0B , A → 1AA / ε , B → 0AA

Rules:

Root Vertex : Start symbol

Vertex : Non-terminal symbol

leaves : Labelled by terminal symbol.

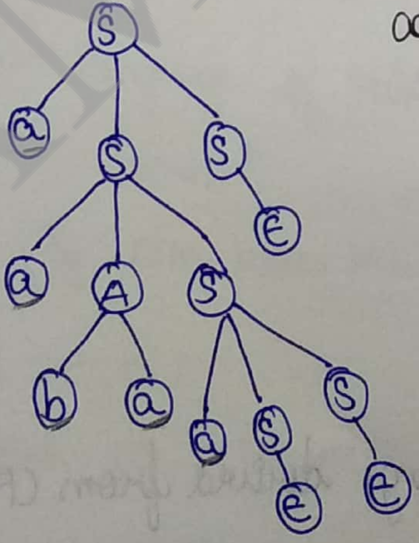


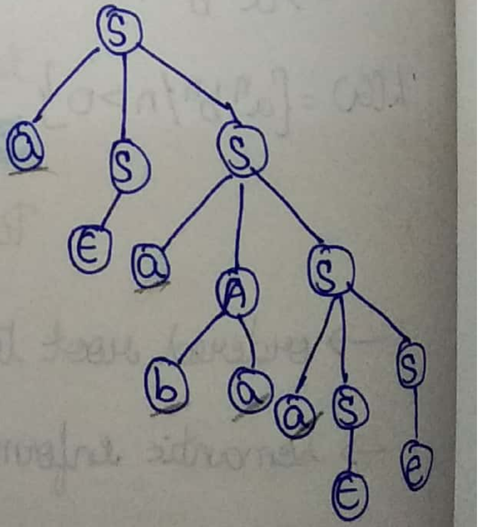
2) G = {V, T, P, S} where S → aAS / aSS / ε , A → SbA / ba .

Left derivation tree

aabaa

Right Derivation tree





Ambiguous grammar

two / more derivation tree

↳ string ω

Eg: 2 left derivation tree

1) $G = (\{S\}, \{a+b, +, *\}, P, S\}$ where P consists of $S \to S+S \,/\, S*S \,/\, a \,/\, b$

string $a+a*b$.

$S \to S+S$	$S \to S*S$
$\to S+S*S$ (by $S \to S*S$)	$\to S+S*S$ (by $S \to S+S$)
$\to a+a*S$ (by $S \to a$)	$\to a+a*S$ (by $S \to a$)
$\to a+a*b$ (by $S \to b$)	$\to a+a*b$ (by $S \to b$)

|9|2020

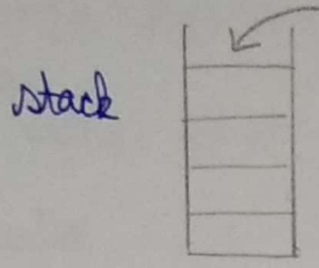
Push down automata:

A PDA is a way to implement a context free grammar in a similar way we design finite automata for regular grammar.

* It is more powerful than FSM.

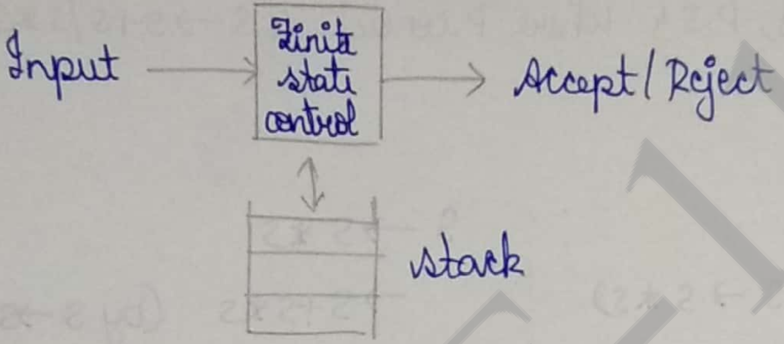
* FSM has very limited memory but PDA has more memory

* PDA = finite state machine + A stack.



Push - A new element is added to the topof stack

POP - The top element of the stack is read + removed.



A PDA has 3 components

1. An i/p tape
2. A finite control unit
3. A stack with infinite size.

A PDA is defined by 7 tupes as show below

$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

where,

Q = A finite set of states

Σ = A finite set of input symbols

Γ = A finite stack alphabet

δ = The transition function.

q_0 = start state

Z_0 = start stack symbol

F = set of final/accepting states

δ takes as argument a triple δ(q, a, X) where

(i) q is a state in Q.

(ii) a is either as i/p symbol in Σ or a = ε.

(iii) X is a stack symbol, ie is a member of F.

The O/P of δ is a finite set of pairs (P, V) where:

* P is a new state

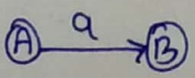
* Y is a string of stack symbols that replaces X at the top of the stack.

Eg: If Y = ε then stack is popped.

If Y = X then the stack is unchanged.

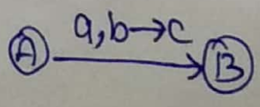
If Y = YZ then X is replaced by Z and Y is pushed onto the stack.

Finite state machine

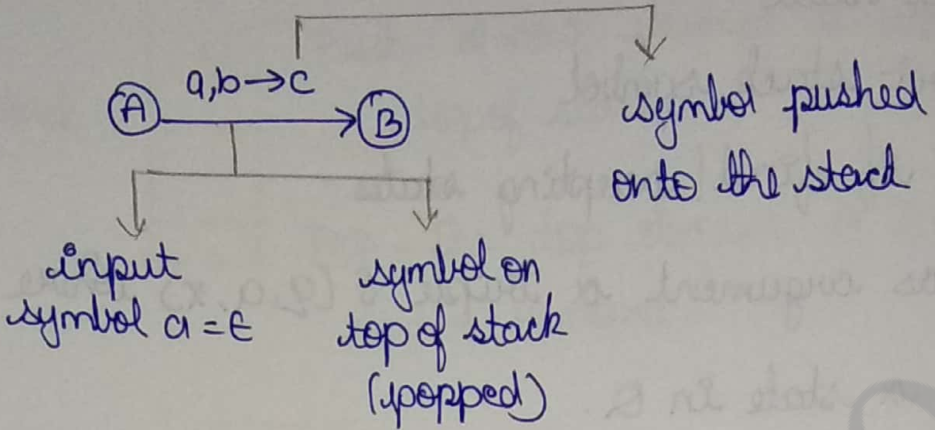


limited memory

PDA



expanded memory.

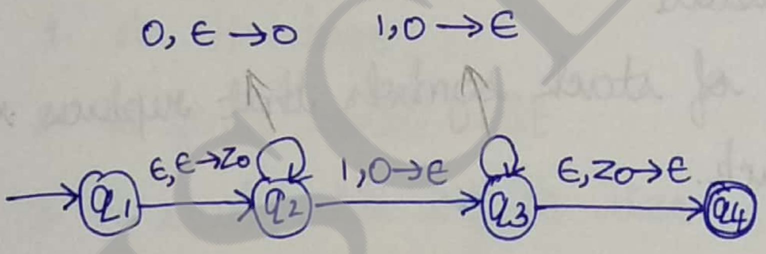


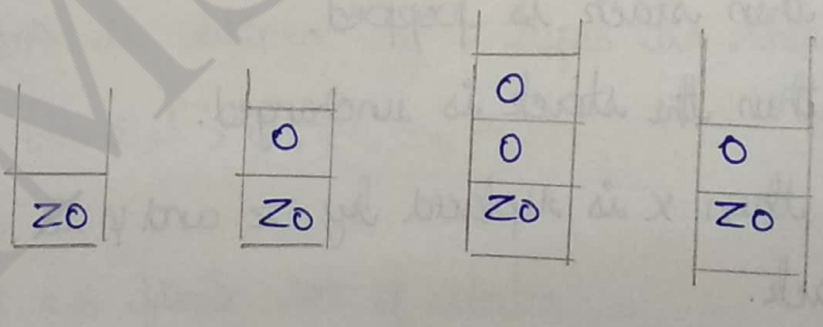
ε → stack → it is neither read or popped

→ stack → Nothing is pushed.

Eg: Construct a PDA that accepts (a011)

$L = \{a^n 1^n \mid n \geq 0\}$





0|2020

Equivalence of CFG and PDA

Theorem: A language is context free if some push down automata recognizes it.

Proof: ① Given CFG, show how to construct a PDA that recognizes it.

② Given a PDA, show how to construct a CFG that recognizes the same language.

Given a grammar

$S \rightarrow BS | A$

$A \rightarrow 0A | \epsilon$

$B \rightarrow BB1 | 2$ Find or build a PDA

Left most derivation:

$\rightarrow S$

$\rightarrow BS$

$\rightarrow BB1S$

$\rightarrow 2B1S$

$\rightarrow 221S$

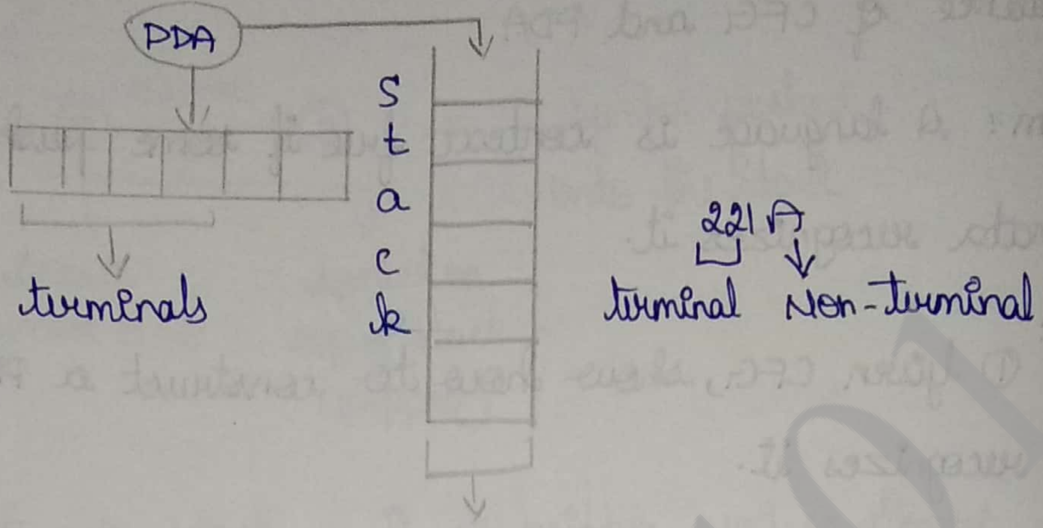
$\rightarrow 221A$

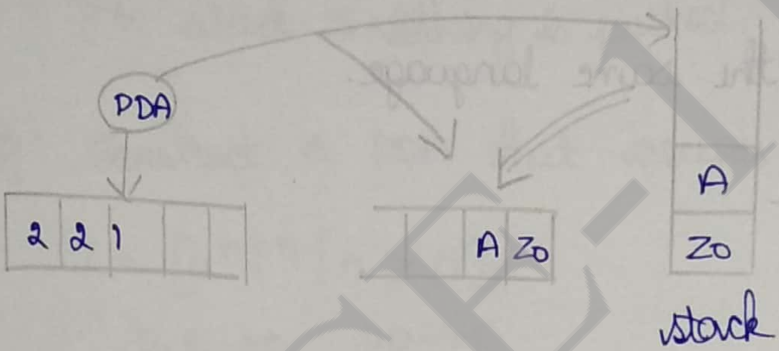
$\rightarrow 221\epsilon$

$\rightarrow 221 \rightarrow$ (left sentential form)

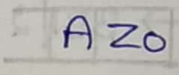
221A (any production)

$\underbrace{221}_{\text{terminal symbol}}\underbrace{A}_{\text{Non-terminal symbol.}}$



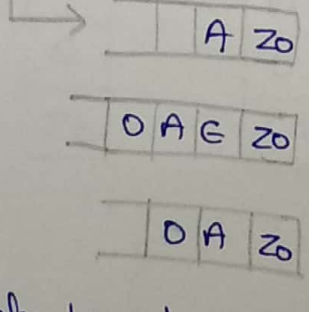


Left most derivation $S \longrightarrow 221$



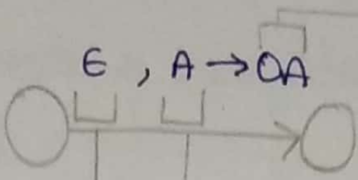
At each step expand left most derivation

Eg: $A \longrightarrow 0A|\epsilon$



* Match stack top to a rule
* Pop stack
* Push right hand side of rule onto stack.

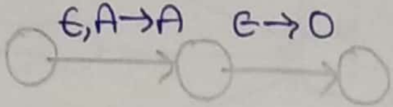
Rule A → 0A Add this to PDA



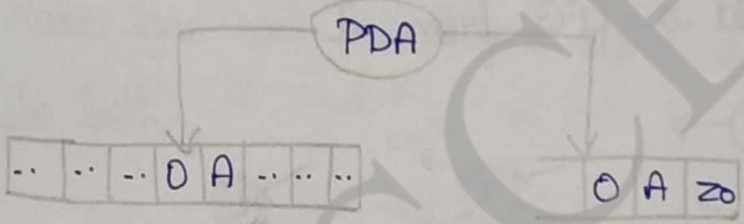
→ Right hand side pushed onto the stack.

→ Match the top of stack & Pop.

→ No i/p at initial stage.



Eg: Rule A → 0A



Terminal symbol encountered

→ Match it

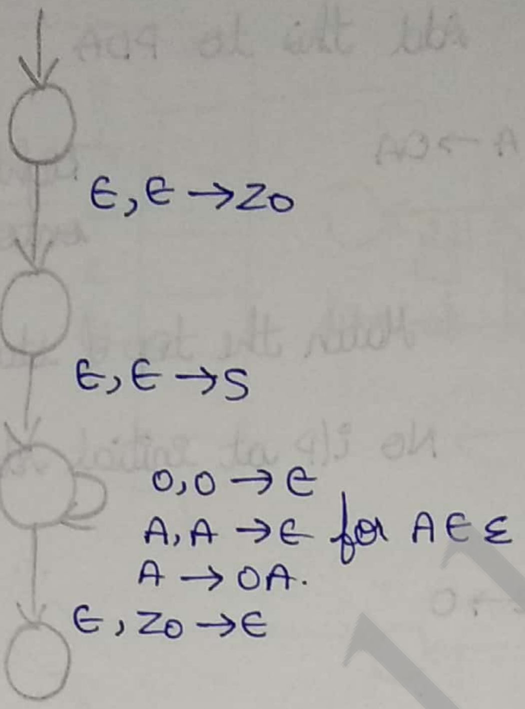
→ Pop it

→ Advance it

PDA design terminal $\begin{cases} 0,0 \to \epsilon \\ 1,1 \to \epsilon \\ z,z \to \epsilon \end{cases}$

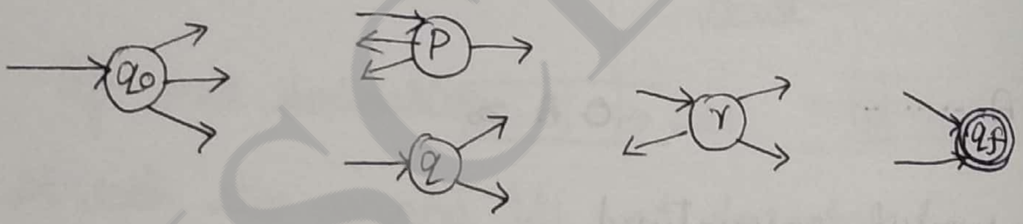
Non terminal $\begin{cases} A,A \to \epsilon \\ \quad \text{for all } A \in \Sigma. \end{cases}$

Final PDA



5/10/2020

1) Given a PDA --> Build a CFG from it

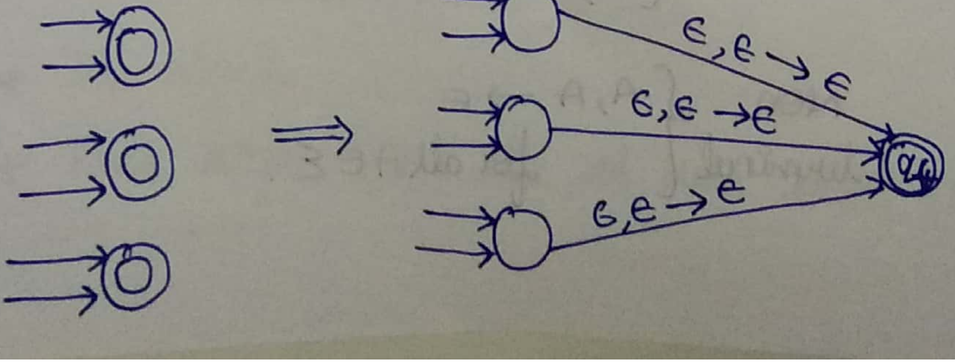


step 1: simplify the PDA

step 2: Build a CFG

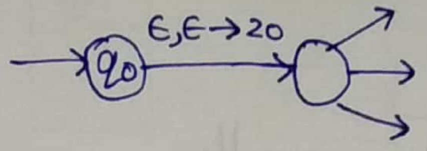
starting non-terminal $= A_{q_0 q_f}$

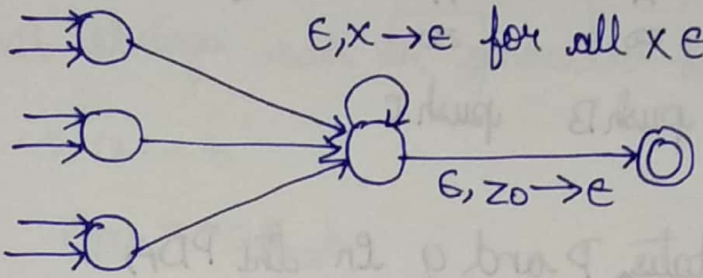
other non-terminal states: $A_{pq}, A_{qr}, A_{rq_0}, \ldots$



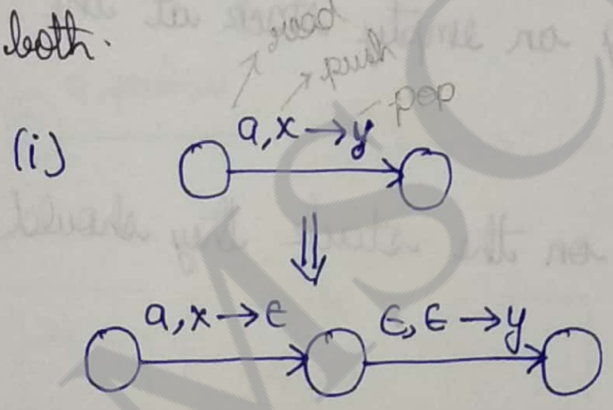
2) The PDA should empty its stack before accepting.

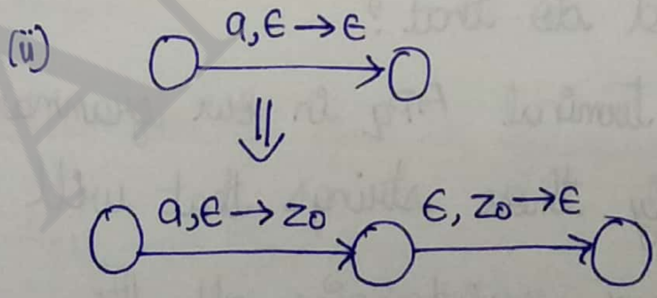
→ create a new start state q0 which put z0 to the stack



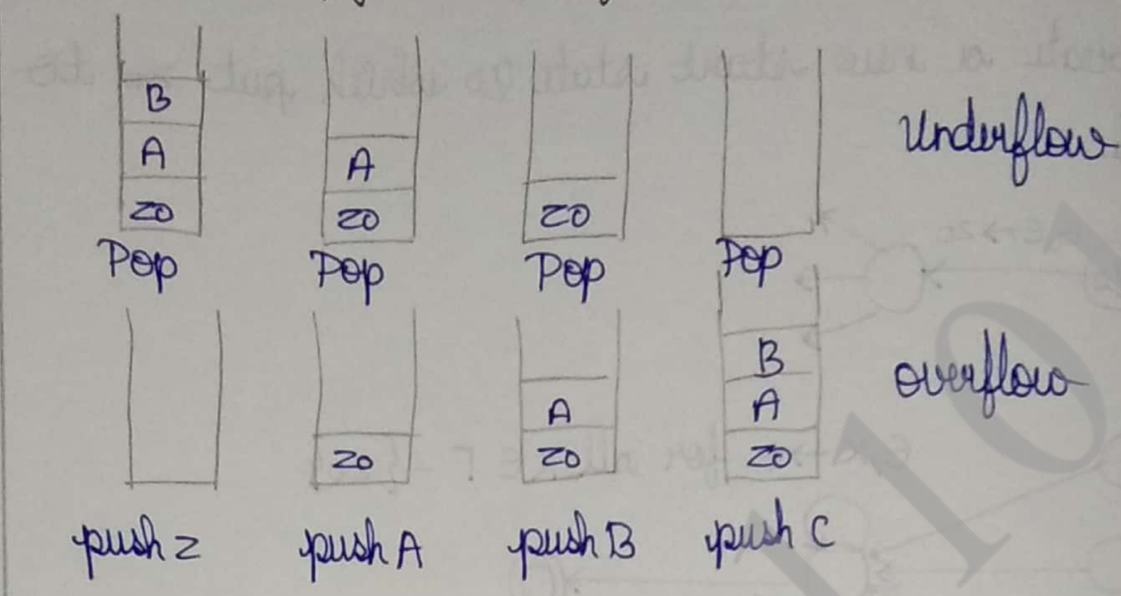


3) Make sure each transitions either pushes or pops but does not do both.





start with empty stack & fenish with an empty stack.



consider two states P and q in the PDA

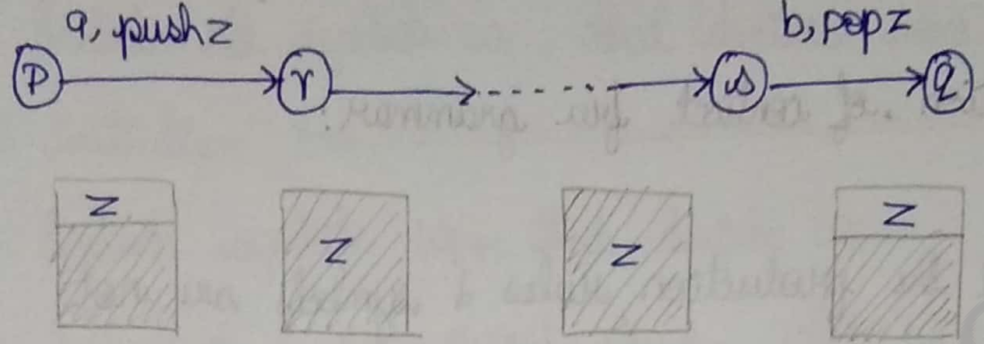
→ could we go from P to q without stack underflow and maintaing an empty stack at the beginning and end?

→ If something already on the stack they should not be changed.

What strings would do that?

We will a Non-terminal Apq in our grammar. Apq will generate exactly those strings that will take us from p to q maintaining all the above stack conditions.

case 1 :



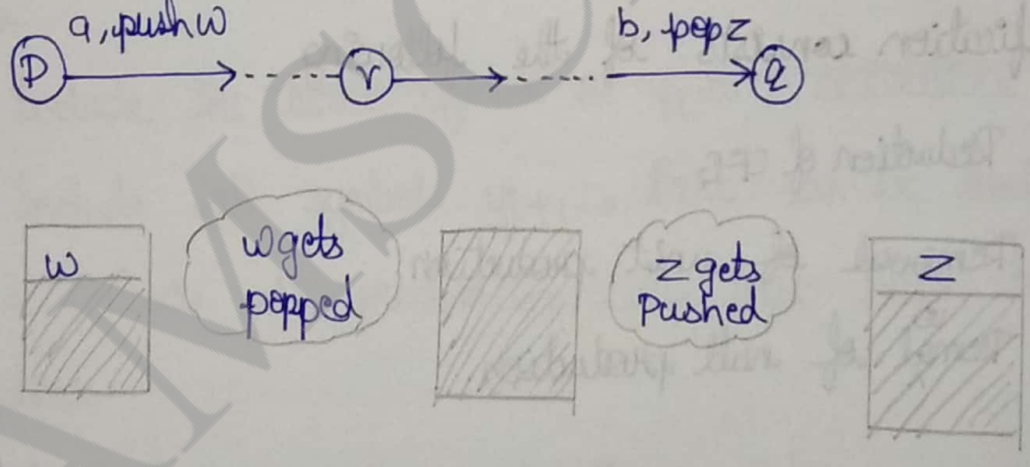
what strings can be generated by following this path ?

--> "a --- b"

$A_{pq} \rightarrow aA_{rs}b$

→ This rule will generate exactly those strings.

case 2 :



what strings can be generated by following this path ?

$A_{pq} \longrightarrow A_{pr} A_{rq}$

--> This rule will generate exactly those strings.

8/10/2020

Unit - IV Properties of Context free languages

simplification of context free grammar:

In CFG

→ all the production rules & symbols are not needed for the derivation of strings.

→ Null production & unit production are also found

5 Normal form - Elimination of these production and symbols is called simplification of CFG.

simplification consists of the following

① Reduction of CFG.

② Removal of unit production.

③ Removal of null production.

1. Reduction of CFG

CFG are reduced in two phases

Phase 1 : Derivation of an equivalent grammar G', from the CFG, G such that each variable derives some terminal string.

steps:

1. Include all symbols W_1, that derives some terminal and initialize $i=1$.

2. Include symbols W_{i+1} that derives W_i

3. Increment i and repeat step 2, until $W_{i+1} = W_i$

4. Include all production rules that have W_i in it.

<u>Phase 2:</u> Derivation of an equivalent grammar G'', from the CFG, G', such that each symbol appears in an sentential form.

steps:

1. Include the start symbol in Y_1 and initialize $i=1$.

2. Include all symbols Y_{i+1}, that can be derived from Y_i & include all production rules that have been applied.

3. Increment i and repeat step 2 until $Y_{i+1} = Y_i$

<u>Example</u>: Find a reduced grammar equivalent to the grammar G, having production rules

$P: S \rightarrow AC \mid B$, $A \rightarrow a$, $C \rightarrow c \mid BC$, $E \rightarrow aA/e$.

solution :

Part 1: $T = \{a, c, e\}$

$W_1 = \{A, C, E\}$

$W_2 = \{A, C, E, S\}$

$W_3 = \{A, C, E, S\}$

$G' = \{(A, C, E, S), \{a, c, e\}, P, (S)\}$

$P = \{S \rightarrow AC, A \rightarrow a, C \rightarrow c, E \rightarrow aA|e$

Part 2: $Y_1 = \{S\}$

$Y_2 = \{S, A, C\}$

$Y_3 = \{S, A, C, a, c\}$

$X_4 = \{S, A, C, a, c\}$

$G'' = \{(A, C, S), \{a, c\}, P, \{S\}\}$

$P : S \rightarrow AC, A \rightarrow a, C \rightarrow c.$

2. <u>Removal of unit Production:-</u>

Any production rule of the form $A \rightarrow B$, where $A, B \in$ Non terminals is called unit production.

<u>Procedure for removal:</u>

S1 → To remove $A \rightarrow B$, add production A that gives X to the grammar rule whenever

$B \rightarrow x$ occurs in the grammar

[$x \in$ terminal x can be Null]

S2 → Delete $A \rightarrow B$ from the grammar.

S3 → Repeat from step 1 until all unit production are removed.

$Z \rightarrow M$

Non-terminal symbol Non-termi symbol

Example:

1) Remove unit production from the grammar whose production rule is given by

P: $S \rightarrow XY$; $X \rightarrow a$, $Y \rightarrow Z|b$, $Z \rightarrow M$, $M \rightarrow N$, $N \rightarrow a$.

$Y \rightarrow Z$, $Z \rightarrow M$, $M \rightarrow N$.

① since $N \rightarrow a$ we add $M \rightarrow a$

P: $S \rightarrow XY$, $X \rightarrow a$, $Y \rightarrow Z|b$, $Z \rightarrow M$, $M \rightarrow a$, $N \rightarrow a$.

② since $M \rightarrow a$, we add $Z \rightarrow a$

P: $S \rightarrow XY$, $X \rightarrow a$, $Y \rightarrow Z|b$, $Z \rightarrow a$, $M \rightarrow a$, $N \rightarrow a$.

③ since $Z \rightarrow a$, we add $Y \rightarrow a$

P: $S \rightarrow XY$, $X \rightarrow a$, $Y \rightarrow a|b$, $Z \rightarrow a$, $M \rightarrow a$, $N \rightarrow a$.

Remove the unreachable symbols :- P: $S \rightarrow XY$, $X \rightarrow a$, $Y \rightarrow a|b$.

10/10/2020

3. Removal of Null Production:-

Procedure for Removal :

step 1: To remove $A \to \epsilon$, look for all productions whose right side contains A.

step 2: Replace each occureness of 'A' in each of these productions with ϵ.

step 3: Add the resultant productions to the grammar

Example :

1) Remove null production from the following grammar

$S \to ABAC$, $A \to aA/\epsilon$, $B \to bB/\epsilon$, $C \to c$,

$A \to \epsilon$, $B \to \epsilon$

I) To eliminate $A \to \epsilon$

$S \to ABAC$

$S \to ABC / BAC / BC$

$A \to aA$

$A \to a$

New production : $S \to ABAC / ABC / BAC / BC$

$A \to aA / a$, $B \to bB / \epsilon$, $C \to c$.

2) To eliminat $B \to \epsilon$.

$S \to AAC | AC | C$, $B \to b$

New production : $S \to ABAC | ABC | BAC | BC | AAC | AC | C$

$A \to aA | a$, $B \to bB$.

12/10/20

Pumping Lemma
For context free languages.

Pumping Lemma (for CFL) is used to prove that a language is not context free.

If A is a context free language, then A has a pumping length 'P' such that any string 'S', where $|S| \geq P$ may be divided into 5 parts $S = uvxyz$ such that the following conditions must be true:

(1) $u v^i x y^i z$ is in A for every $i \geq 0$

(2) $|vy| > 0$

(3) $|vxy| \leq P$.

To prove that a language is not context free using Pumping lemma

1) Assume A is context free.

2) It has to have a pumping length (say P)

3) All strings longer than P can be pumped $|S| \geq P$.

4) Now find a string 's' in A such that $|S| \geq P$.

5) Divide s into uvxyz.

6) show that $uv^ixy^iz \notin A$ for some i

7) The consider the ways that s can be divided into uvxyz.

8) show that none of these can satisfy all the 3 pumping conditions at the same time

9) s cannot be pumped == contradiction.

Example : 1) show that $L=\{a^Nb^Nc^N \mid N \geq 0\}$ is Not.

(i) Assume that L is context free

(ii) L must have pumping length (say P)

(iii) Now we take a string s such that $s = a^Pb^Pc^P$.

(iv) we divide s into parts uvxyz.

Eg: P=4, so $a^4 b^4 c^4$.

case 1: v and y each contain only one type of symbol.

$$\underbrace{a}_{u}\ \underbrace{aa}_{v}\ \underbrace{abbbb}_{x}\ \underbrace{c}_{y}\underbrace{ccc}_{z}$$

$uv^i x y^i z$ $\quad i=2$

$uv^2 x y^2 z$

aaaa aa bbbb cc ccc

$a^6 b^4 c^5 \notin L$

case 2: Either v or y has more that one kind of symbols.

$$\underbrace{aa}_{u}\ \underbrace{aabb}_{v}\ \underbrace{b}_{x}\ \underbrace{b}_{y}\ \underbrace{cccc}_{z}$$

$uv^i x y^i z$ $\quad (i=2)$

$uv^2 x y^2 z$

aaaabbaabbbbbcccc

$a^N b^N c^N \notin L$.

15/10/2020

Closure Properties of context free languages.

① substitutions

② Application of substitution

Theorem: * Union

* Concatenation

* Closure (*) & Positive closure (+)

* Homomorphism

③ Reversal ⑤ Inverse homomorphism.

④ Intersection with Regular language

① substitutions:

$\Sigma \rightarrow a$

(alphabet) (symbol)

La as s(a) for each symbol a

$s(w) = s(a_1) \cdot s(a_2) \dots s(a_n)$

where

$w = a_1 a_2 \dots a_n$

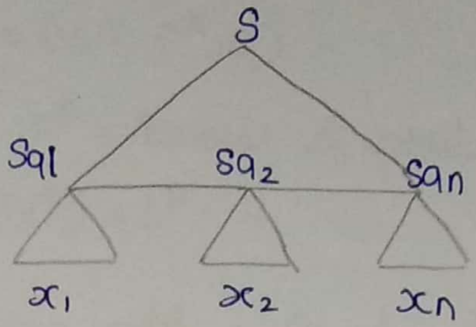
$s = x_1 x_2 \dots x_n = a_i$

$s(a_i)$

where $i = 1, 2, \dots n$.

$s(L)$ = Union of $s(w)$

for $\forall$ w in L



Theorem: The CFL are closed under the following operation.

1. Union
2. concatenation
3. closure (*) & positive (+)
4. Homomorphism.

Proof:

* Proper substitution
* From one CFL to another
* Produced CFL'

1. Union:

$s(L) = L_1 \cup L_2$

where $L = \{1, 2\}$

$s(1) = L_1$ & $s(2) = L_2$

2. Concatenation:

$s(L) = L_1 . L_2$

Where $L = \{12\}$

$s(1) = L_1$ & $s(2) = L_2$

3. Colosure & positive closure:

L_1 is CFL

where $L = \{1\}^*$

$s(1) = L_1$

$s(L) = L_1^*$

where $L = \{1\}^+$

$s(1) = L_1$

$s(L) = L_1^+$

4. Homomorphism:

$L \rightarrow$ CFL over alphabet Σ

$h \rightarrow$ homomorphism on Σ

$s \rightarrow h$

$s(a) = \{h(a)\}$

for all a in Σ

$\therefore s(L) = h(L)$

③ Reversal:

CFL's are also closed under reversal

No substitution method is used.

Theorem: If L is a CFL then so is L^R

Proof: $L = L(G)$

where some CFL $G = (V, T, P, S)$

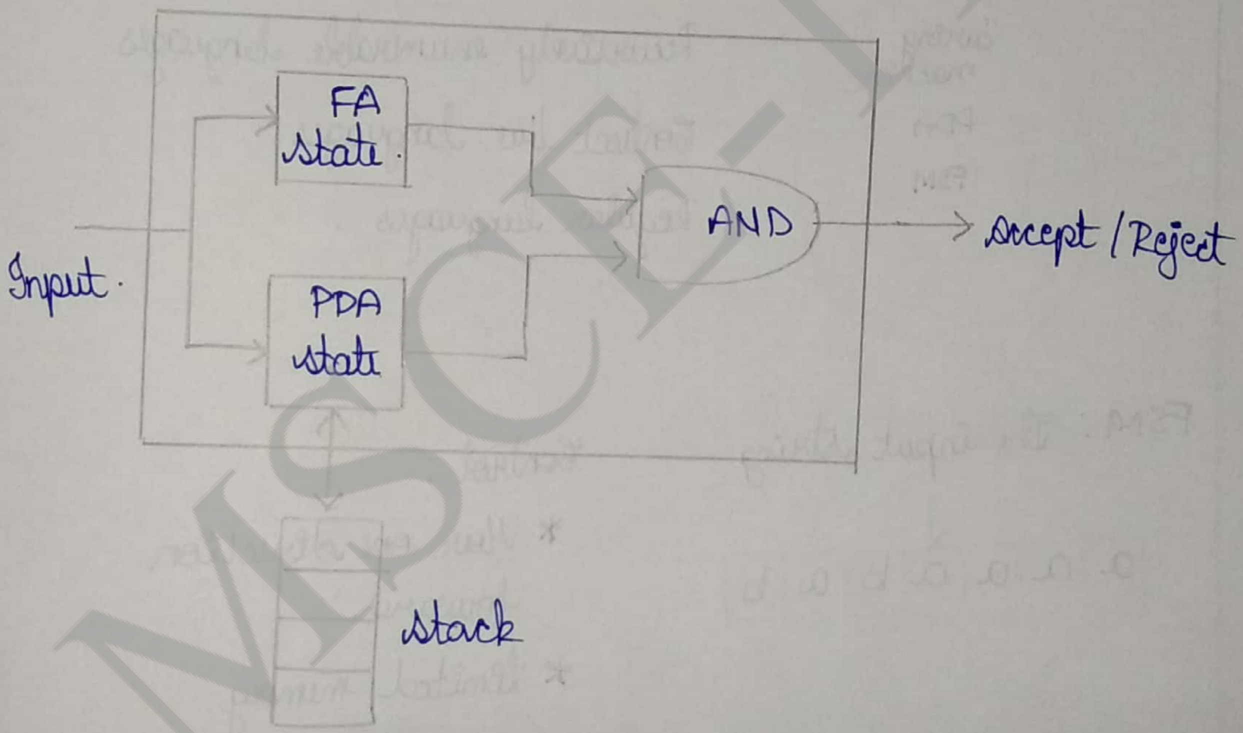
$G^R = (V, T, P^R, S)$

Where P^R = Reverse of production

$L(G^R) = L^R$

④ Intersection with a Regular language :

Theorem: If L is a CFL + R is a regular language, then $L \cap R$ is a CFL.



Proof :

$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$

$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$

$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta(q_P, q_A), Z_0, F_P \times F_A)$

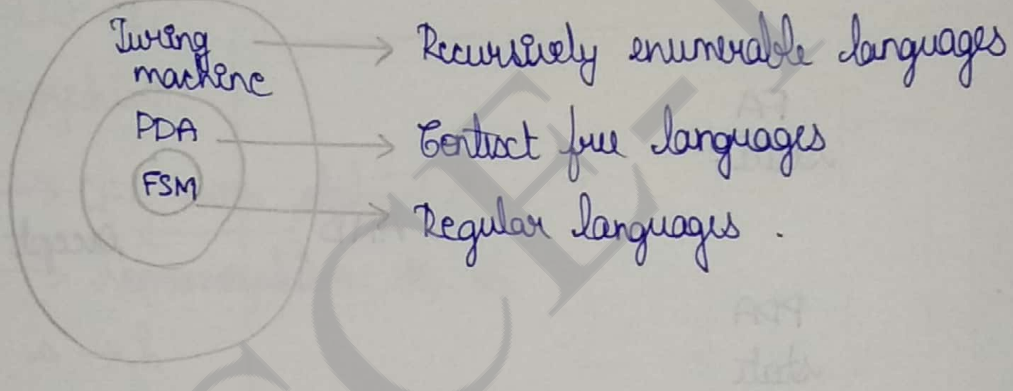
$\delta((q,p),a,x) \rightarrow ((r,s),\gamma)$ is defined to set of all pairs such that

1. $s = \hat{\delta}_A(p,a)$

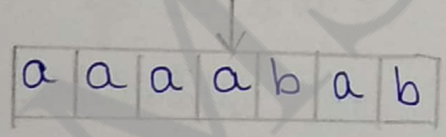
2. pair (r,γ) is in $\delta_P(q,a,X)$

19/10/2020

Turing machine :



FSM : The input string



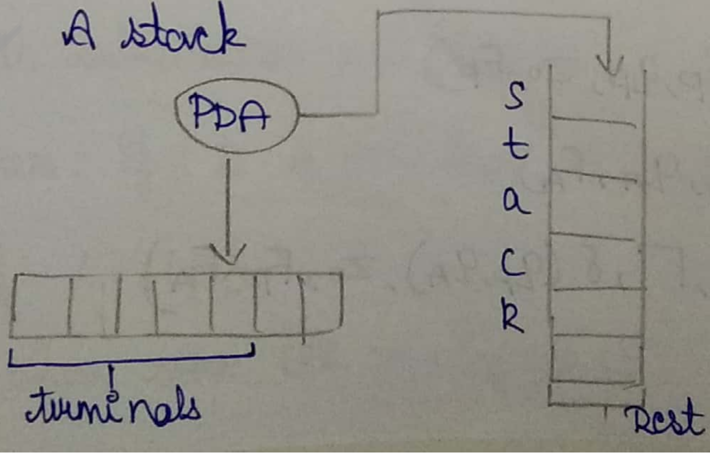
Control :

* Move one direction forward

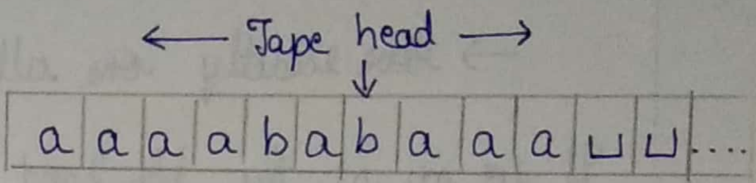
* Limited memory

PDA : The Input string

A stack



Turing machine :



A tape

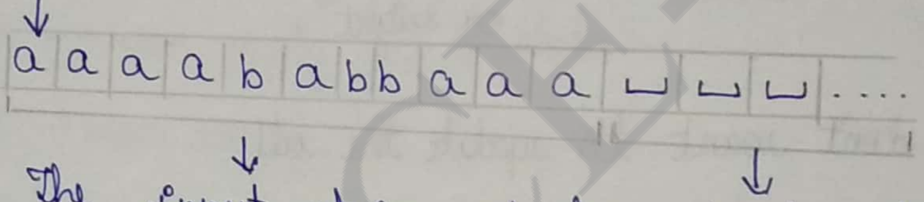
↳ sequence of infinite symbols

1) Tape alphabets : $\Sigma = \{0, 1, a, b, x, z_0\}$

2) The Blank ⊔ is a special symbol

→ It is used to fill the infinite tape does not belong to Σ.

Initial configuration :



The input string Blanks out to infinity.

Operations on the tape :

→ Read / scan symbol below the tape head

→ update / Write a symbol below the tape head.

Rules of operation 1 :

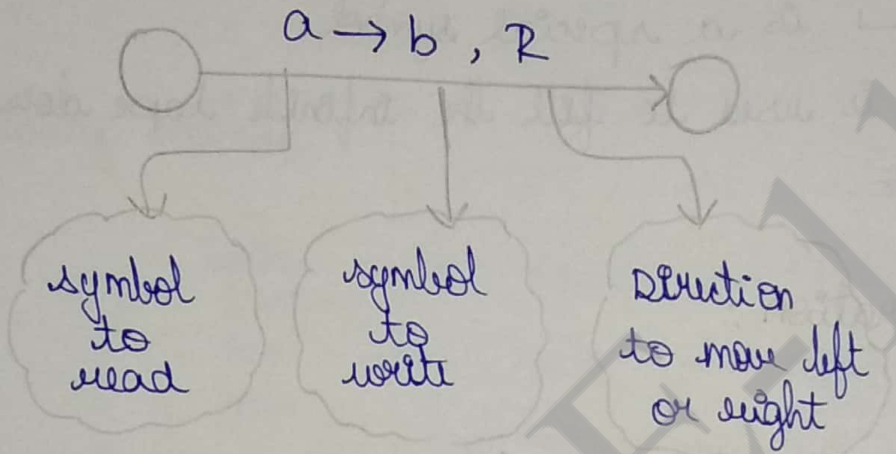
At each step of computation.

→ Read the current symbol

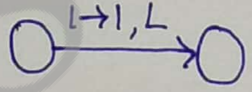
→ update (ie. write) the same cell.

→ Move excactly one cell either left or right.

If we are at the left hand (end) of the tape and trying to move left, then do not move. stay at the left end



If you don't want to update the cell, Just write the same symbol



Rules of operation 2:

→ control is with a sort of FSM

→ Initial state

→ Final states : (there are two final states)

1) The accept state

2) The reject state.

→ computation can either

1) HALT and accept

2) HALT and reject.

3) LOOP (the machine fails to HALT).

22/10/2020

Turing machine:

A turing machine is defined with 7 tuples

$(Q, \Sigma, \Gamma, \delta, q_0, b, F)$

Q → Non empty set of states

Σ → Non empty set of symbols

Γ → Non empty set of tape symbols.

δ → Transition function defined as

$$Q \times \Sigma \rightarrow \Gamma \times (R/L) \times Q$$

q_0 → Initial state

b → Blank symbol

F → set of final states (Accept state & Reject state)

Thus, the production rule of turing machine will be written as

$$\cancel{\delta}\ \delta(q_0, a) \rightarrow (q_1, y, R)$$

Turing's Thesis :

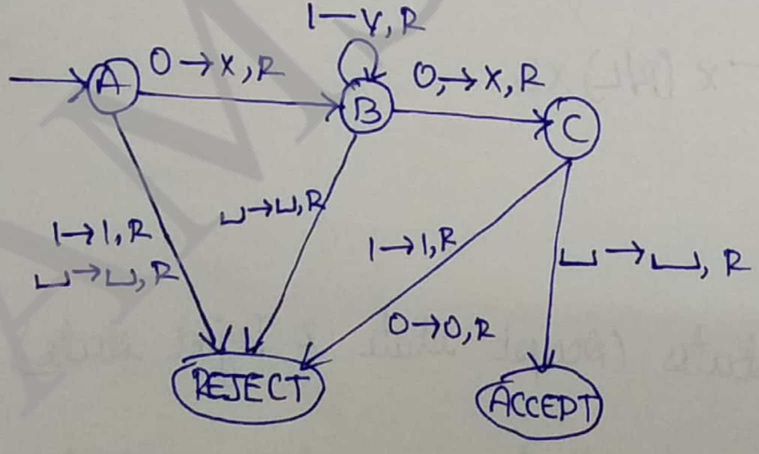
Turing's thesis states that any computation that can be carried out by mechanical means can be performed by some turing machine.

Few arguments for accepting this thesis are :

(i) Anything that can be done on existing digital computer can also done by turing machine.

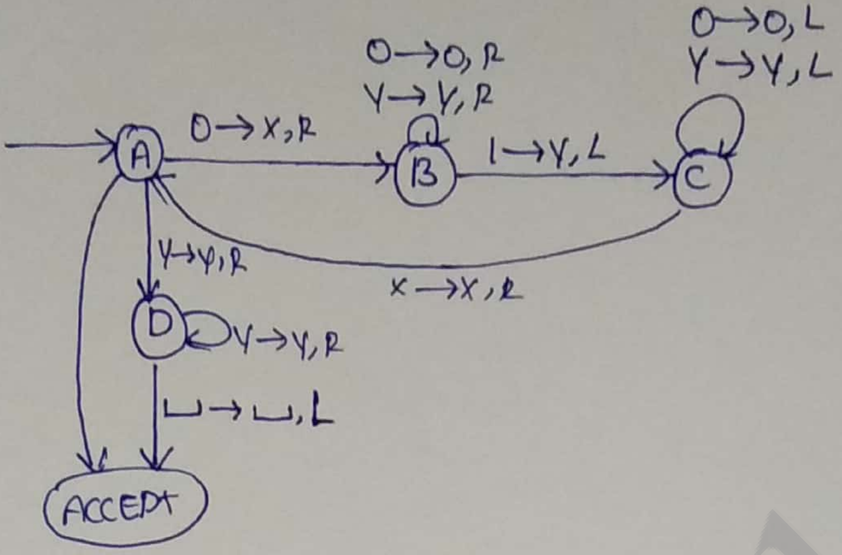
1) Design a turing machine which recognizes the language

$L = 01^*0$



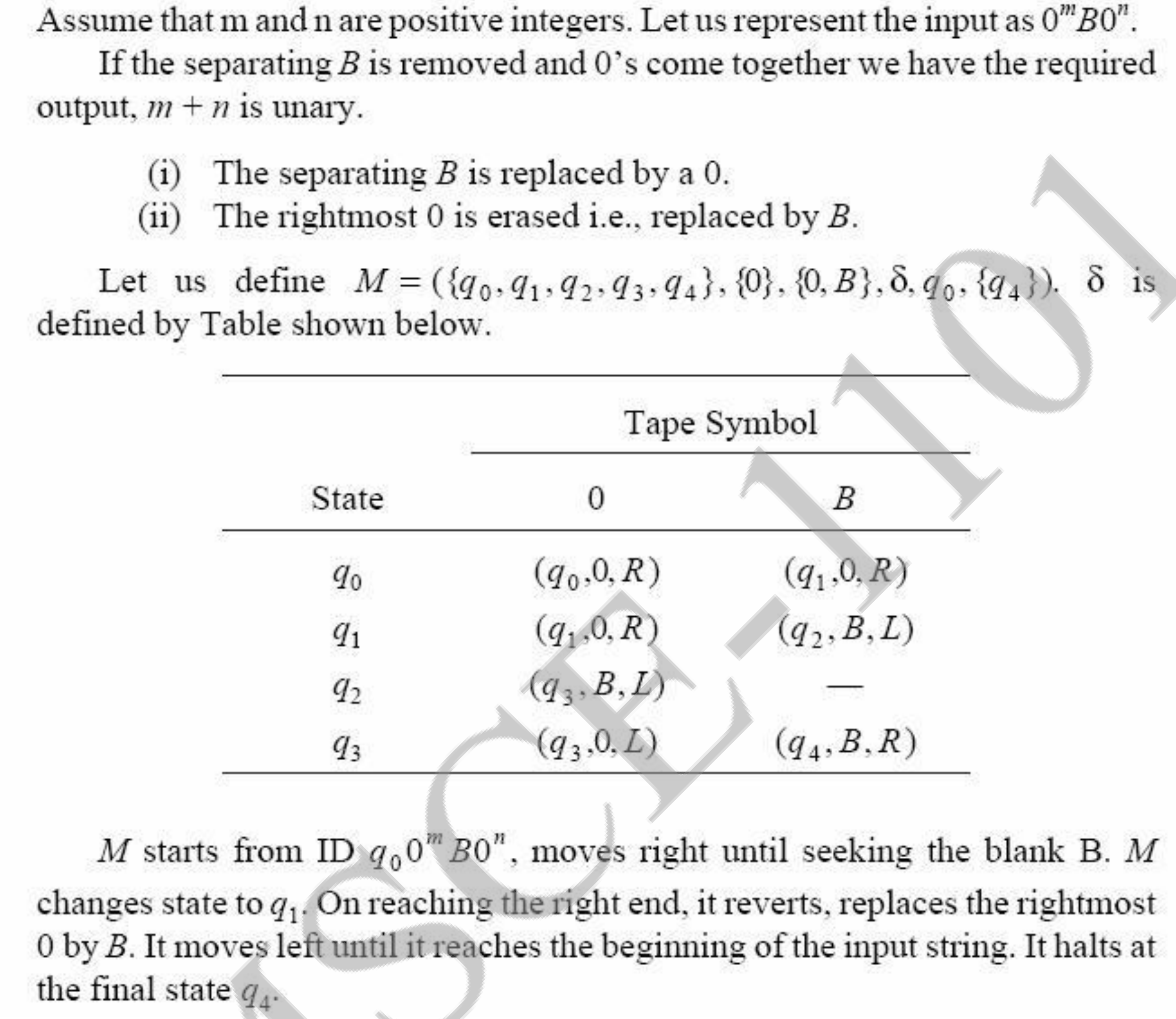
X	Y	Y	X	⊔	....

2. $0^N 1^N$:



↓				↓	↓	↓	↓	↓	↓	
0	0	0	0	1	1	1	1	␣	␣	....
X	X	X	X	Y	Y	Y	Y			

Design a Turing machine to add two given integers.
Solution:



Assume that m and n are positive integers. Let us represent the input as $0^m B0^n$.

If the separating B is removed and 0's come together we have the required output, $m + n$ is unary.

(i) The separating B is replaced by a 0.
(ii) The rightmost 0 is erased i.e., replaced by B.

Let us define $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0\}, \{0, B\}, \delta, q_0, \{q_4\})$. δ is defined by Table shown below.

	Tape Symbol	
State	0	B
q_0	$(q_0, 0, R)$	$(q_1, 0, R)$
q_1	$(q_1, 0, R)$	(q_2, B, L)
q_2	(q_3, B, L)	—
q_3	$(q_3, 0, L)$	(q_4, B, R)

M starts from ID $q_0 0^m B0^n$, moves right until seeking the blank B. M changes state to q_1. On reaching the right end, it reverts, replaces the rightmost 0 by B. It moves left until it reaches the beginning of the input string. It halts at the final state q_4.

Some unsolvable Problems are as follows:
(i) Does a given Turing machine *M* halts on all input?
(ii) Does Turing machine *M* halt for any input?
(iii) Is the language *L*(*M*) finite?
(iv) Does *L*(*M*) contain a string of length *k*, for some given *k*?
(v) Do two Turing machines *M*1 and *M*2 accept the same language?

It is very obvious that if there is no algorithm that decides, for an arbitrary given Turing machine *M* and input string *w*, whether or not *M* accepts *w*. These problems for which no algorithms exist are called "UNDECIDABLE" or "UNSOLVABLE".

Code for Turing Machine:

Our next goal is to devise a binary code for Turing machines so that each TM with input alphabet $\{0, 1\}$ may be thought of as a binary string. Since we just saw how to enumerate the binary strings, we shall then have an identification of the Turing machines with the integers, and we can talk about "the ith Turing machine, M_i." To represent a TM $M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F)$ as a binary string, we must first assign integers to the states, tape symbols, and directions L and R.

- We shall assume the states are $q_1, q_2, \ldots, q_r$ for some r. The start state will always be q_1, and q_2 will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.

- We shall assume the tape symbols are $X_1, X_2, \ldots, X_s$ for some s. X_1 always will be the symbol 0, X_2 will be 1, and X_3 will be B, the blank. However, other tape symbols can be assigned to the remaining integers arbitrarily.

- We shall refer to direction L as D_1 and direction R as D_2.

Since each TM M can have integers assigned to its states and tape symbols in many different orders, there will be more than one encoding of the typical TM. However, that fact is unimportant in what follows, since we shall show that no encoding can represent a TM M such that $L(M) = L_d$.

Once we have established an integer to represent each state, symbol, and direction, we can encode the transition function δ. Suppose one transition rule is $\delta(q_i, X_j) = (q_k, X_l, D_m)$, for some integers i, j, k, l, and m. We shall code this rule by the string $0^i10^j10^k10^l10^m$. Notice that, since all of i, j, k, l, and m are at least one, there are no occurrences of two or more consecutive 1's within the code for a single transition.

A code for the entire TM M consists of all the codes for the transitions, in some order, separated by pairs of 1's:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

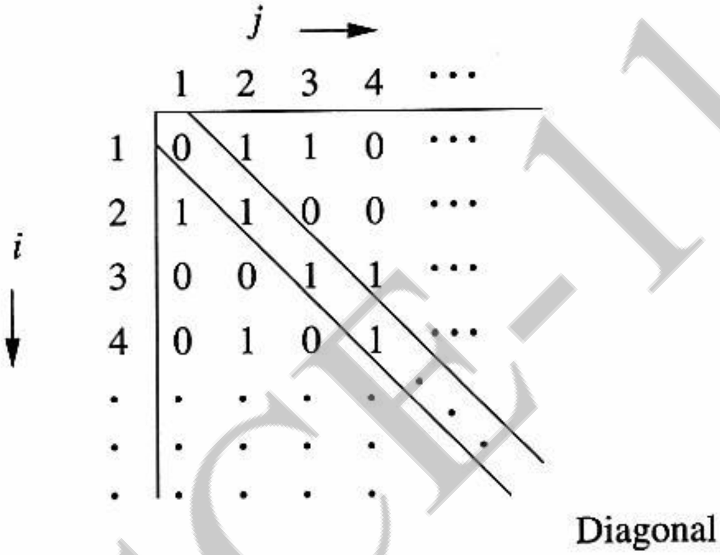
where each of the C's is the code for one transition of M.

Diagonalization language:

- The language L_d, the *diagonalization language*, is the set of strings w_i such that w_i is not in $L(M_i)$.

That is, L_d consists of all strings w such that the TM M whose code is w does not accept when given w as input.

The reason L_d is called a "diagonalization" language can be seen if we consider Fig. 9.1. This table tells for all i and j, whether the TM M_i accepts input string w_j; 1 means "yes it does" and 0 means "no it doesn't."[1] We may think of the ith row as the *characteristic vector* for the language $L(M_i)$; that is, the 1's in this row indicate the strings that are members of this language.



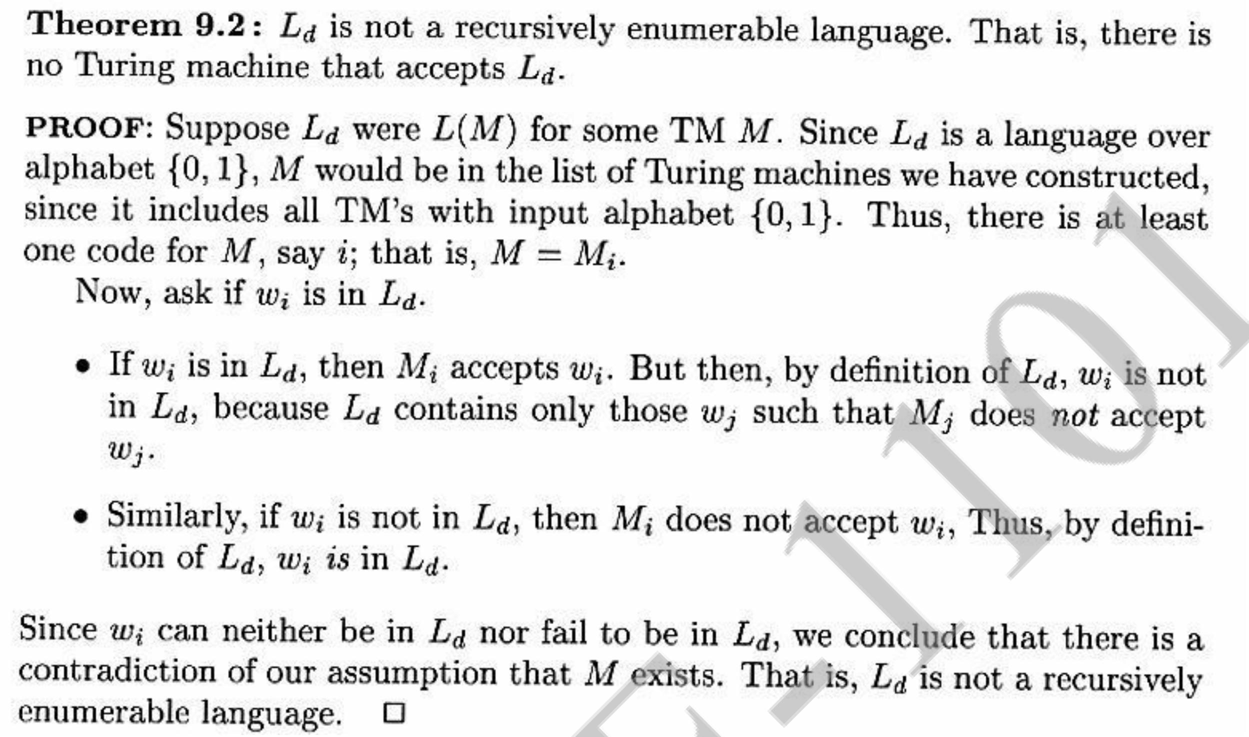
i \ j →	1	2	3	4	$\cdots$
1	0	1	1	0	$\cdots$
2	1	1	0	0	$\cdots$
3	0	0	1	1	$\cdots$
4	0	1	0	1	$\cdots$
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

This table represents language acceptable by Turing machine

The diagonal values tell whether M_i accepts w_i. To construct L_d, we complement the diagonal. For instance, if Fig. 9.1 were the correct table, then the complemented diagonal would begin $1, 0, 0, 0, \ldots$. Thus, L_d would contain $w_1 = \epsilon$, not contain w_2 through w_4, which are 0, 1, and 00, and so on.

The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row, is called *diagonalization*. It works because the complement of the diagonal is

Proof that L_d is not recursively enumerable:



Theorem 9.2: L_d is not a recursively enumerable language. That is, there is no Turing machine that accepts L_d.

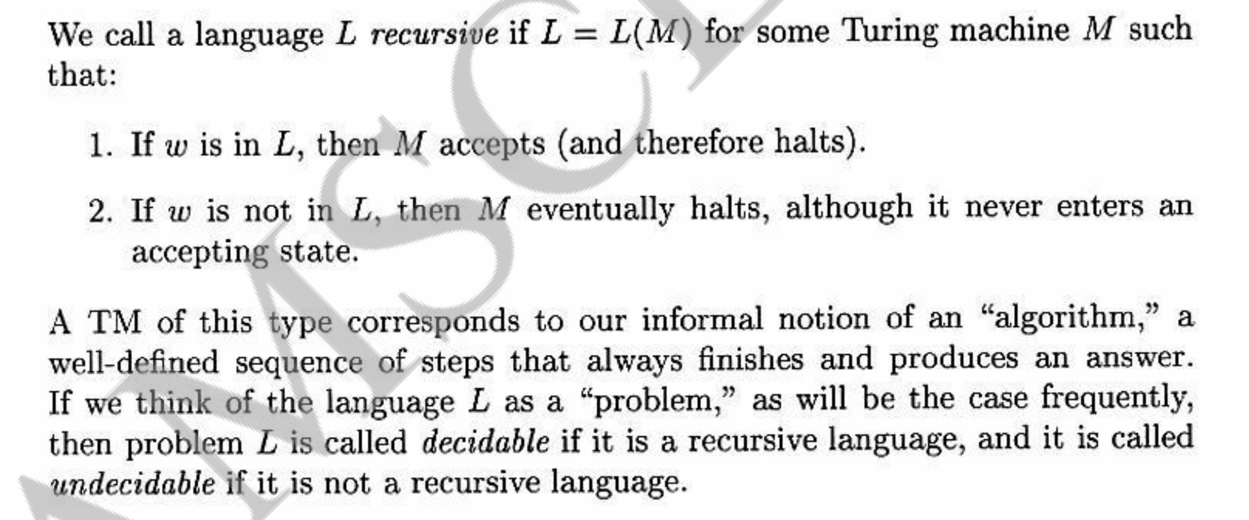
PROOF: Suppose L_d were $L(M)$ for some TM M. Since L_d is a language over alphabet $\{0, 1\}$, M would be in the list of Turing machines we have constructed, since it includes all TM's with input alphabet $\{0, 1\}$. Thus, there is at least one code for M, say i; that is, $M = M_i$.

Now, ask if w_i is in L_d.

- If w_i is in L_d, then M_i accepts w_i. But then, by definition of L_d, w_i is not in L_d, because L_d contains only those w_j such that M_j does *not* accept w_j.
- Similarly, if w_i is not in L_d, then M_i does not accept w_i, Thus, by definition of L_d, w_i *is* in L_d.

Since w_i can neither be in L_d nor fail to be in L_d, we conclude that there is a contradiction of our assumption that M exists. That is, L_d is not a recursively enumerable language. □

Recursive Languages:



We call a language L *recursive* if $L = L(M)$ for some Turing machine M such that:

1. If w is in L, then M accepts (and therefore halts).
2. If w is not in L, then M eventually halts, although it never enters an accepting state.

A TM of this type corresponds to our informal notion of an "algorithm," a well-defined sequence of steps that always finishes and produces an answer. If we think of the language L as a "problem," as will be the case frequently, then problem L is called *decidable* if it is a recursive language, and it is called *undecidable* if it is not a recursive language.

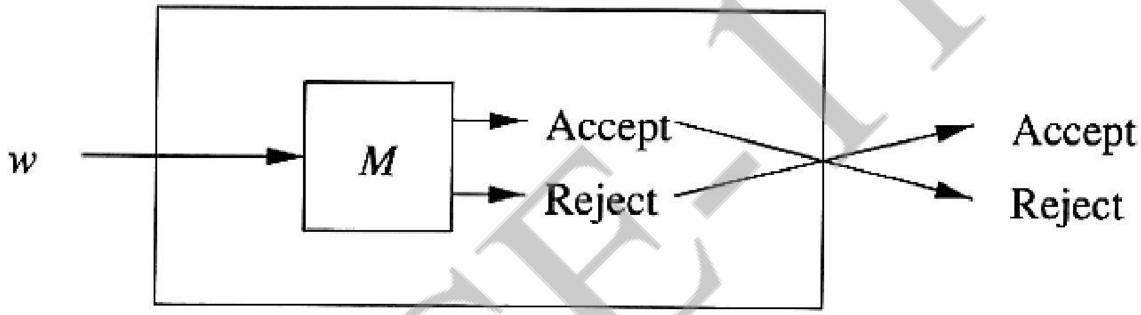
Theorem 9.3: If L is a recursive language, so is $\overline{L}$.

PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$ by the construction suggested in Fig. 9.3. That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$:

1. The accepting states of M are made nonaccepting states of $\overline{M}$ with no transitions; i.e., in these states $\overline{M}$ will halt without accepting.

2. $\overline{M}$ has a new accepting state r; there are no transitions from r.

3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.



Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. □

Theorem 9.4: If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 9.3, $\overline{L}$ is recursive as well.

PROOF: The proof is suggested by Fig. 9.4. Let $L = L(M_1)$ and $\overline{L} = L(M_2)$. Both M_1 and M_2 are simulated in parallel by a TM M. We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious. One tape of M simulates the tape of M_1, while the other tape of M simulates the tape of M_2. The states of M_1 and M_2 are each components of the state of M.

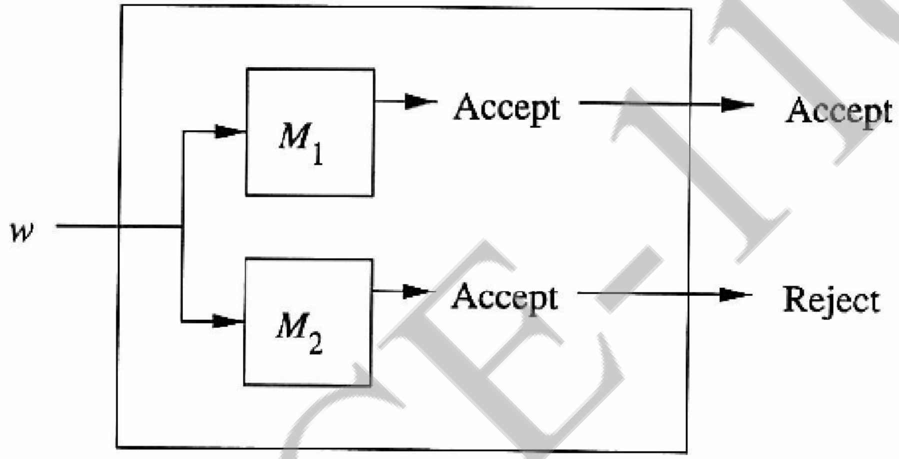


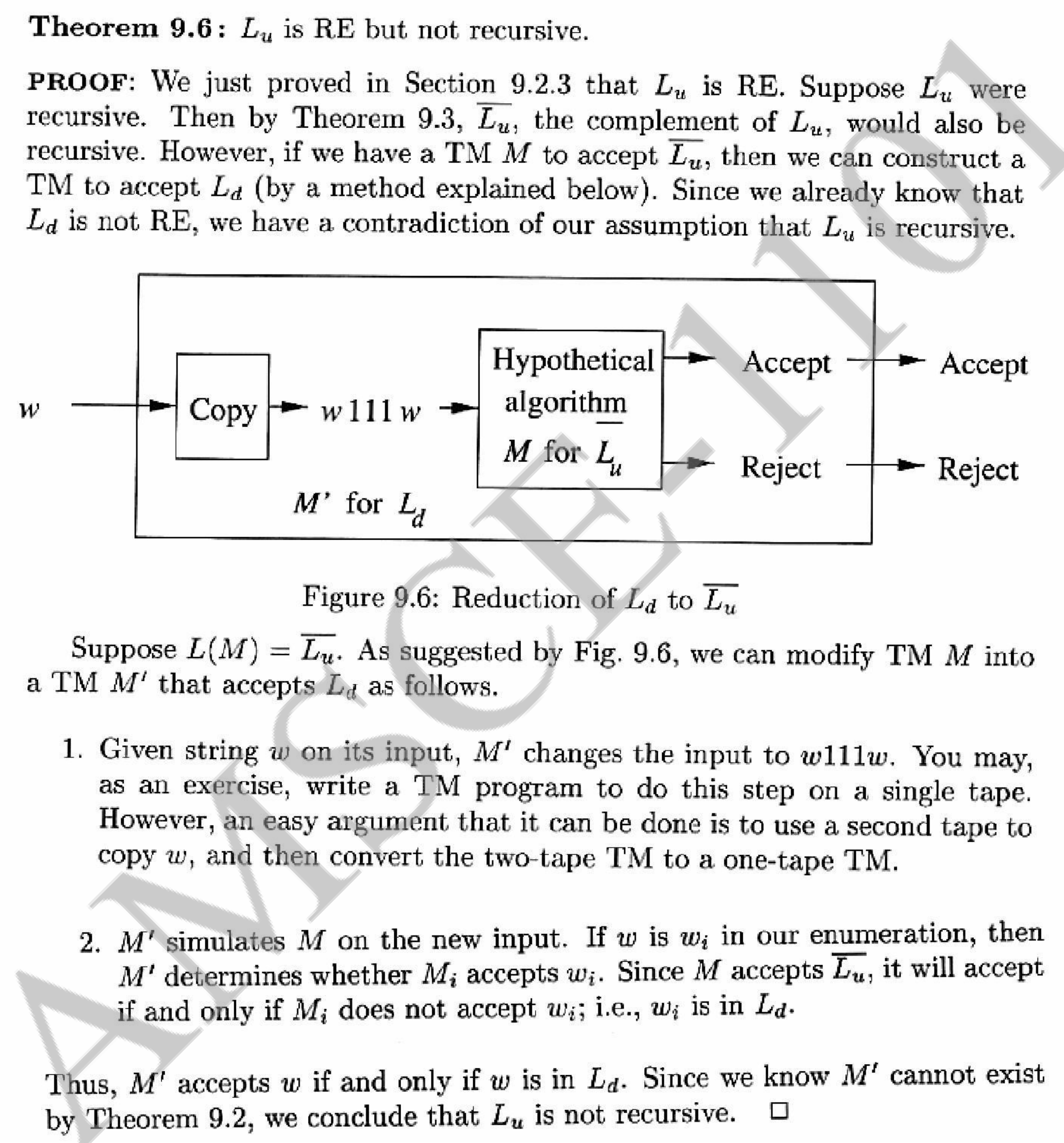
Figure 9.4: Simulation of two TM's accepting a language and its complement

If input w to M is in L, then M_1 will eventually accept. If so, M accepts and halts. If w is not in L, then it is in $\overline{L}$, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and $L(M)$ is exactly L. Since M always halts, and $L(M) = L$, we conclude that L is recursive. □

Universal Language:

We define L_u, the *universal language*, to be the set of binary strings that encode, in the notation of Section 9.1.2, a pair (M, w), where M is a TM with the binary input alphabet, and w is a string in $(\mathbf{0}+\mathbf{1})^*$, such that w is in $L(M)$. That is, L_u is the set of strings representing a TM and an input accepted by that TM. We shall show that there is a TM U, often called the *universal Turing machine*, such that $L_u = L(U)$. Since the input to U is a binary string, U is in fact some M_i in the list of binary-input Turing machines we developed in

Undecidability of Universal Language:



Theorem 9.6: L_u is RE but not recursive.

PROOF: We just proved in Section 9.2.3 that L_u is RE. Suppose L_u were recursive. Then by Theorem 9.3, $\overline{L_u}$, the complement of L_u, would also be recursive. However, if we have a TM M to accept $\overline{L_u}$, then we can construct a TM to accept L_d (by a method explained below). Since we already know that L_d is not RE, we have a contradiction of our assumption that L_u is recursive.

Figure 9.6: Reduction of L_d to $\overline{L_u}$

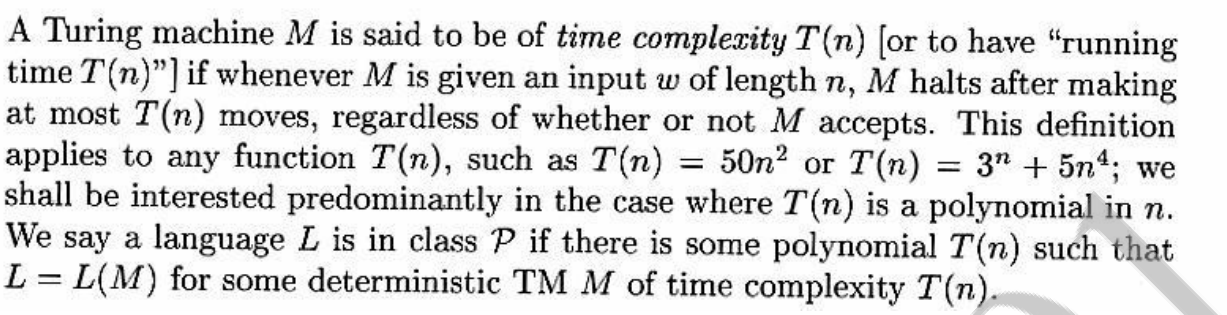
Suppose $L(M) = \overline{L_u}$. As suggested by Fig. 9.6, we can modify TM M into a TM M' that accepts L_d as follows.

1. Given string w on its input, M' changes the input to $w111w$. You may, as an exercise, write a TM program to do this step on a single tape. However, an easy argument that it can be done is to use a second tape to copy w, and then convert the two-tape TM to a one-tape TM.

2. M' simulates M on the new input. If w is w_i in our enumeration, then M' determines whether M_i accepts w_i. Since M accepts $\overline{L_u}$, it will accept if and only if M_i does not accept w_i; i.e., w_i is in L_d.

Thus, M' accepts w if and only if w is in L_d. Since we know M' cannot exist by Theorem 9.2, we conclude that L_u is not recursive. □

Class p-problem solvable in polynomial time:



A Turing machine M is said to be of *time complexity* $T(n)$ [or to have "running time $T(n)$"] if whenever M is given an input w of length n, M halts after making at most $T(n)$ moves, regardless of whether or not M accepts. This definition applies to any function $T(n)$, such as $T(n) = 50n^2$ or $T(n) = 3^n + 5n^4$; we shall be interested predominantly in the case where $T(n)$ is a polynomial in n. We say a language L is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM M of time complexity $T(n)$.

Non deterministic polynomial time:

A nondeterministic TM that never makes more than p(n) moves in any sequence of choices for some polynomial p is said to be non polynomial time NTM.

- NP is the set of languags that are accepted by polynomial time NTM's
- Many problems are in NP but appear not to be in p.
- One of the great mathematical questions of our age: is there anything in NP that is not in p?

NP-complete problems:

If We cannot resolve the "p=np question, we can at least demonstrate that certain problems in NP are the hardest , in the sense that if any one of them were in P , then P=NP.

- These are called NP-complete.
- Intellectual leverage: Each NP-complete problem's apparent difficulty reinforces the belief that they are all hard.

Methods for proving NP-Complete problems:

- Polynomial time reduction (PTR): Take time that is some polynomial in the input size to convert instances of one problem to instances of another.
- If P1 PTR to P2 and P2 is in P1 the so is P1.
- Start by showing every problem in NP has a PTR to Satisfiability of Boolean formula.
- Then, more problems can be proven NP complete by showing that SAT PTRs to them directly or indirectly.

Undecidable Problem about Turing Machine

- Reduction is a technique in which if a problem P1 is reduced to a problem P2 then any solution of P2 solves P1. In general, if we have an algorithm to convert an instance of a problem P1 to an instance of a problem P2 that have the same answer then it is called as P1 reduced P2.
- Hence if P1 is not recursive then P2 is also not recursive. Similarly, if P1 is not recursively enumerable then P2 also is not recursively enumerable.

- **Theorem:** if P1 is reduced to P2 then
- If P1 is undecidable, then P2 is also undecidable.
- If P1 is non-RE, then P2 is also non-RE.

Proof:

- Consider an instance w of P1. Then construct an algorithm such that the algorithm takes instance w as input and converts it into another instance x of P2. Then apply that algorithm to check whether x is in P2.
- If the algorithm answer 'yes' then that means x is in P2, similarly we can also say that w is in P1. Since we have obtained P2 after reduction of P1. Similarly if algorithm answer 'no' then x is not in P2, that also means w is not in P1. This proves that if P1 is undecidable, then P1 is also undecidable.

- There are two types of languages empty and non empty language. L_et L_e denotes an empty language, and L_{ne} denotes non empty language. L_et w be a binary string, and Mi be a TM. If $L(M_j) = \Phi$ then Mi does not accept input then w is in L_e. Similarly, if $L(M_j)$ is not the empty language, then w is in L_{ne}. Thus we can say that

- $L_e = \{M \mid L(M) = \Phi\}$
  $L_{ne} = \{M \mid L(M) \neq \Phi\}$
- Both L_e and L_{ne} are the complement of one another.

Post Correspondance Problem

- The Post Correspondence Problem (PCP) was invented by Emil Post in 1946. It is called as an undecidable decision problem. The PCP problem rather than an alphabet ∑ is considered
- Given the following two lists, **M** and **N** of non-empty strings over ∑ –
- M = (x_1, x_2, x_3,………, x_n)
- N = (y_1, y_2, y_3,………, y_n)

- The Post Correspondence Solution, if for some $i_1, i_2, \ldots\ldots\ldots\ldots i_k$, where $1 \leq i_j \leq n$, the condition $x_{i1} \ldots\ldots x_{ik} = y_{i1} \ldots\ldots y_{ik}$ satisfies.

Example

- M = (abb, aa, aaa) and N = (bba, aaa, aa)
- Include a Post Correspondence Solution?
- Solution
- $x_1x_2x_3$MAbbaaaaaNBbaaaaaa

The Class P

- Definition: The complexity class P is the set of all decision problems that can be solved with worst-case polynomial time-complexity.
- A problem is in the class P if it is a decision problem and there exists an algorithm that solves any instance of size n in O(n k) time, for some integer k.

- Strictly, n must be the number of bits needed for a ‘reasonable’ encoding of the input. But we won’t get bogged down in such fine details.
- So P is just the set of tractable decision problems: the decision problems for which we have polynomial-time algorithms.

- The problems in the picture that are in NP but not in P are ones that we're not sure about: –
- there is no known polynomial-time algorithm; –
- but no proof of intractability.

- We know that P $\subseteq$ NP. But much more than that we don’t know.
- The definition of NP allows for the inclusion of problems that may not be in P. But it may turn out that there are no such problems and that P = NP

The Class P and NP

P and NP problems

- Assume we have a "conventional" deterministic computer.
  - The class of problems which can be solved on such a computer in polynomial time is called P (for Polynomial).
- Suppose we have a (theoretical) non-deterministic computer that can "guess" the right option when faced with choices.
  - The class of problems which can be solved on a non-deterministic computer in polynomial time is called NP (for Nondeterministic Polynomial).

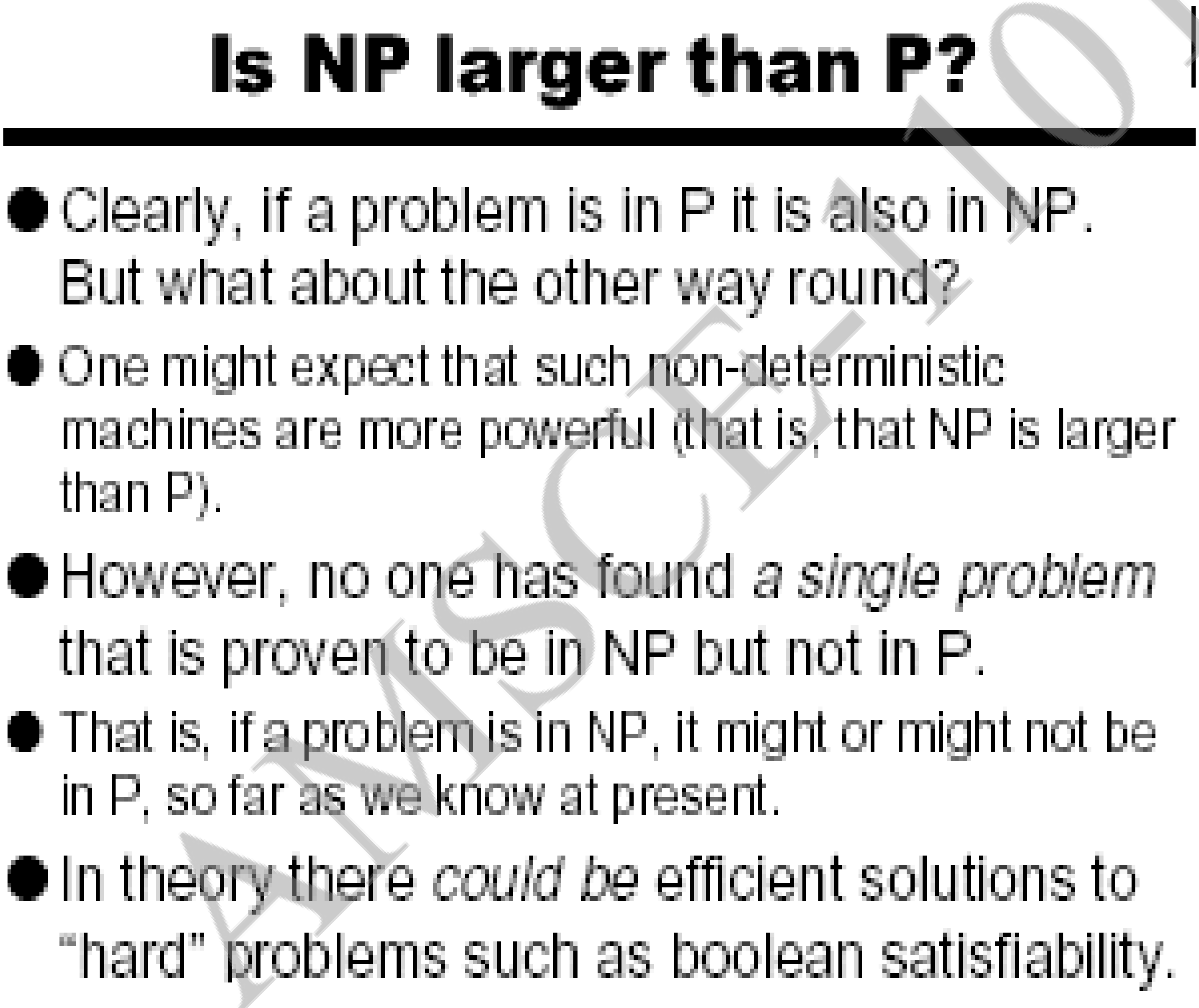
- Partition Given $A = \{a_1, \ldots, a_n\}$ each a_i with $s(a_i) \in \int$ is there a $S \subset [n]$ s.t. $\sum_{i \in S} s(a_i) = \sum_{j \notin S} s(a_j)$?

certificate: S. To verify check in $O(n)$ that $\sum_{i \in S} s(a_i) = \sum_{j \notin S} s(a_j)$

Theorem: $\mathrm{P} \subseteq \mathrm{NP}$.

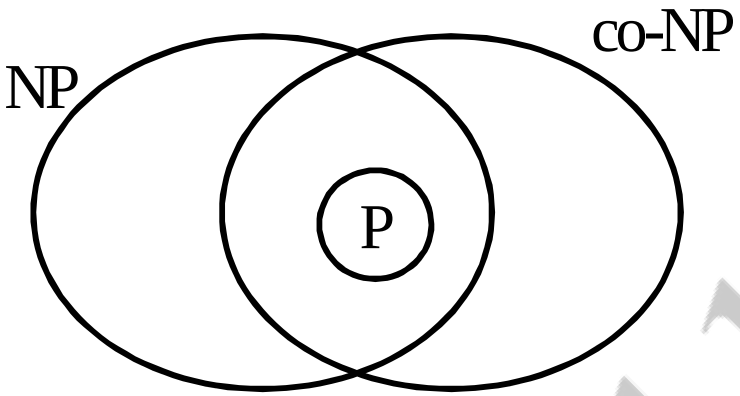
The US$ 10^6 Question: Is $\mathrm{P} \neq \mathrm{NP}$ or $\mathrm{P} = \mathrm{NP}$?

http://www.claymath.org/prizeproblems/pvsnp.html



Is NP larger than P?

- Clearly, if a problem is in P it is also in NP. But what about the other way round?
- One might expect that such non-deterministic machines are more powerful (that is, that NP is larger than P).
- However, no one has found *a single problem* that is proven to be in NP but not in P.
- That is, if a problem is in NP, it might or might not be in P, so far as we know at present.
- In theory there *could be* efficient solutions to "hard" problems such as boolean satisfiability.



One of the central (and widely and intensively studied 30 years) problems of (theoretical) computer science is to prove that

(a) **P□NP** (b) **NP □ co-NP**.

- All evidence indicates that these conjectures are true.
- Disproving any of these two conjectures would not only be considered truly spectacular, but would also come as a tremendous surprise (with a variety of far-reaching counterintuitive consequences).

NP-complete: Collection Z of problems is NP-complete if (a) it is NP and (b) if polynomial-time algorithm existed for solving problems in Z, then P=NP.

NP-completeness

A problem $A \in$ NP is NP-complete if for every Bin NP, $B \leq A$. If for Bin NP, $B \leq A$ but $B \notin$ NP then A is said to be NP-hard.

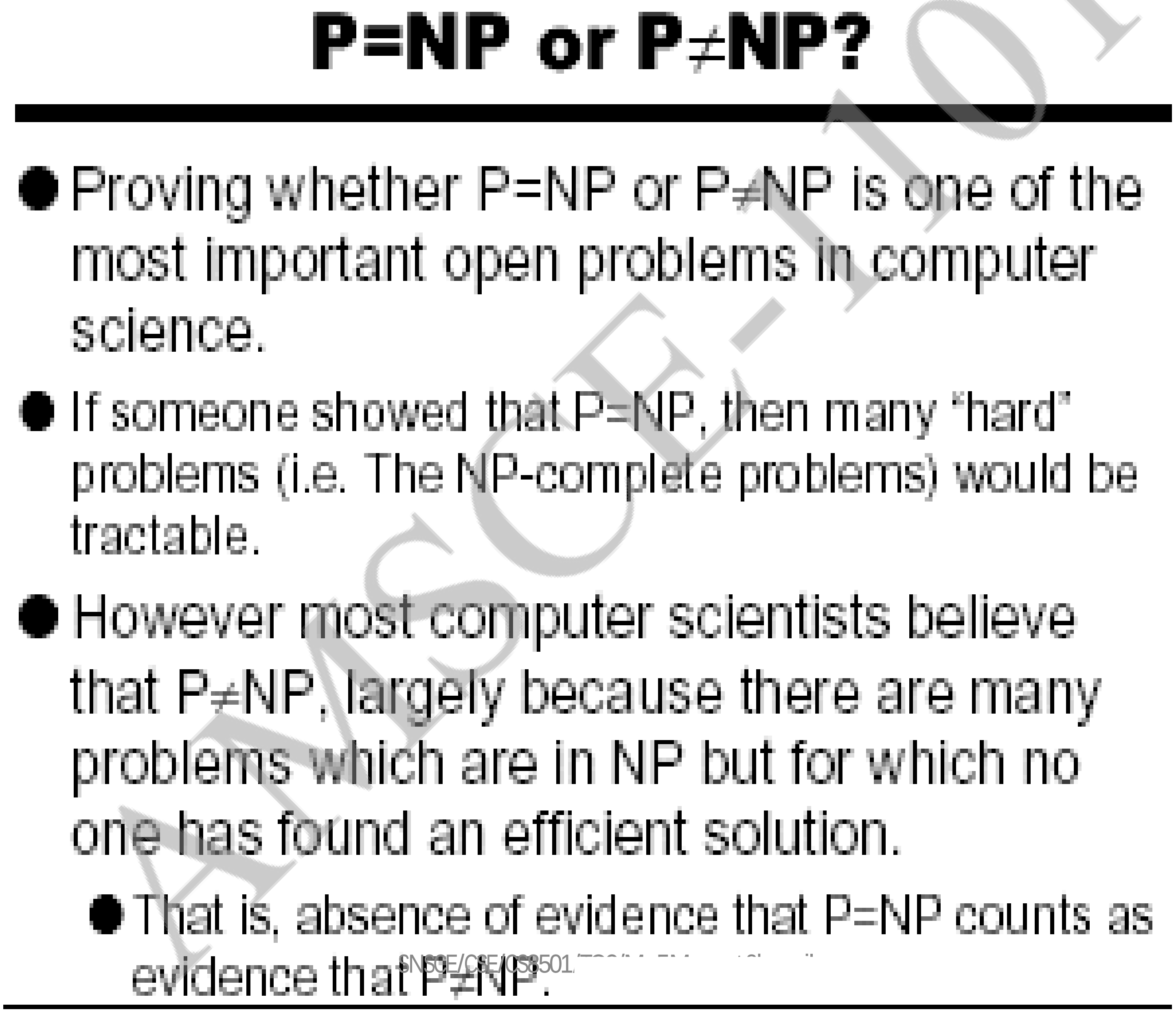
Lemma: If A is NP-complete, the A in P iff $P = NP$.

So once we prove that a problem is NP-complete, either A has no efficient algorithm or all NP problemas are in P.

Majority conjecture: P ≠ NP

Some NP-complete problems

- Many practical problems are NP-complete.
  - Given a linear program (a set of linear inequalities) is there an integer solution to the variables?
  - Given a set of integers, can they be divided into two sets whose sum is equal?
  - Given two identical processors, a set of tasks of varying length, and a deadline, can the tasks be scheduled so that they finish before the deadline?
  - If there is an efficient solution to any of these, then all NP problems have efficient solutions! This would have a major impact.



P=NP or P≠NP?

- Proving whether P=NP or P≠NP is one of the most important open problems in computer science.
- If someone showed that P=NP, then many “hard” problems (i.e. The NP-complete problems) would be tractable.
- However most computer scientists believe that P≠NP, largely because there are many problems which are in NP but for which no one has found an efficient solution.
  - That is, absence of evidence that P=NP counts as evidence that P≠NP.

Summary: P and NP

- Some problems seem to be intrinsically very complex (NP). The only "efficient" known solutions require a non-deterministic computer.
- At present we have no proof that such problems do not have efficient solutions (they could be in P).
- Some NP problems are significant in the sense that if they are in P, then so are all NP problems.