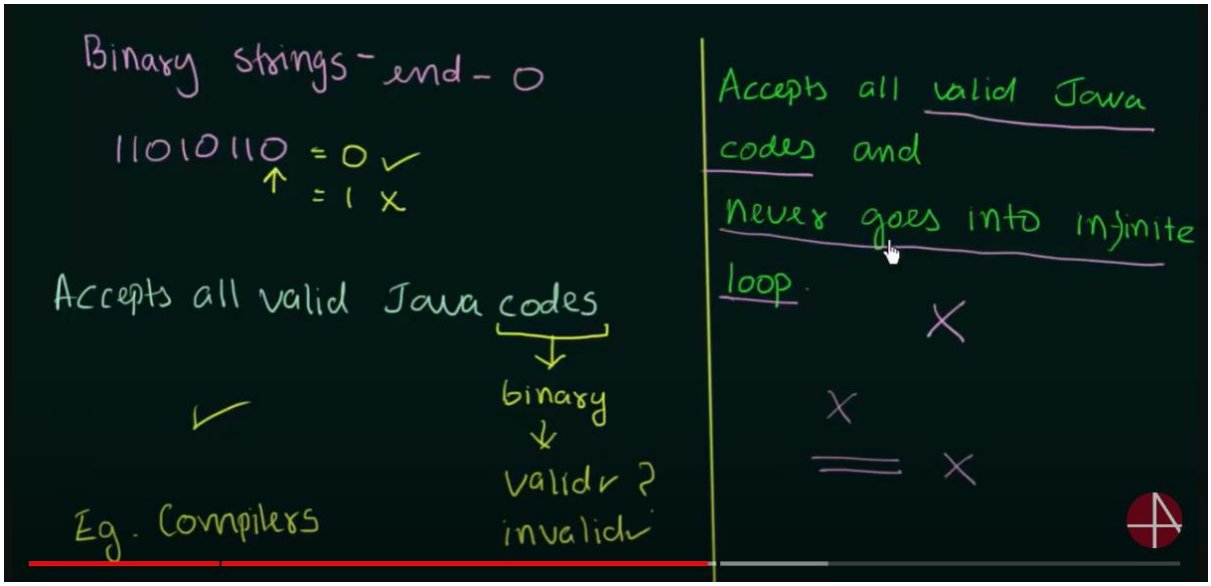
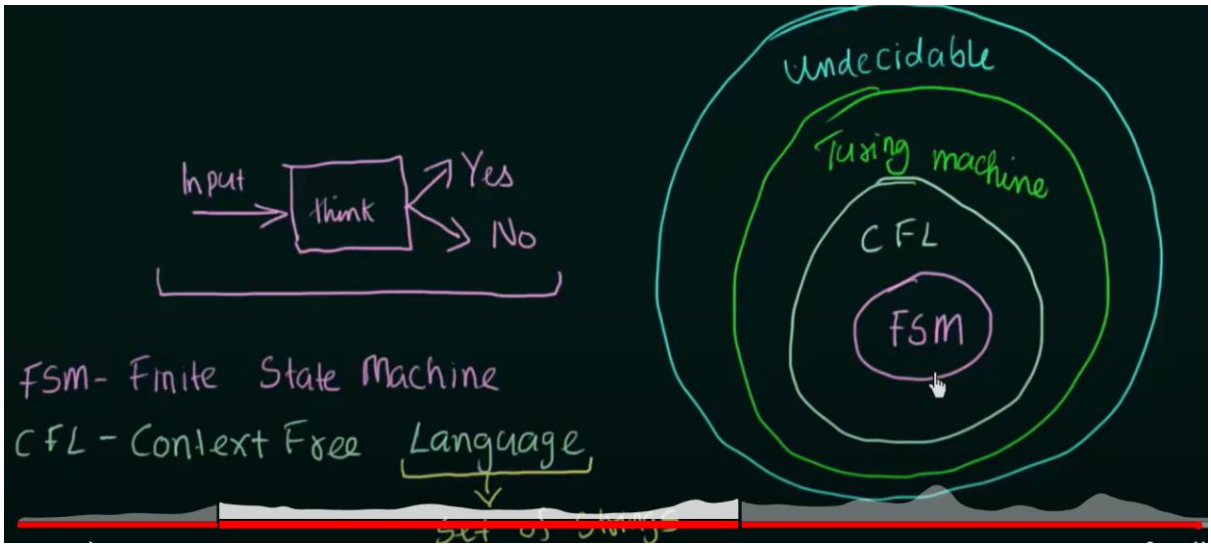
INTRODUCTION (Theory Of Computation)

>> One of the most fundamental courses of Computer Science

>> Will help you understand how people have thought about Computer Science as a Science in the past 50 years

>> It is mainly about what kind of things can you really compute mechanically, how fast and how much space does it take to do so





Finite State Machine (Prerequisites)

Symbol - a, b, c, 0, 1, 2, 3, ...

Alphabet - Σ - collection of symbols - Eg. {a, b}, {d, e, f, g} {0, 1, 2} ....

String - sequence of symbols. Eg. a, b, 0, 1, aa, bb, ab, 01, ..

Language - set of strings

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

L_1 = set of all strings of length 2

= {00, 01, 10, 11}

L_1 = set of all strings of length 2.

= {00, 01, 10, 11}

L_2 = set of all strings of length 3

= {000, 001, 010, 011, 100, 101, 110, 111}

finite

L_3 = set of all strings that begin with 0

= {0, 00, 01, 000, 001, 010, 011, 0000, ...}

infinite

Powers of Σ : $\Sigma = \{0, 1\}$

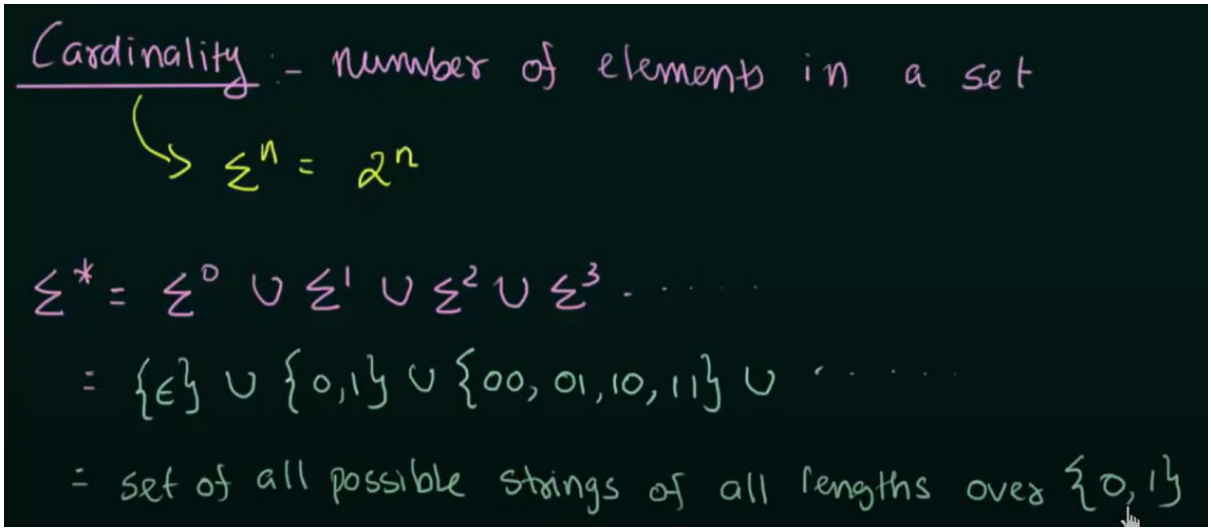
Σ^0 = set of all strings of length 0 : $\Sigma^0 = \{\epsilon\}$

Σ^1 = set of all strings of length 1 : $\Sigma^1 = \{0, 1\}$

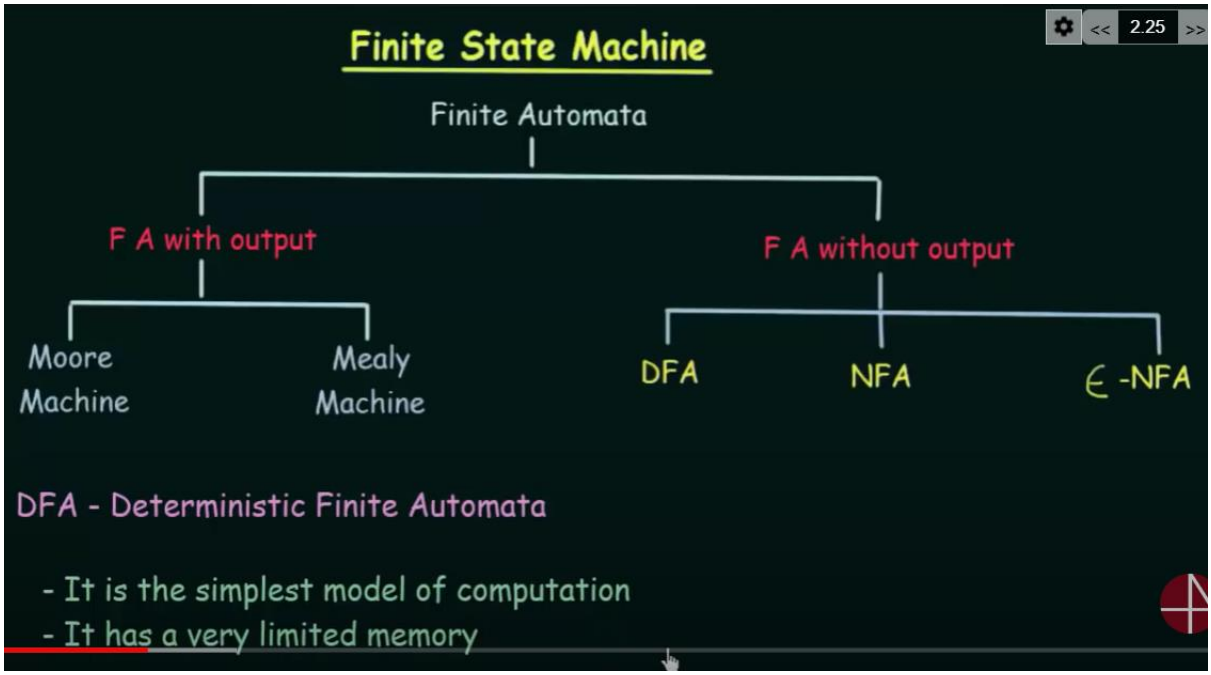
Σ^2 = set of all strings of length 2 : $\Sigma^2 = \{00, 01, 10, 11\}$

Σ^3 = set of all strings of length 3 : $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

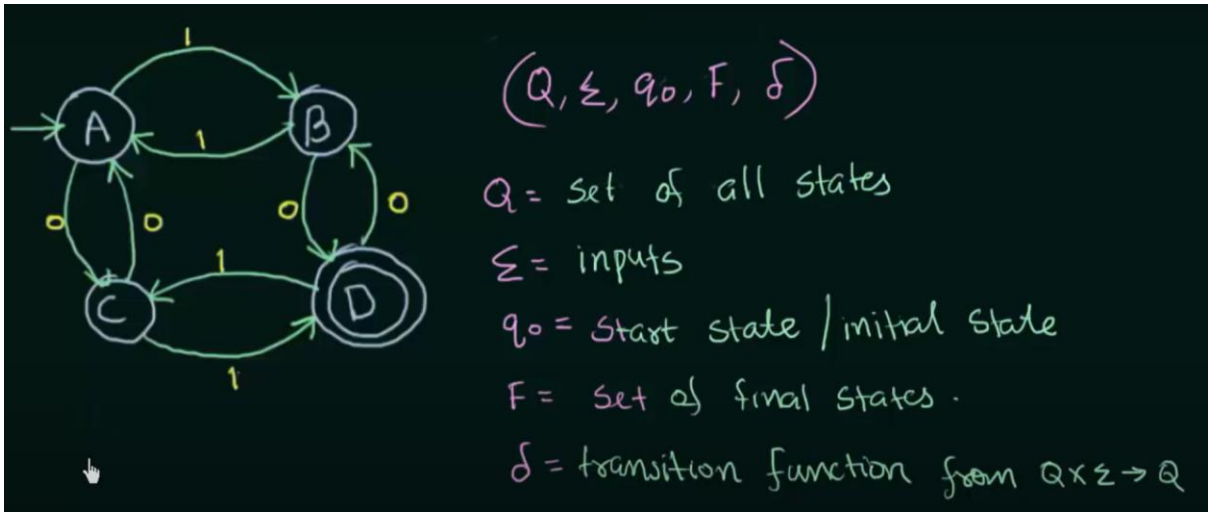
Σ^n = set of all strings of length n

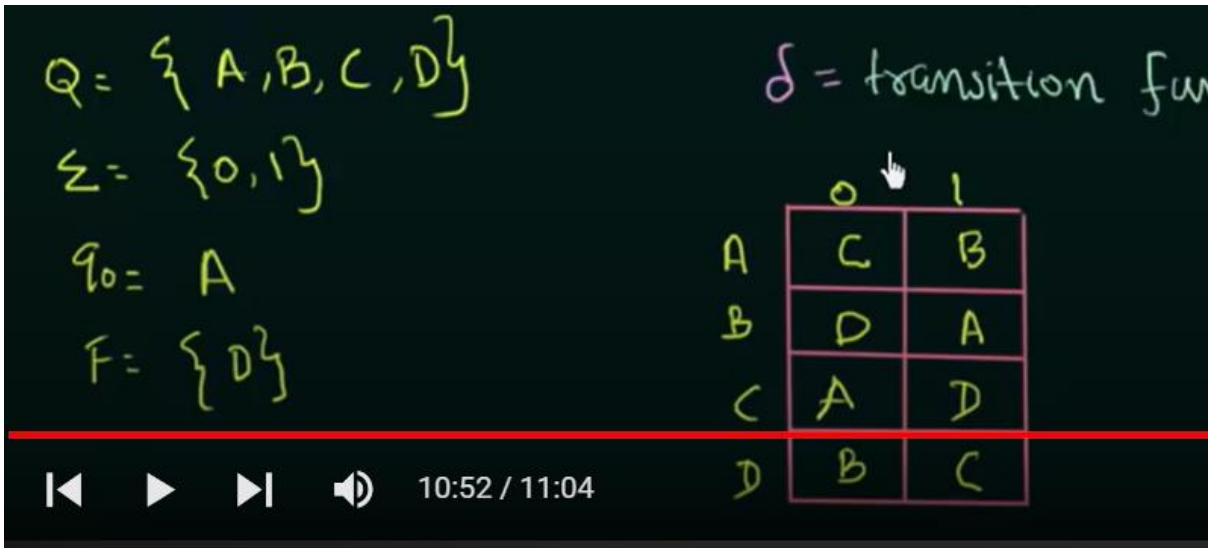


Note here sigma ^ n = 2 ^ n because the alphabet has only two digits 0 and 1.



Deterministic Finite Automata



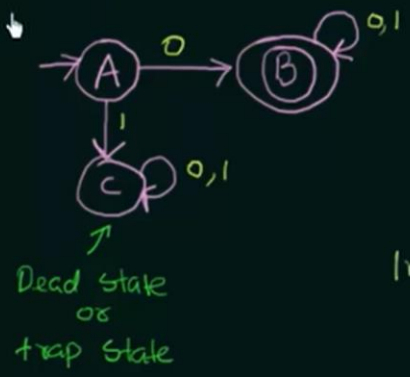


DFA example 1

Deterministic Finite Automata (Example-1)

L1 = Set of all strings that start with '0'

= { 0, 00, 01, 000, 010, 011, 0000, ... }



Dead state
or
trap state

Eg. 001 ✓

Initial state A → B → B → B – Final state

Eg. 101 ✗

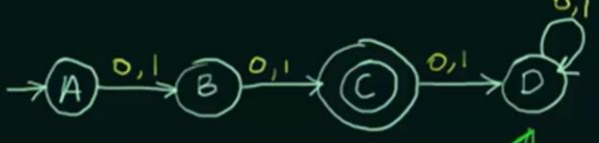
Initial state A → C → C → C – Not final state

Deterministic Finite Automata (Example-2)

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

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

$L = \{00, 01, 10, 11\}$



Dead State
or
Trap State.

Eg. 00 ✓

A → B → C – Final State

Eg. 10 ✓

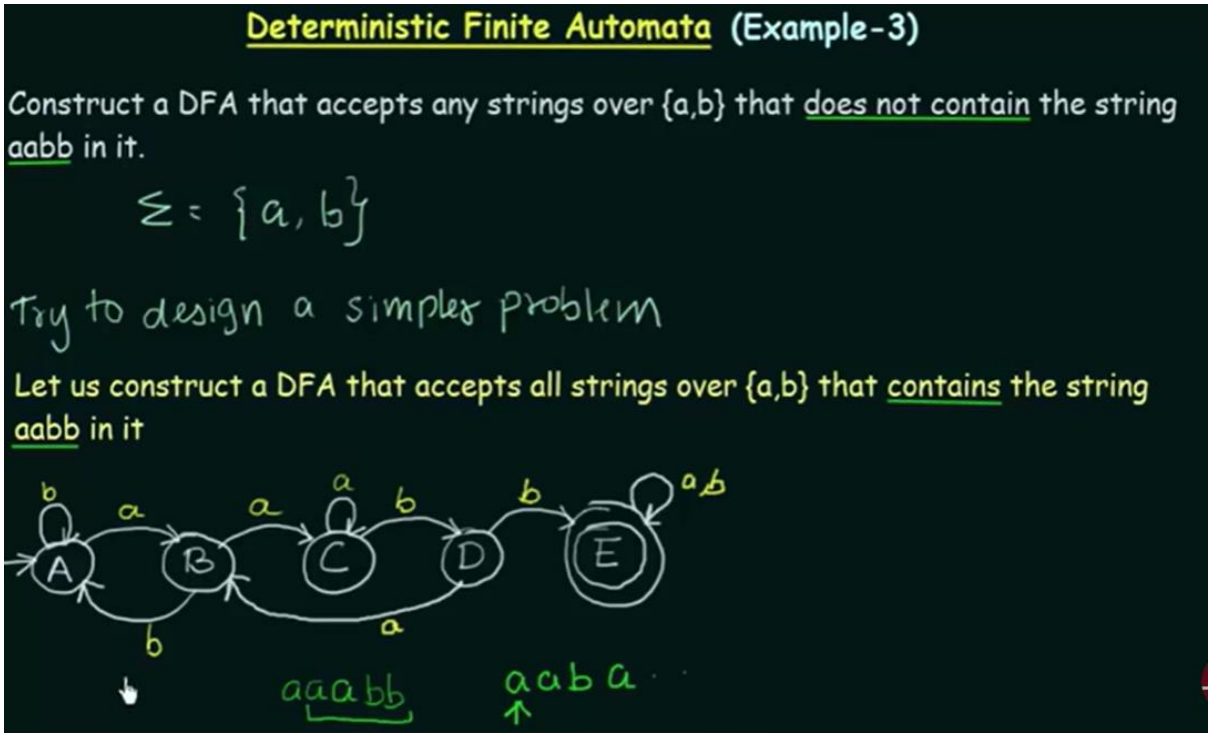
A → B → C – Final State

Eg. 001

A → B → C → D – Not final St

Eg. 1

A → B – Not final state

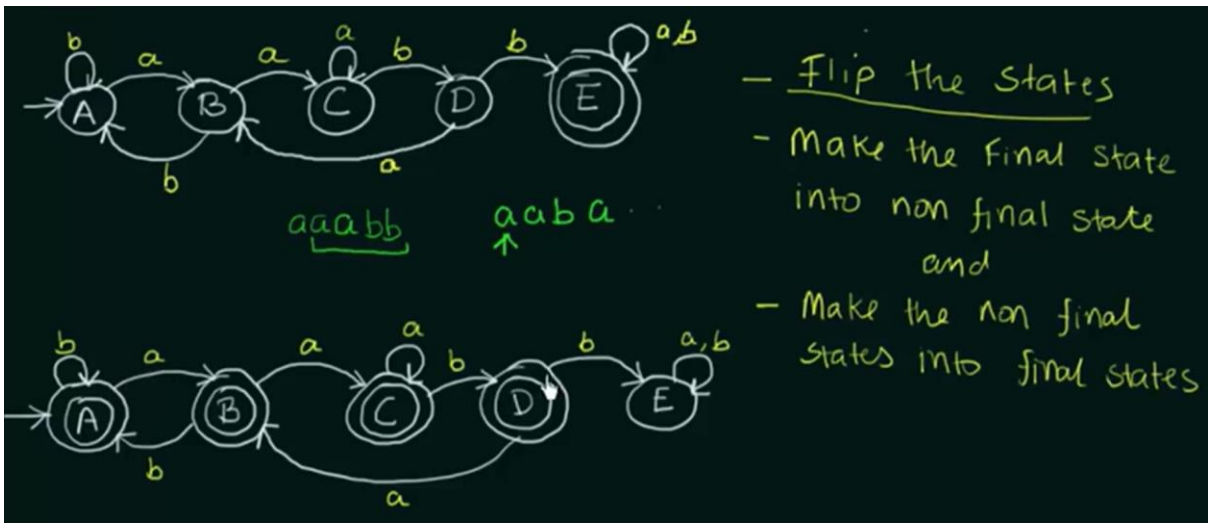


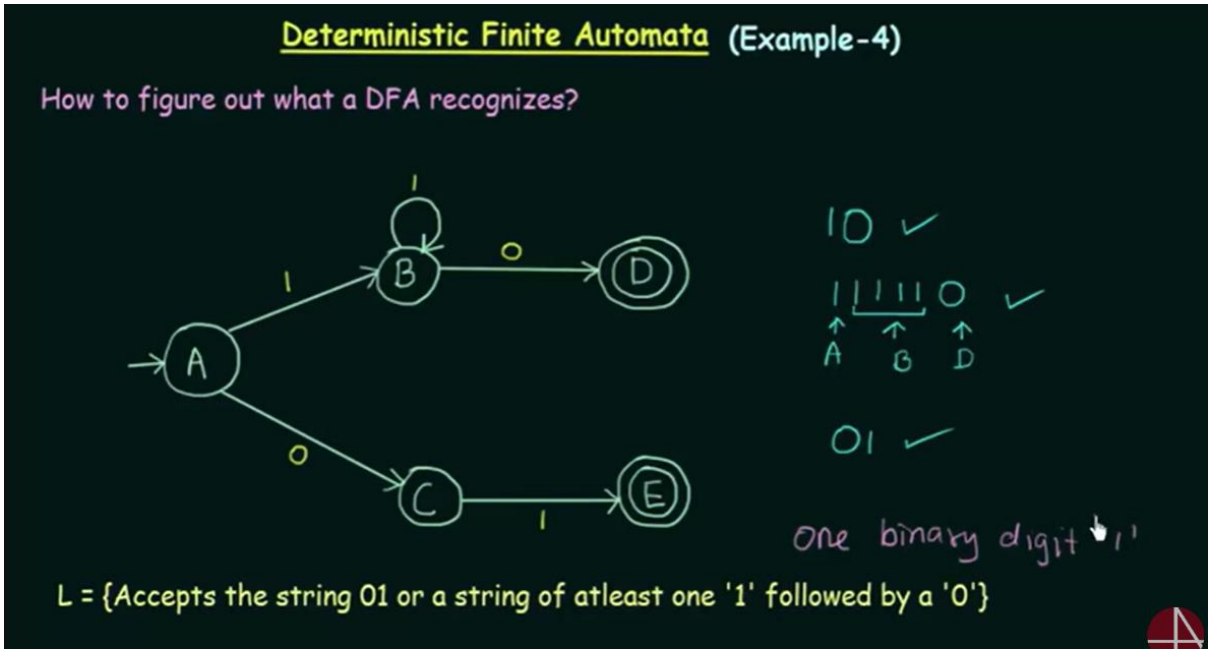
From C if we get we would have aaa, now if we get a b we could proceed to the next state so for a we have put a self-loop over C.

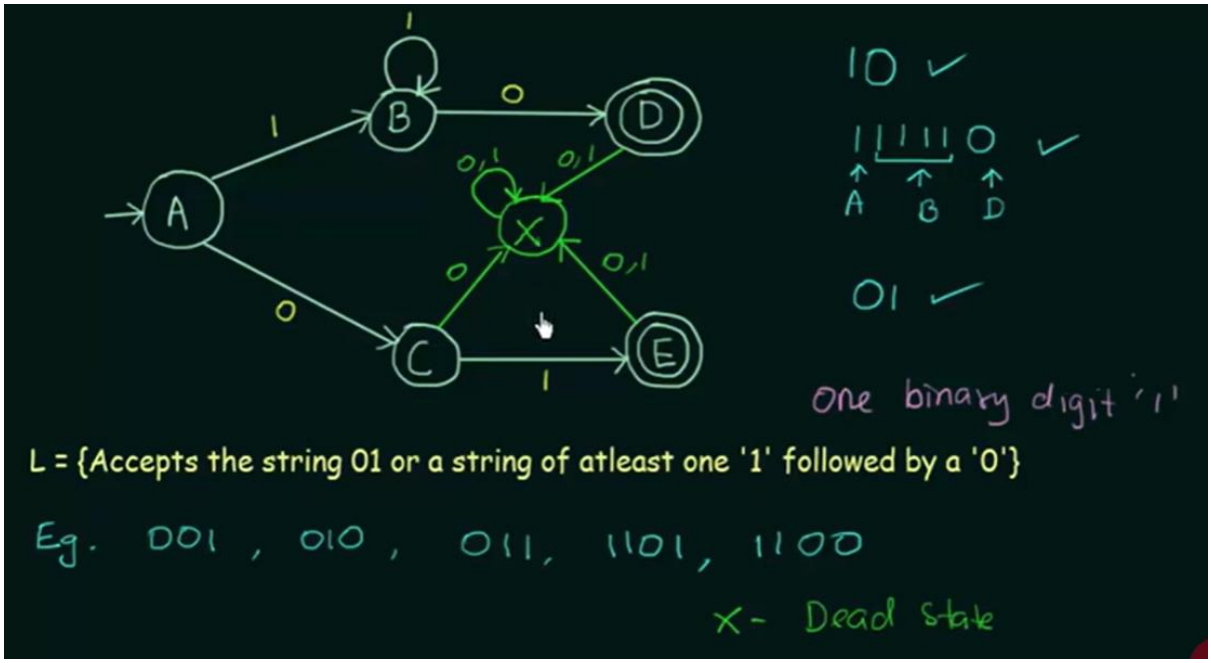
For state B if we get a b then we can't construct the required string, so we go back to state A.

For State D , if we get an a we go back to state B because we already have one a to consider as the string would be aaba.

But the actual problem is just the opposite, so now we flip the states







Regular Languages

Regular Languages

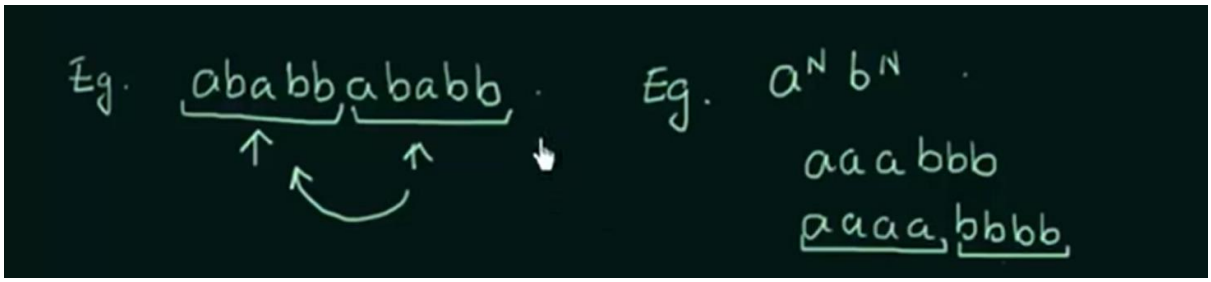
- A language is said to be a REGULAR LANGUAGE if and only if some Finite State Machine recognizes it

So what languages are NOT REGULAR ?

The languages
>> Which are not recognized by any FSM
>> Which require memory

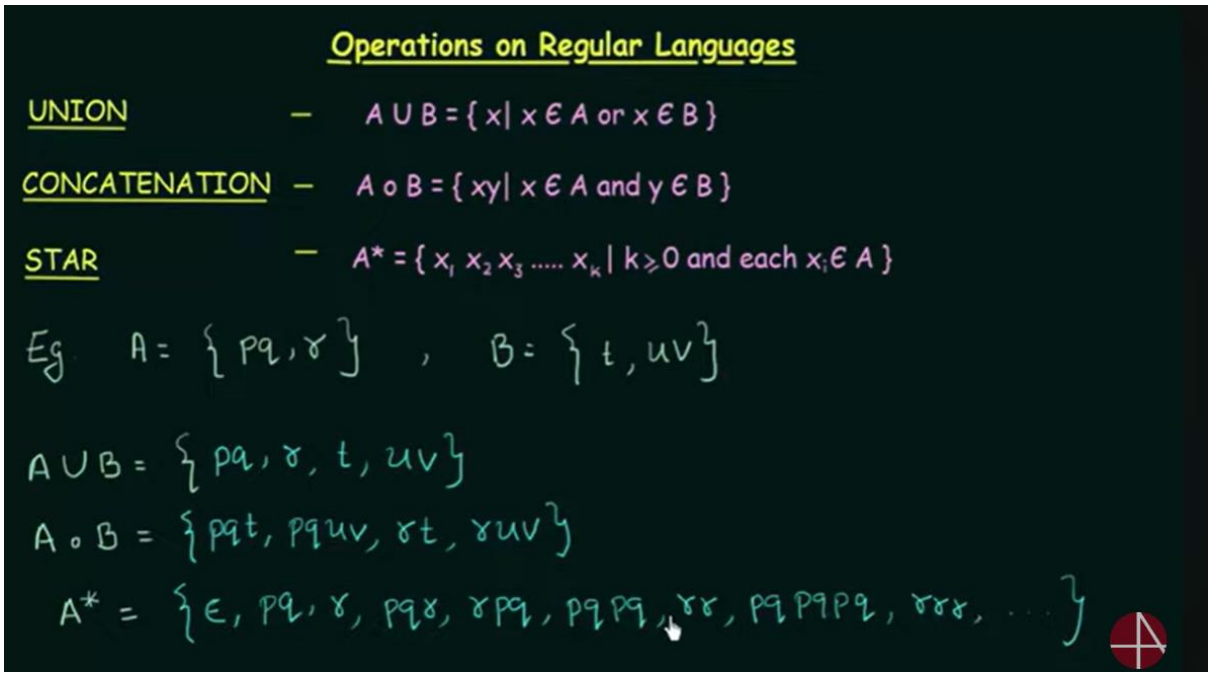
- Memory of FSM is very limited
-It cannot store or count strings

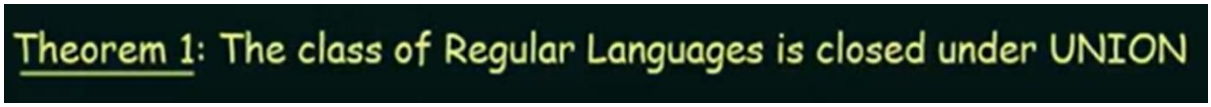
Example of non-regular Languages



These languages are not regular because we need to store and count strings respectively.

Operations of Regular Languages





The UNION of regular languages is also a regular language

Theorem 2: The class of Regular Languages is closed under CONCATENATION

NFA

NFA - Non-deterministic Finite Automata

Deterministic Finite Automata

↓

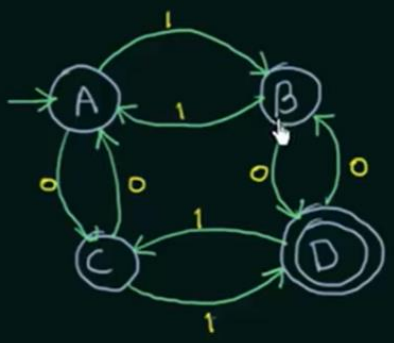
DETERMINISM

>> In DFA, given the current state
we know what the next state will be

>> It has only one unique next state

>> It has no choices or randomness

>> It is simple and easy to design



Non-deterministic Finite Automata

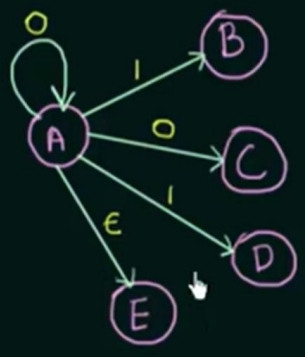
↓

NON-DETERMINISM

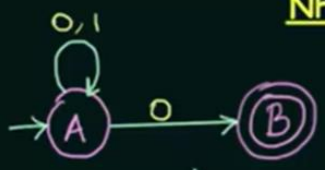
>> In NFA, given the current state
there could be multiple next states

>> The next state may be chosen at random

>> All the next states may be chosen in parallel



NFA - Formal Definition



L = { Set of all strings that end with 0 }

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

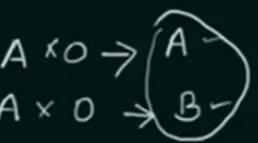
Q = Set of all states - {A, B}

Σ = inputs - {0, 1}

q_0 = Start State / initial State - A

F = Set of final states - B

δ = Q × Σ → __ - ?



A × 0 → A

A × 0 → B

A × 1 → A

B × 0 → ϕ

B × 1 → ϕ

A → A, B, AB, ϕ

F = Set of final states - B

δ = Q × Σ → 2^Q - ?

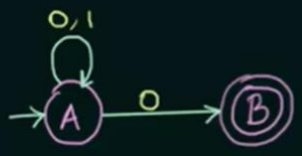
A → A, B, AB, ϕ - $2^2 - 4$

3 States - A, B, C

A → A, B, C, AB, AC, BC, ABC, ϕ

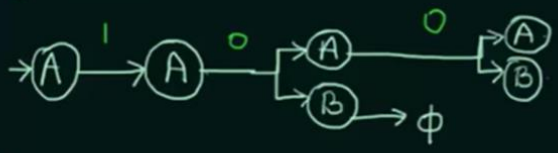
$2^3 - 8$

NFA - Example-1

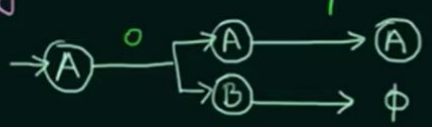


L = { Set of all strings that end with 0 }

Eg. 100



Eg. 01



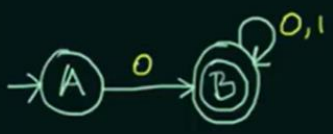
×

If there is any way to run the machine that ends in any set of states out of which atleast one state is a final state, then the NFA accepts

NFA - Example-2

L = { Set of all strings that start with 0 }

= { 0, 00, 01, 000, . . . }



Eg. 001 ✓

Eg. 101 X

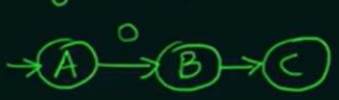
Dead configuration

>> Construct a NFA that accepts sets of all strings over {0,1} of length 2

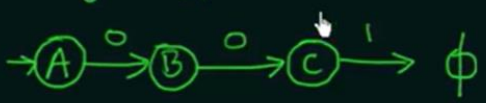
$\Sigma = \{0,1\}$

$L = \{00, 01, 10, 11\}$

Eg. 00 ✓

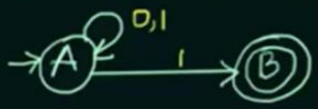


Eg. 001



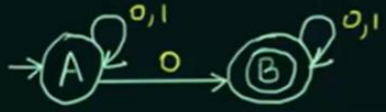
NFA - Example-3

Ex 1) L1 = { Set of all strings that ends with '1' }

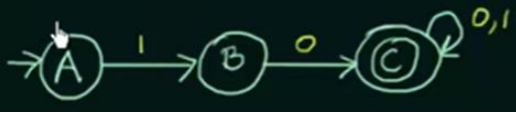


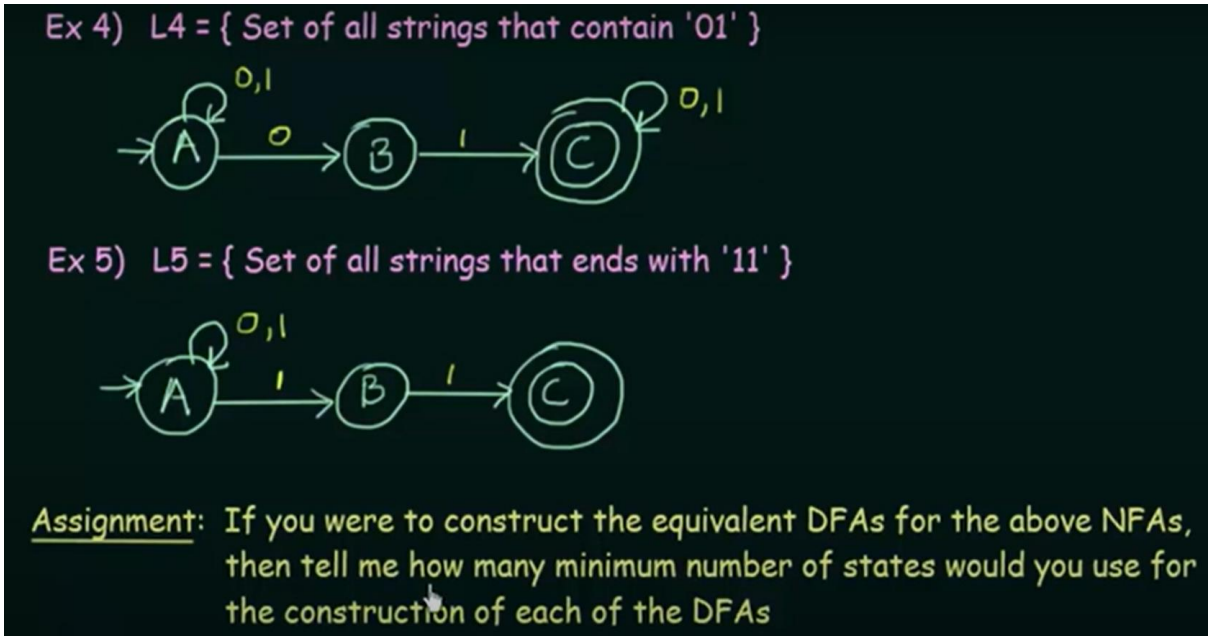
01, 001, 0001, 0*1, 1, 101, 1101,

Ex 2) L2 = { Set of all strings that contain '0' }

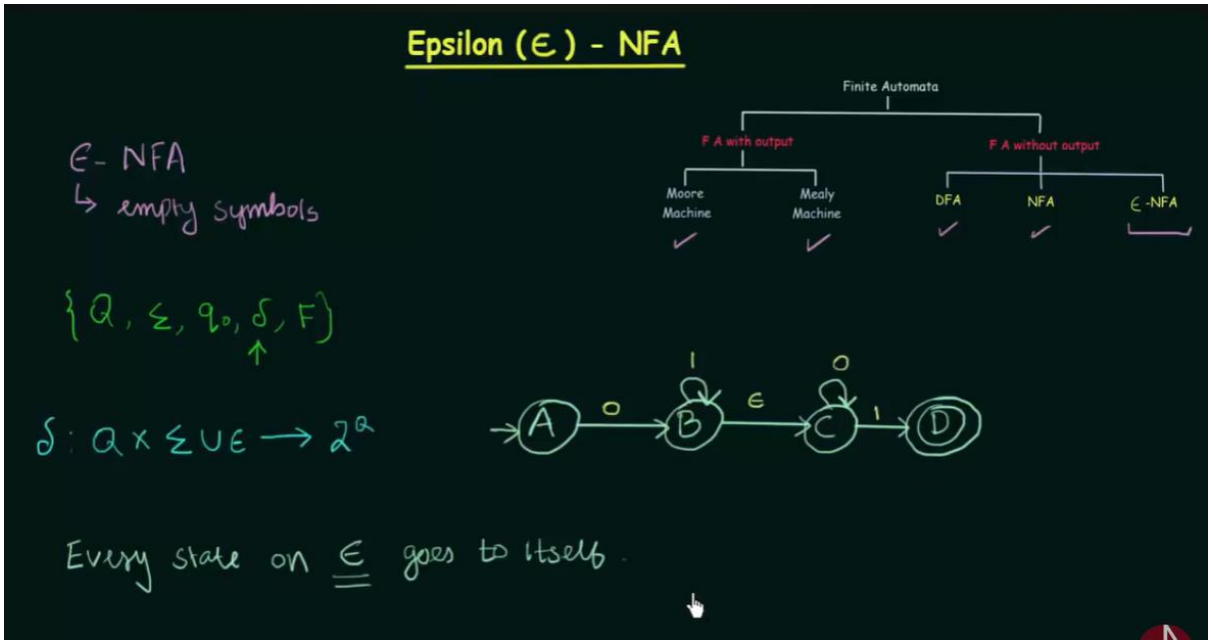


Ex 3) L3 = { Set of all strings that starts with '10' }

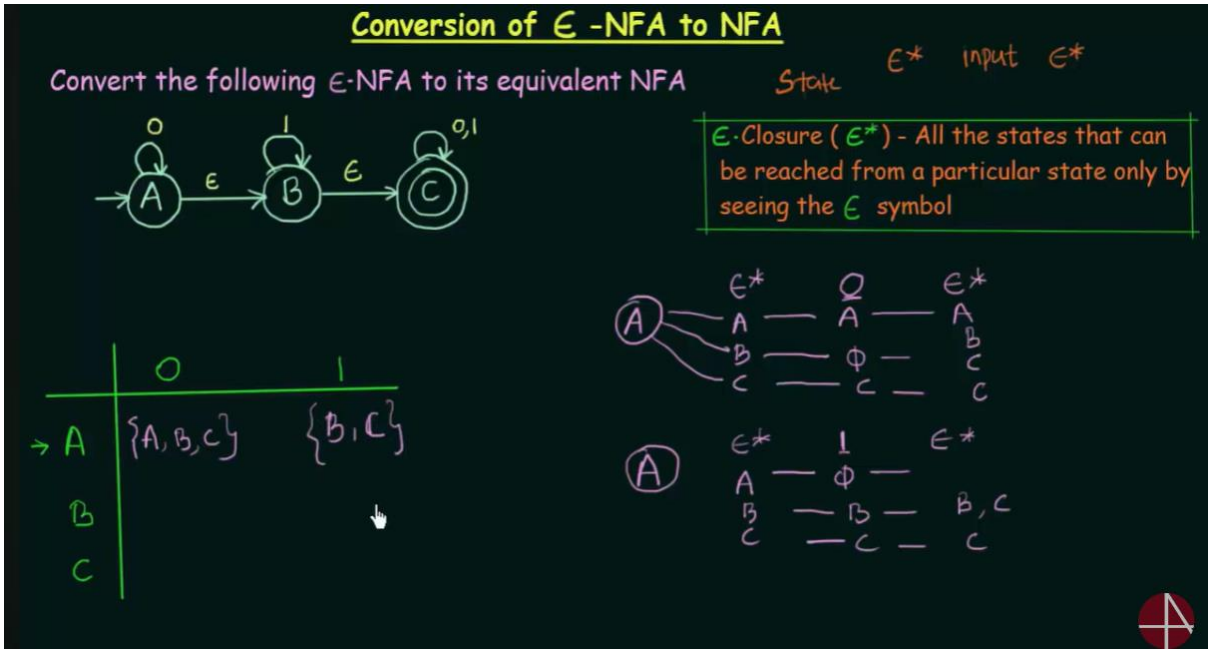


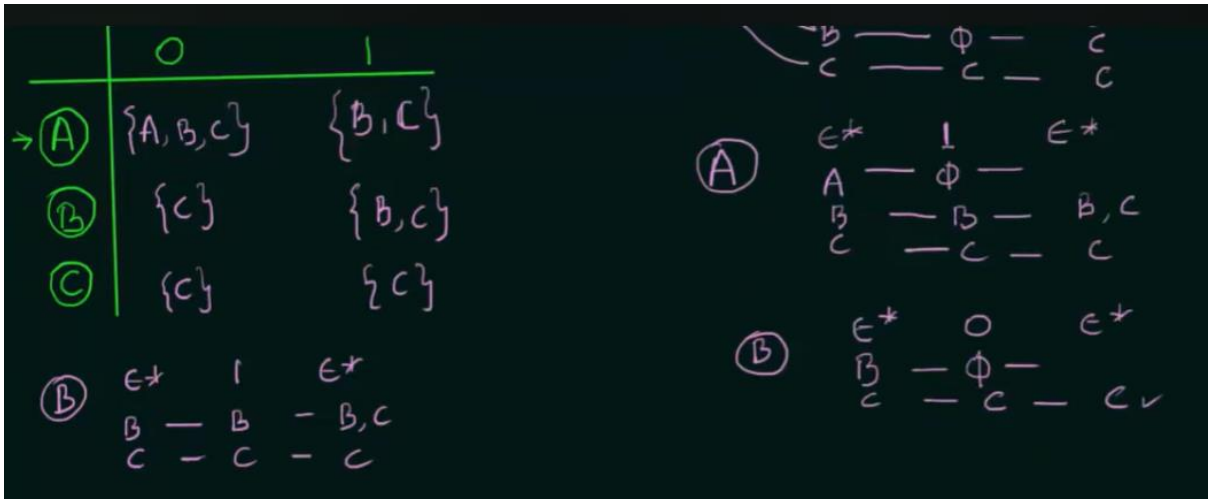


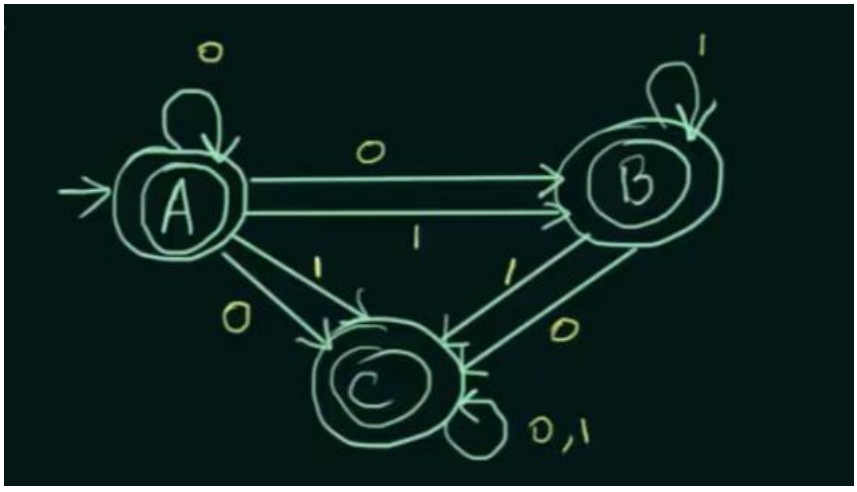
Epsilon NFA



Conversion of epsilon NFA to NFA



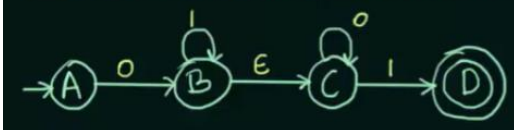




Any state which can reach the final state on just seeing the epsilon symbol is also a final state

Conversion of ϵ -NFA to NFA -Examples (Part-1)

Convert the following ϵ-NFA to its equivalent NFA

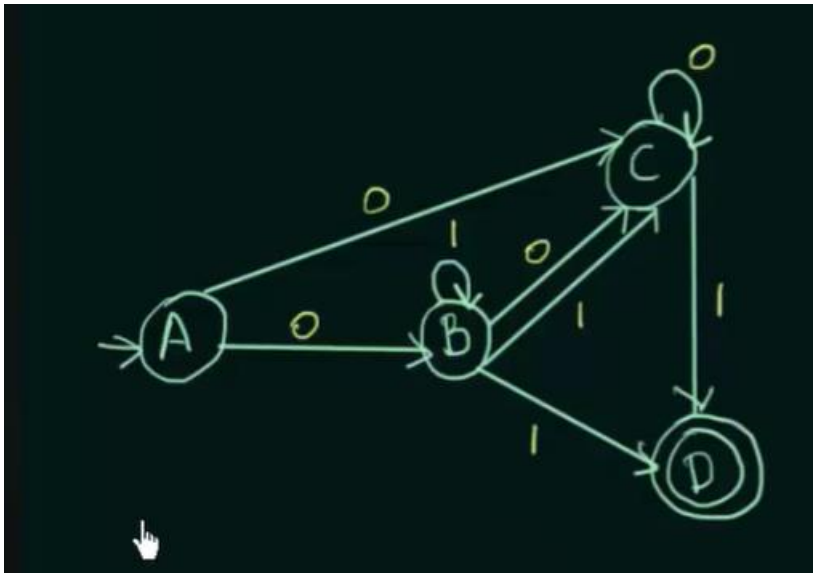


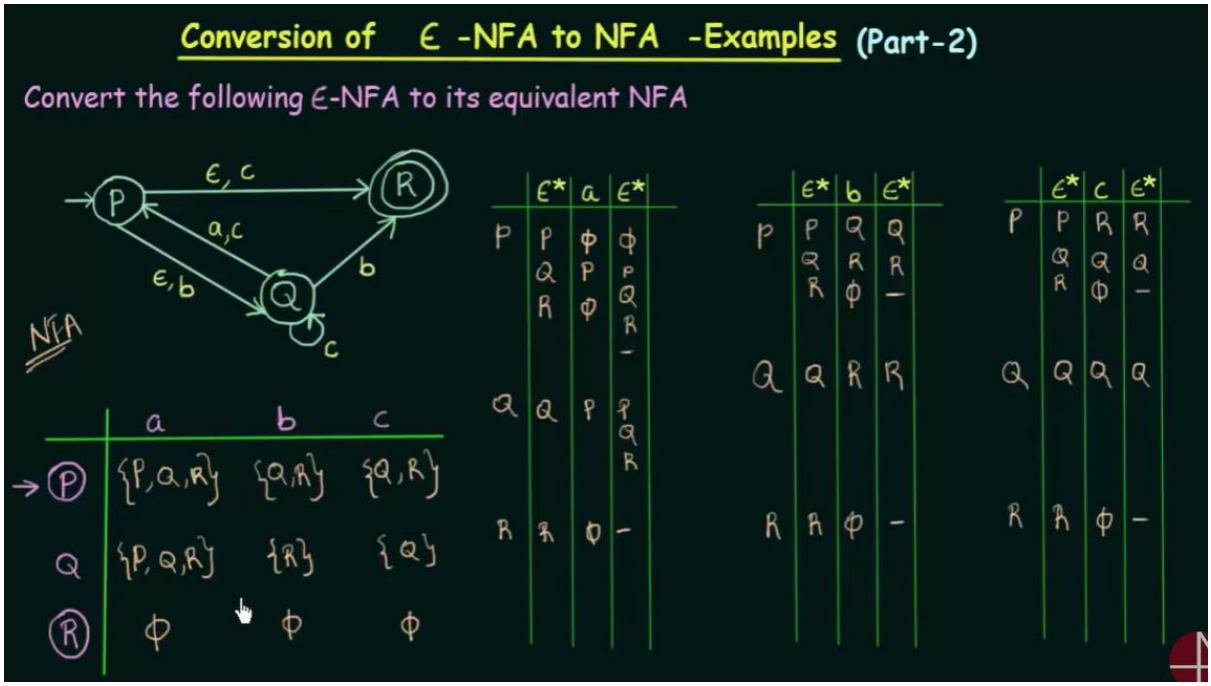
NFA

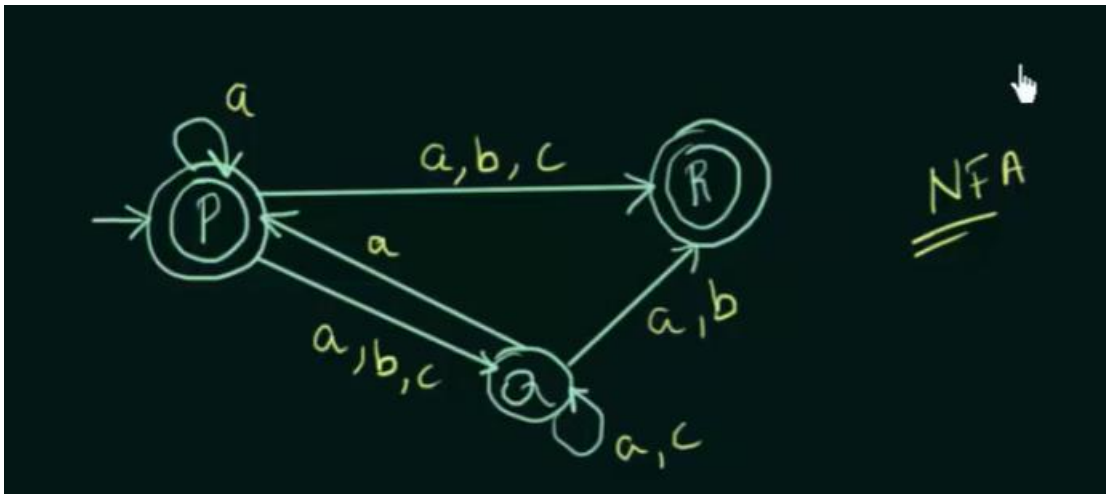
	0	1
→A	B, C	ϕ
B	C	B, C, D
C	C	D
D	ϕ	ϕ

	ϵ^*	0	ϵ^*
A	A	B	B C
B	B C	ϕ	ϕ C
C	C	ϕ	C
D	D	ϕ	ϕ

	ϵ^*	1	ϵ^*
A	A	ϕ	ϕ
B	B C	B D	B C D
C	C	D	D
D	D	ϕ	ϕ







Conversion of NFA to DFA

Conversion of NFA to DFA

Every DFA is an NFA, but not vice versa

But there is an equivalent DFA for every NFA

DFA

$\delta = Q \times \Sigma \rightarrow Q$

NFA

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

NFA $\cong$ DFA

L = { Set of all strings over (0,1) that starts with '0' }

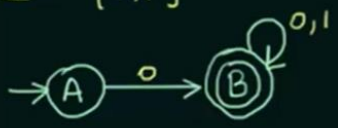
$\Sigma = \{0,1\}$

NFA

L = { Set of all strings over (0,1) that starts with '0' }

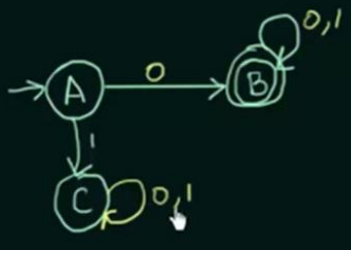
$\Sigma = \{0,1\}$

NFA



	0	1
A	B	ϕ
B	B	B

DFA



	0	1
A	B	C
B	B	B
C	C	C

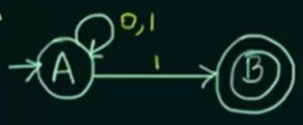
C - Dead state / Trap state

Conversion of NFA to DFA - Examples (Part 1)

L = { Set of all strings over (0,1) that ends with '1' }

$\Sigma = 0, 1$

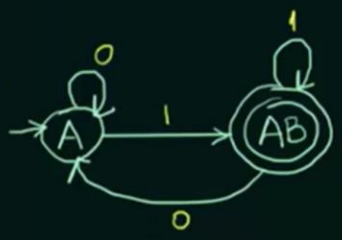
NFA



	0	1
A	{A}	{A, B}
B	ϕ	ϕ

Subset construction method

DFA



	0	1
A	{A}	{AB}
AB	{A}	{AB}

AB - Single State

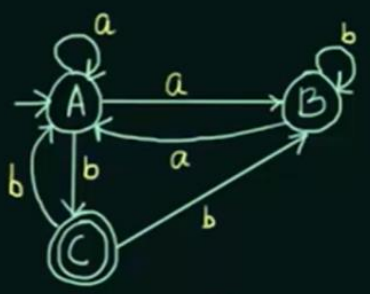
Conversion of NFA to DFA - Examples (Part-2)

Find the equivalent DFA for the NFA given by M= [{A,B,C}, (a,b), δ, A, {C}] where δ is given by:

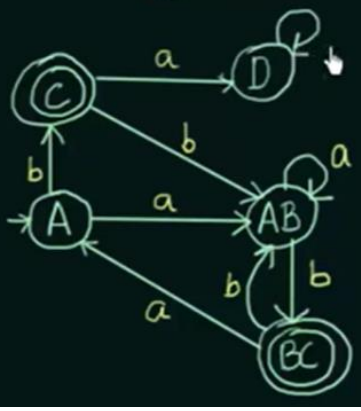
	a	b
→A	A,B	C
B	A	B
©C	–	A,B

δ is given by:

	a	b
→A	A,B	C
B	A	B
(C)	–	A,B



	a	b
→A	AB	C
AB	AB	BC
(BC)	A	AB
(C)	D	AB
D	D	D

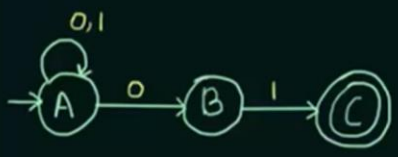


Conversion of NFA to DFA - Examples (Part-3)

Given below is the NFA for a language

L = { Set of all strings over (0,1) that ends with '01' }. Construct its equivalent DFA

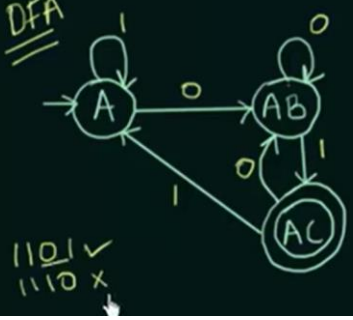
NFA



	0	1
→A	A,B	A
B	ϕ	C
(C)	ϕ	ϕ

	0	1
→A	AB	A
AB	AB	AC
(AC)	AB	A

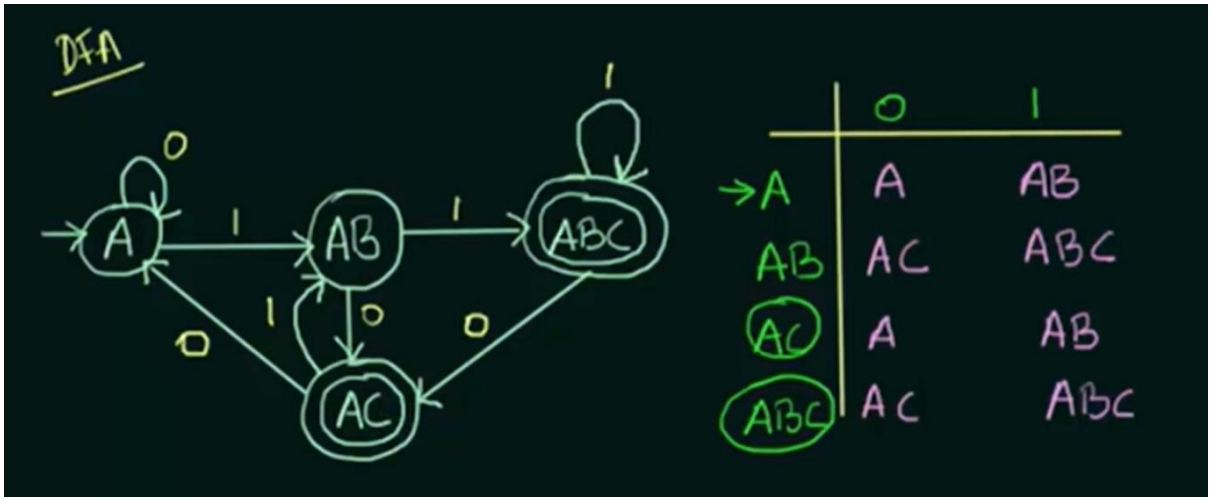
DFA

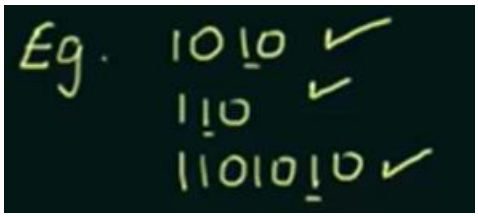


1101 ✓

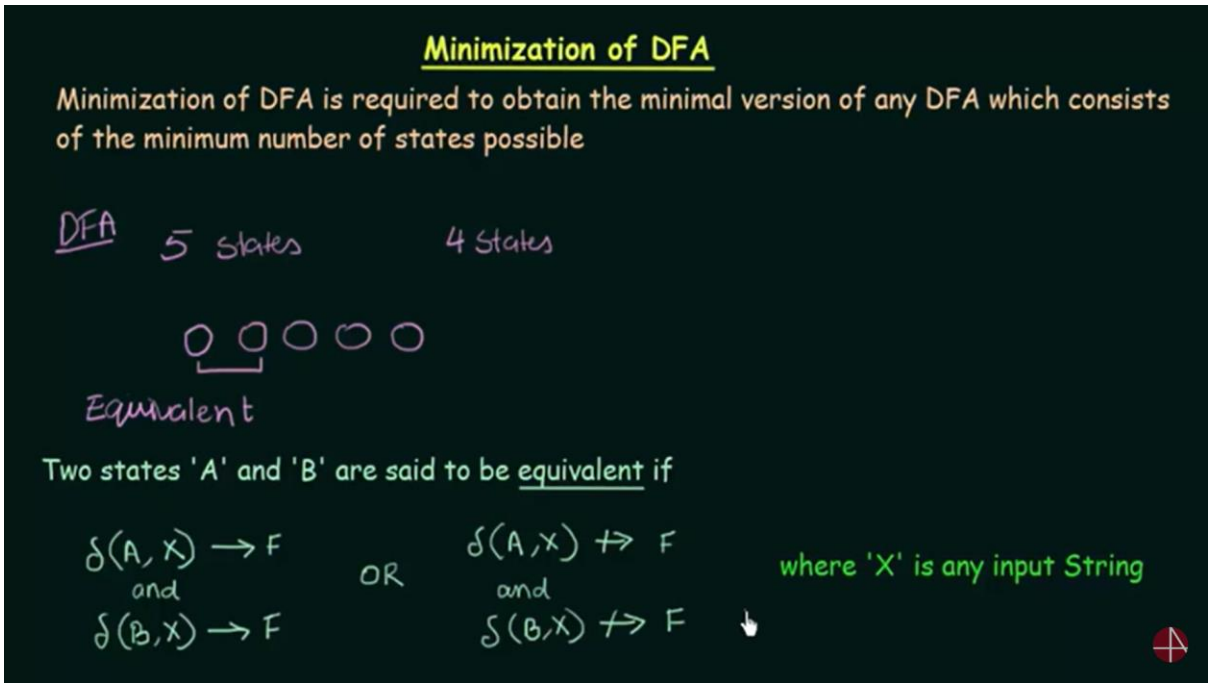
1110 ✗

	0	1
→A	AB	A
AB	AB	AC
(AC)	AB	A





Minimization of DFA



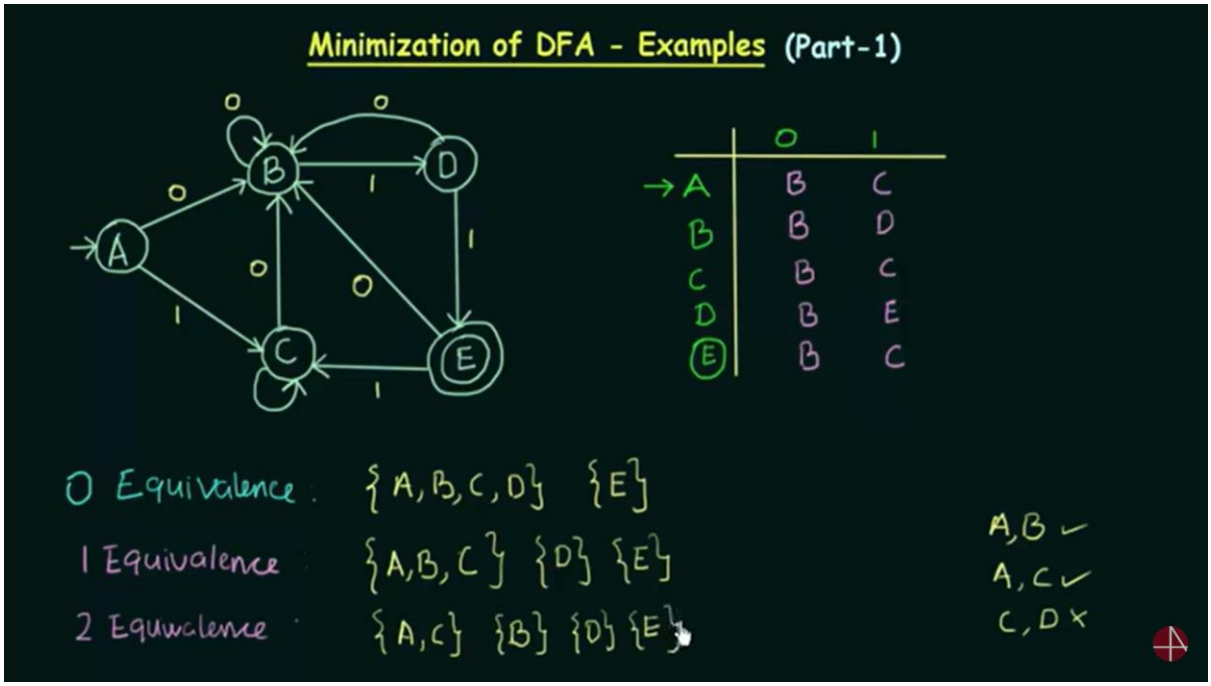
If |X| = 0, then A and B are said to be 0 equivalent

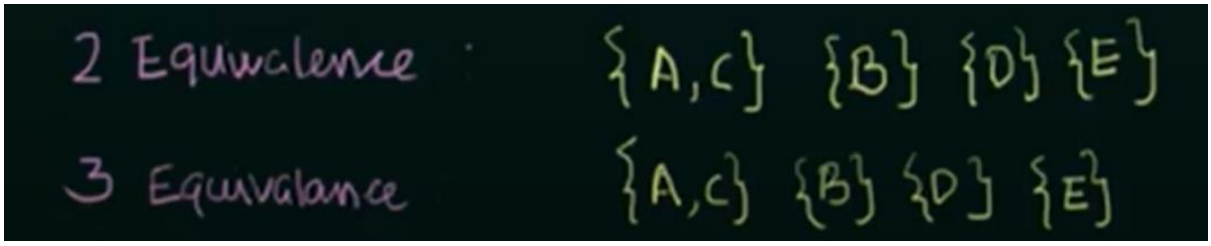
If |X| = 1, then A and B are said to be 1 equivalent

If |X| = 2, then A and B are said to be 2 equivalent

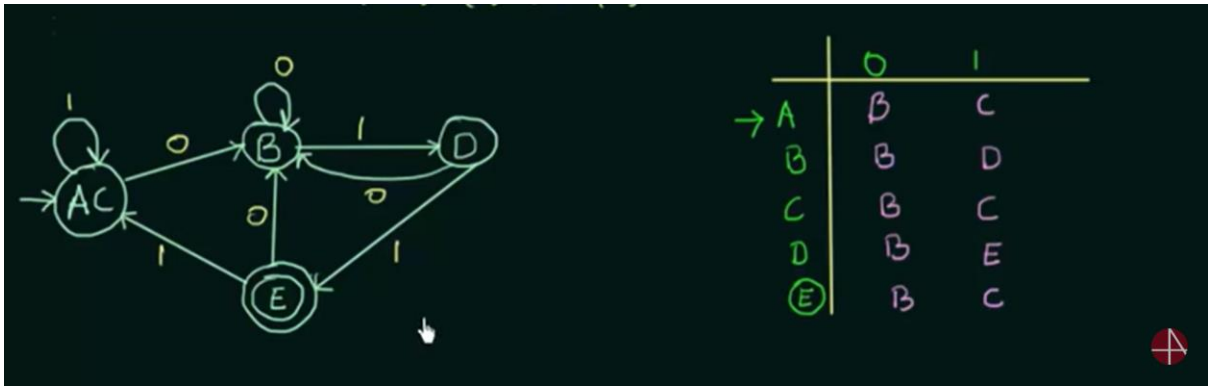
⋮

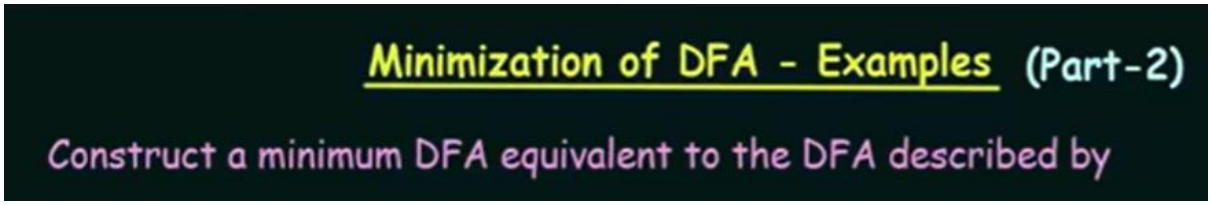
If |X| = n, then A and B are said to be n equivalent

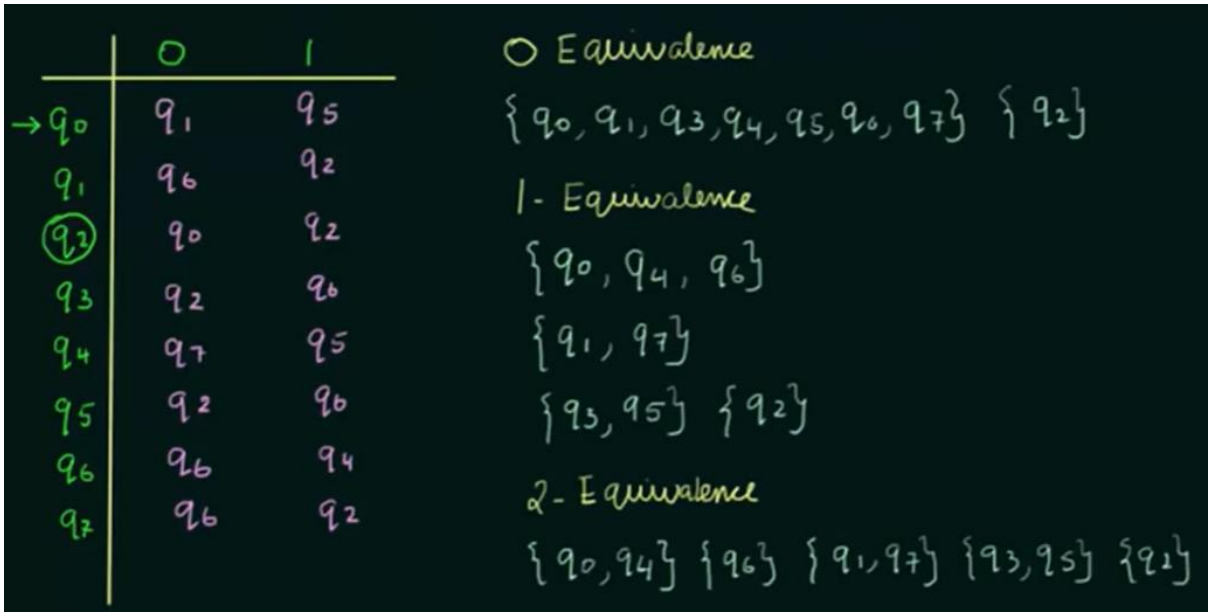


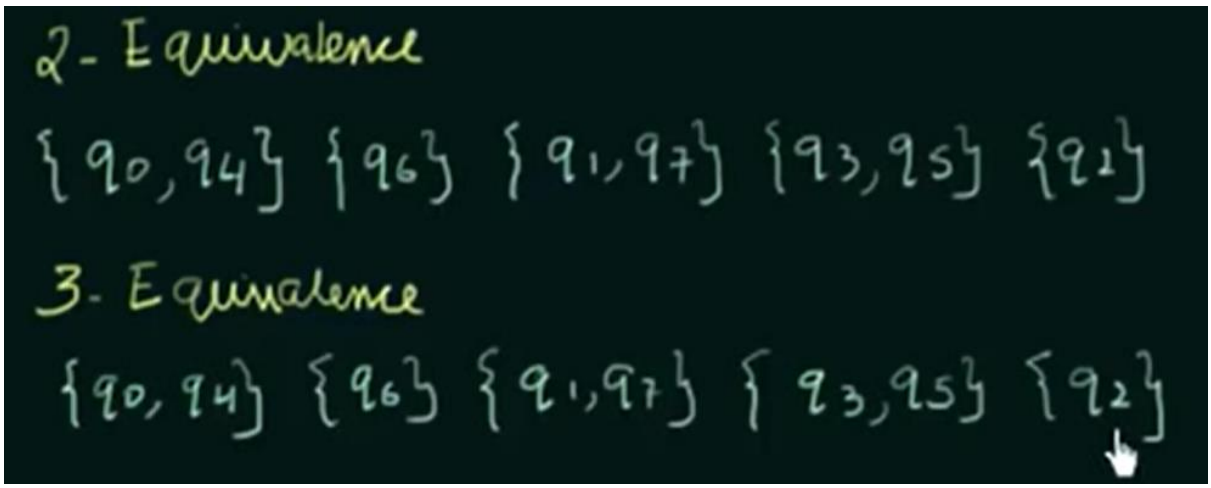


When you see two consecutive equivalences giving the same result, you can stop the process. No need to go further.

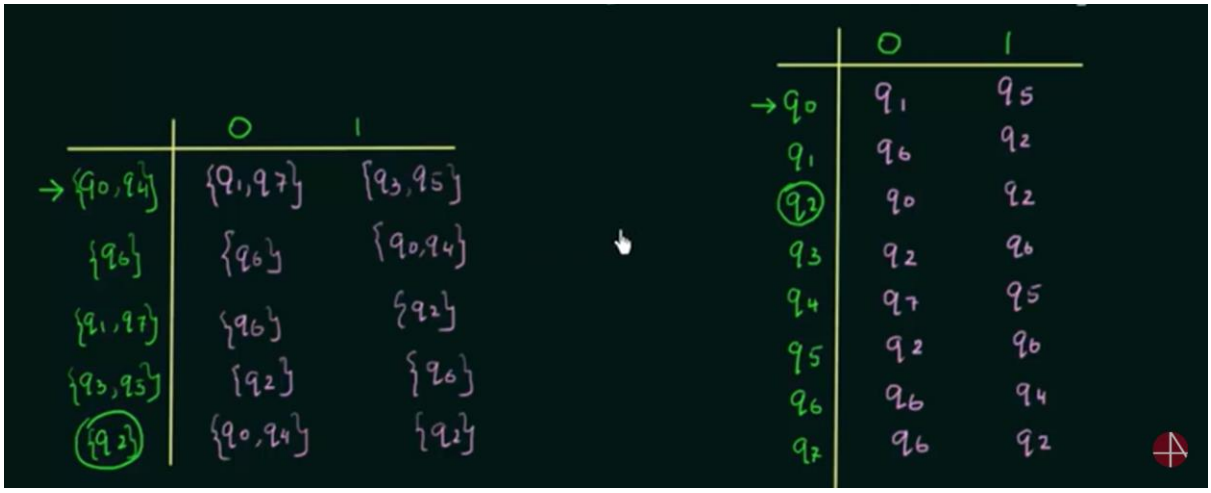




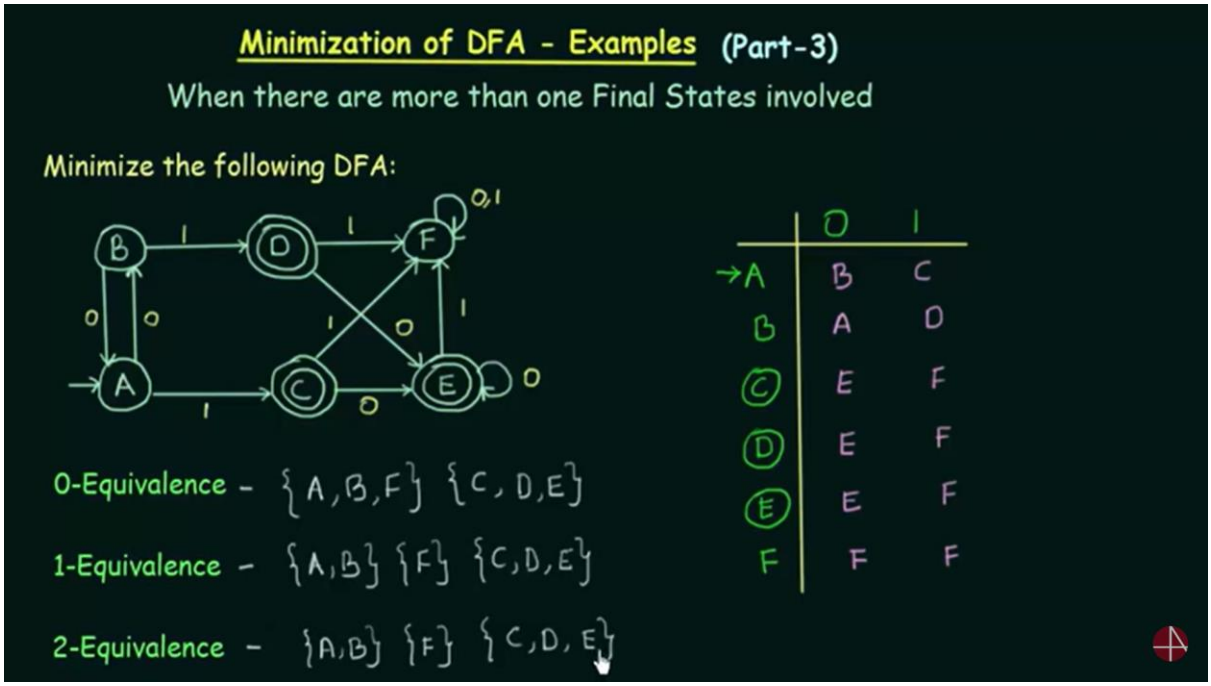




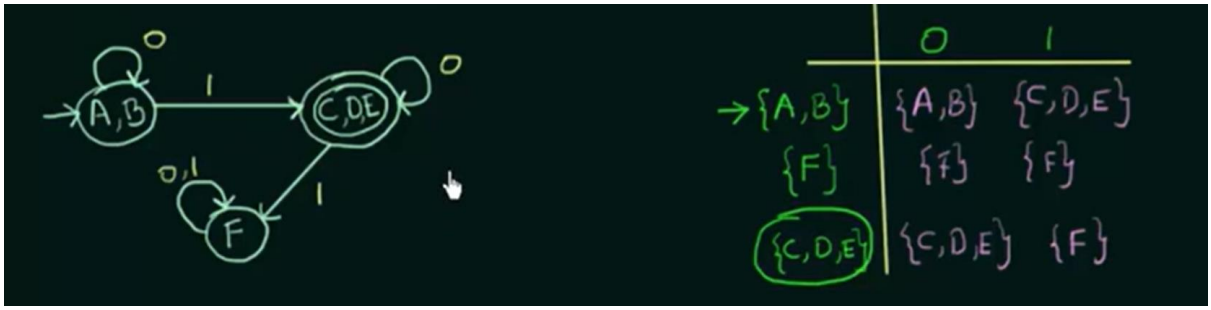
2 steps with the same output found so stop the process.



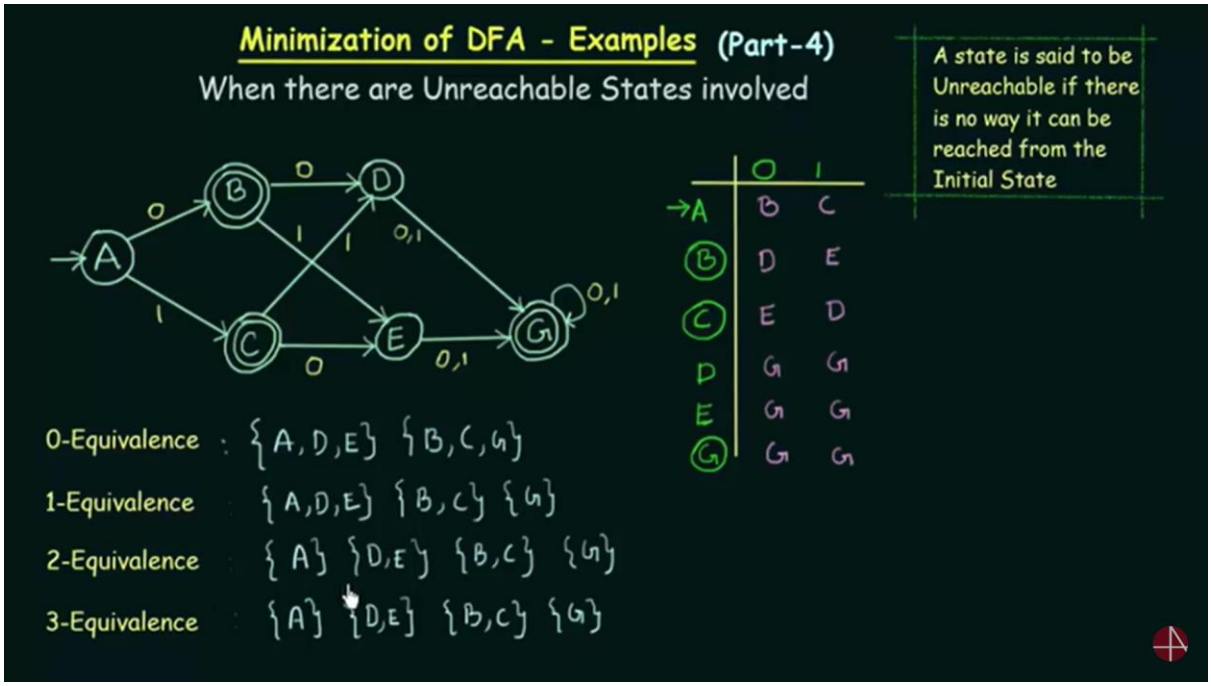
Minimization with multiple final states



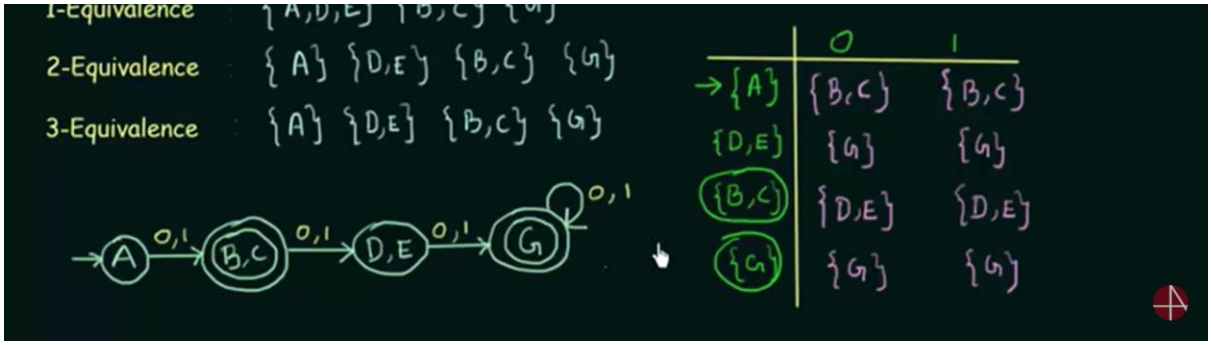
2 steps with the same output found so stop the process.



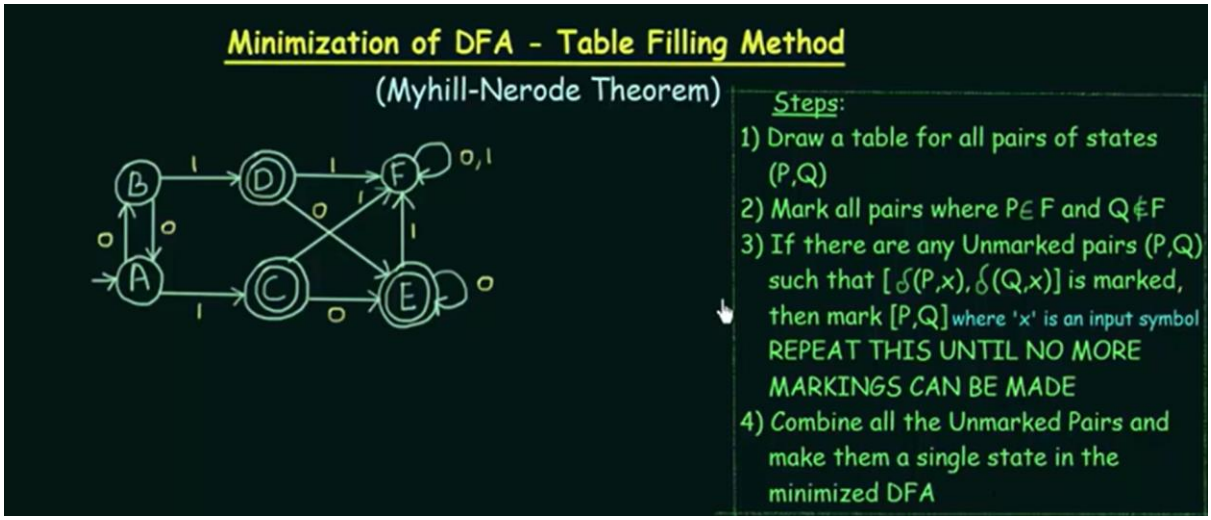
Minimization with unreachable state



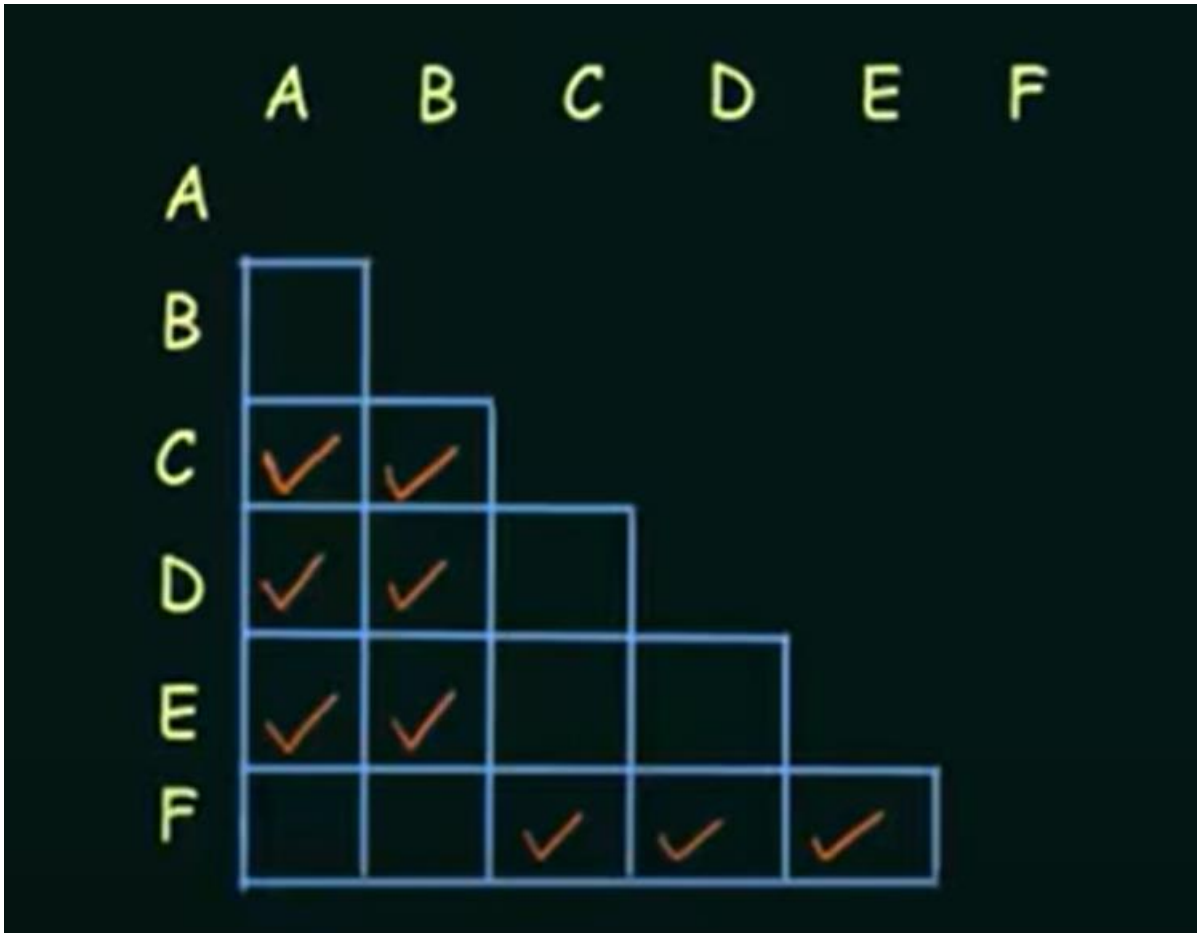
Simple remove the unreachable state.



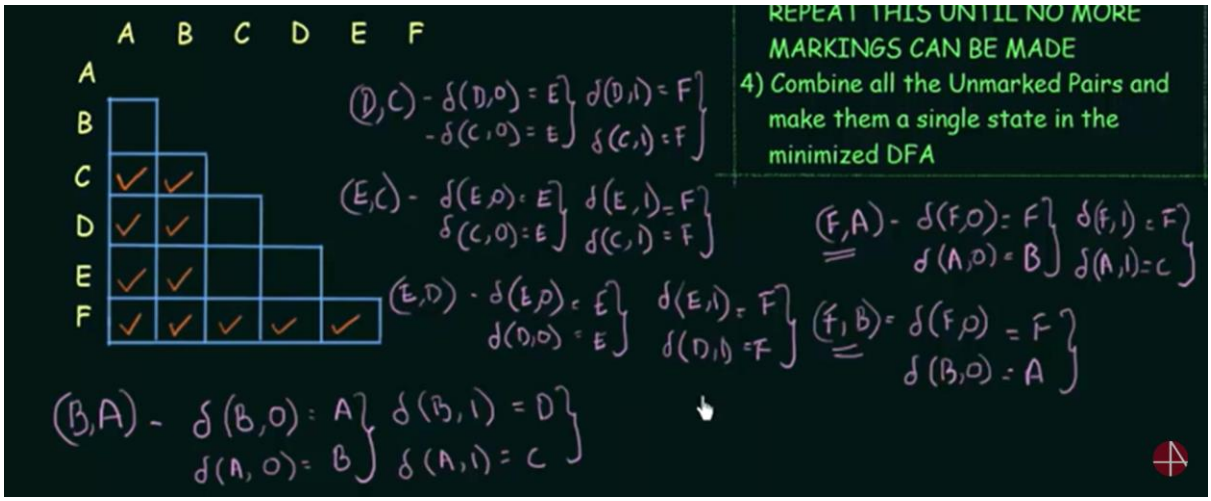
Myhill-Nerode Theorem – Minimization of DFA using Table-filling method



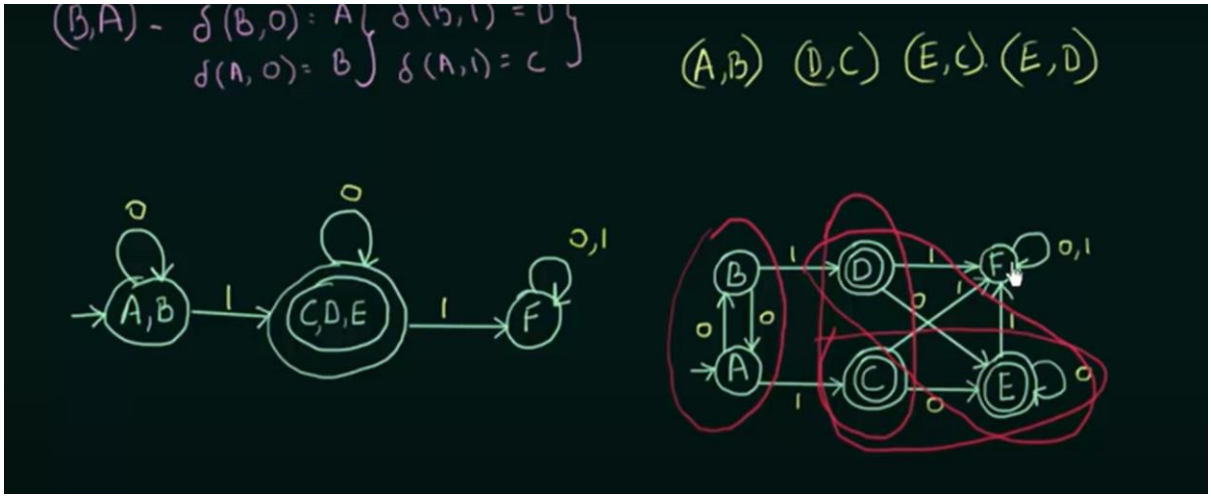
After completing 2nd step



After completing 3rd step

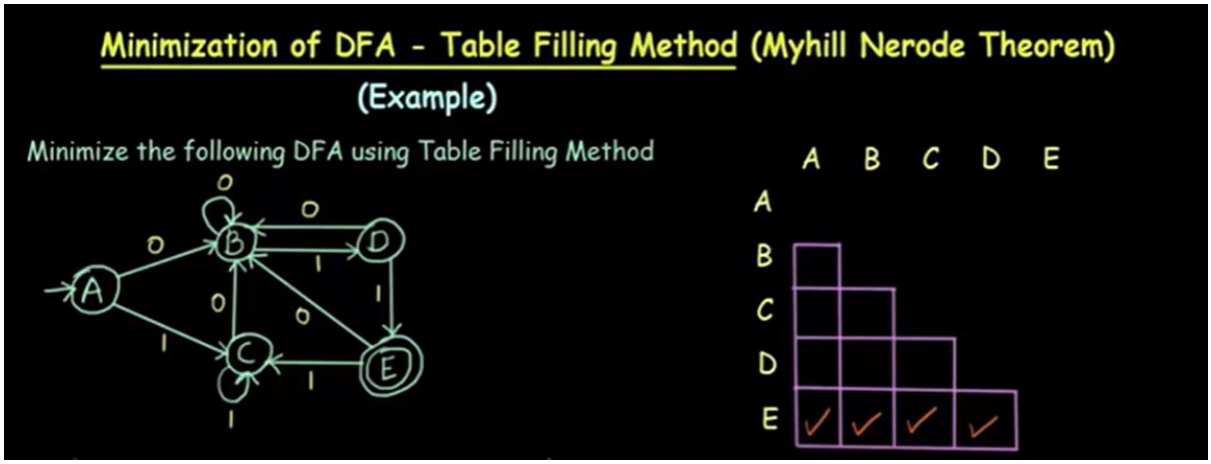


After completing 4th step

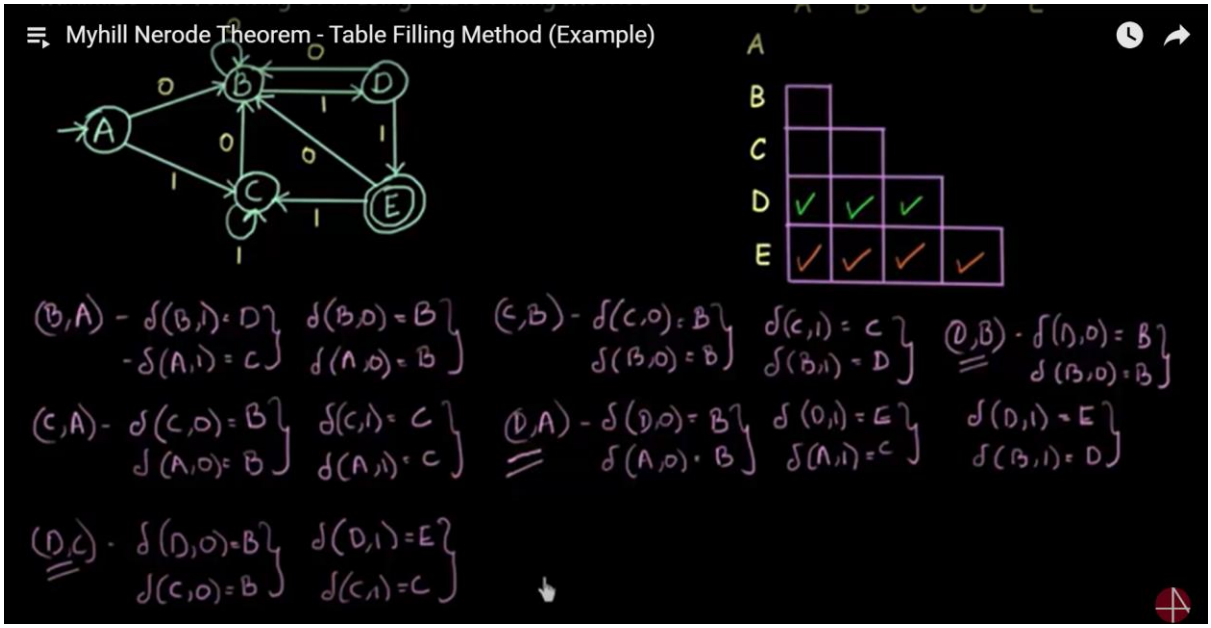


Example 2

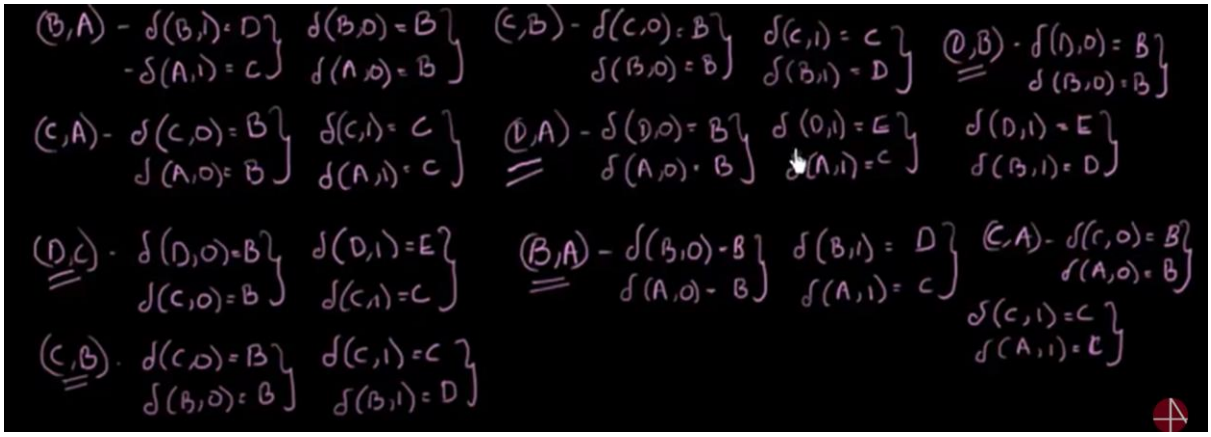
After completing 2nd step

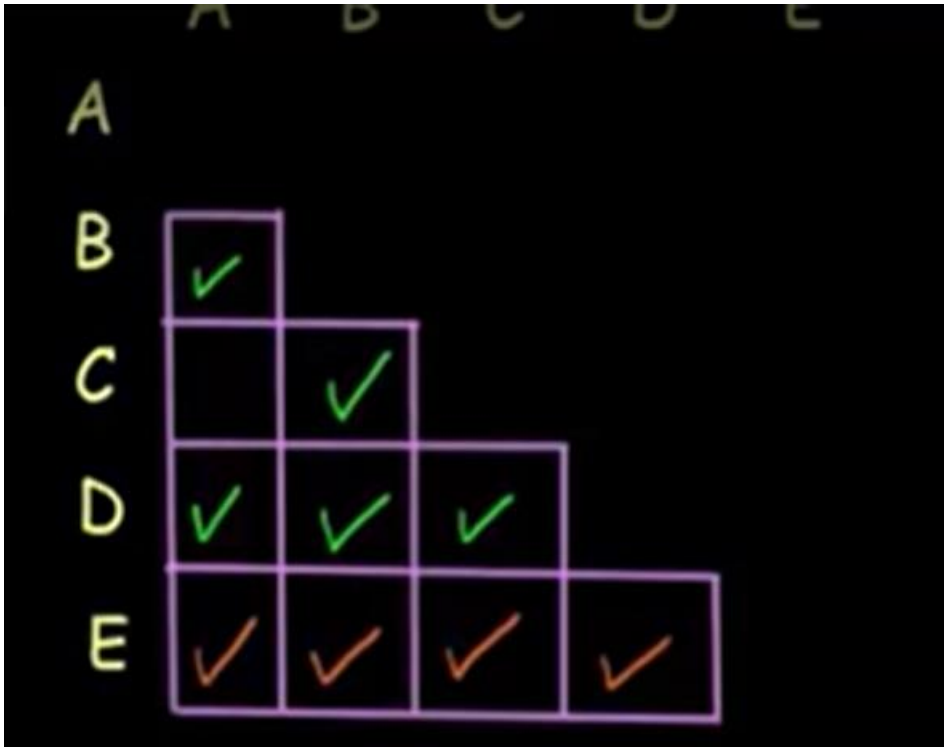


After completing 1st iteration of 3rd step



After completing 2nd iteration of 3rd step





After completing 4th step

Equivalence of two finite automata

Equivalence of two Finite Automata

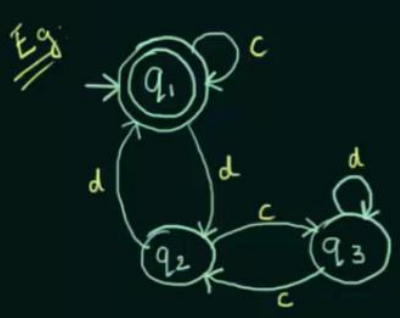
Steps to identify equivalence

1) For any pair of states $\{q_i , q_j\}$ the transition for input $a \in \Sigma$ is defined by $\{q_a , q_b\}$ where $\delta\{q_i , a\} = q_a$ and $\delta\{q_j , a\} = q_b$

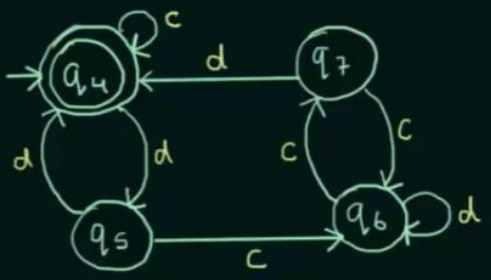
The two automata are not equivalent if for a pair $\{q_a , q_b\}$ one is INTERMEDIATE State and the other is FINAL State.

2) If Initial State is Final State of one automaton, then in second automaton also Initial State must be Final State for them to be equivalent.

Eg



A



B

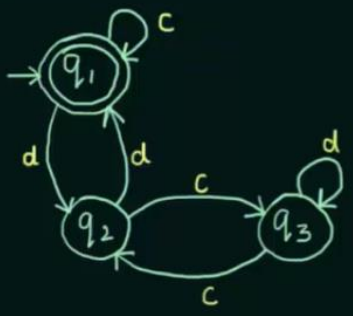
States	c	d
(q_1, q_4)	(q_1, q_4) FS FS	(q_2, q_5) IS IS
(q_2, q_5)	(q_3, q_6) IS IS	(q_1, q_4) FS FS

States	c	d
(q_1, q_4)	(q_1, q_4) FS FS	(q_2, q_5) IS IS
(q_2, q_5)	(q_3, q_6) IS IS	(q_1, q_4) FS FS
(q_3, q_6)	(q_2, q_7) IS IS	(q_3, q_6) IS IS
(q_2, q_7)	(q_3, q_6) IS IS	(q_1, q_4) FS FS

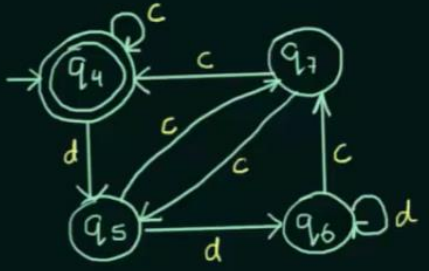
A and B are equivalent

Equivalence of two Finite Automata (Example)

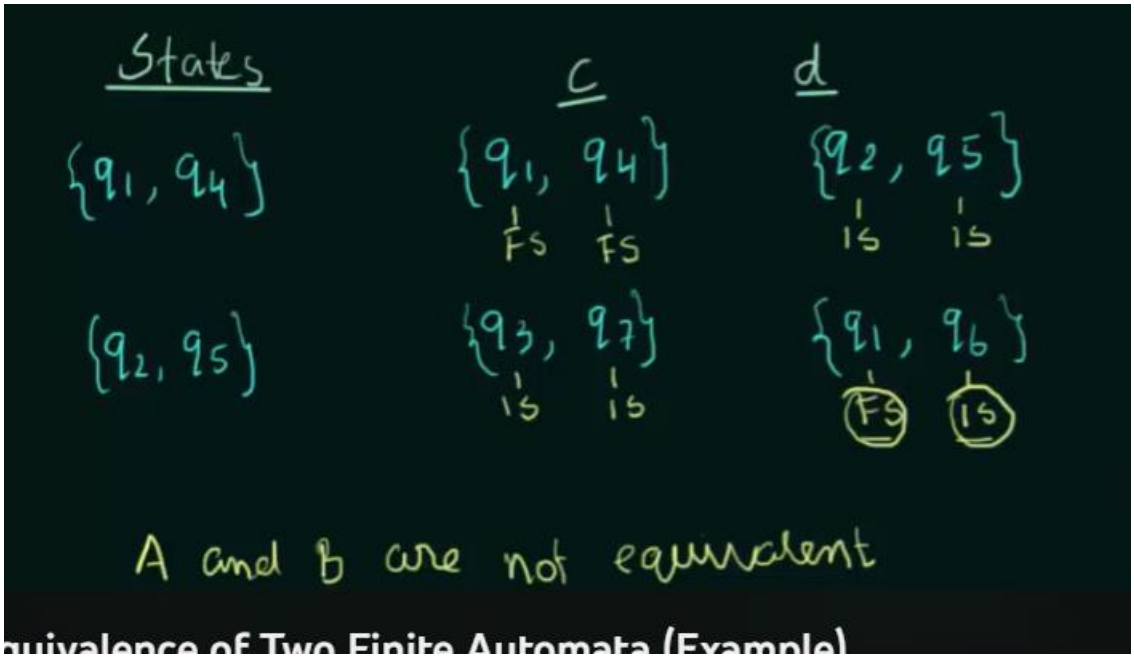
Find out whether the following automata are equivalent or not



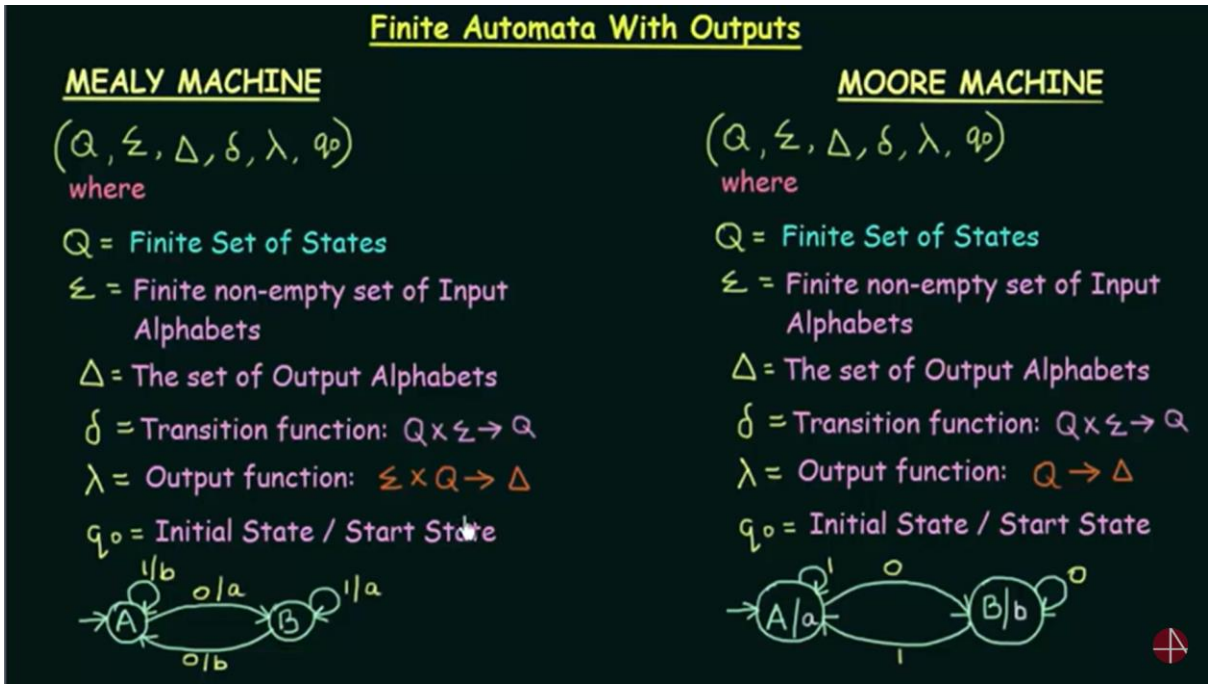
A

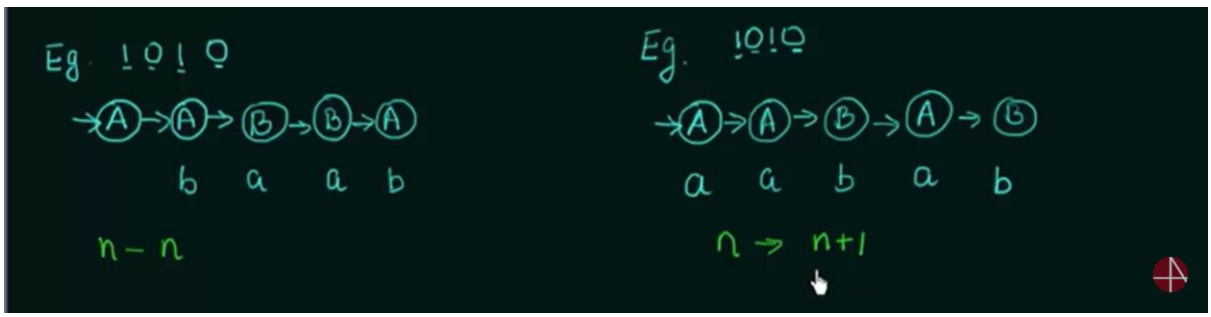


B

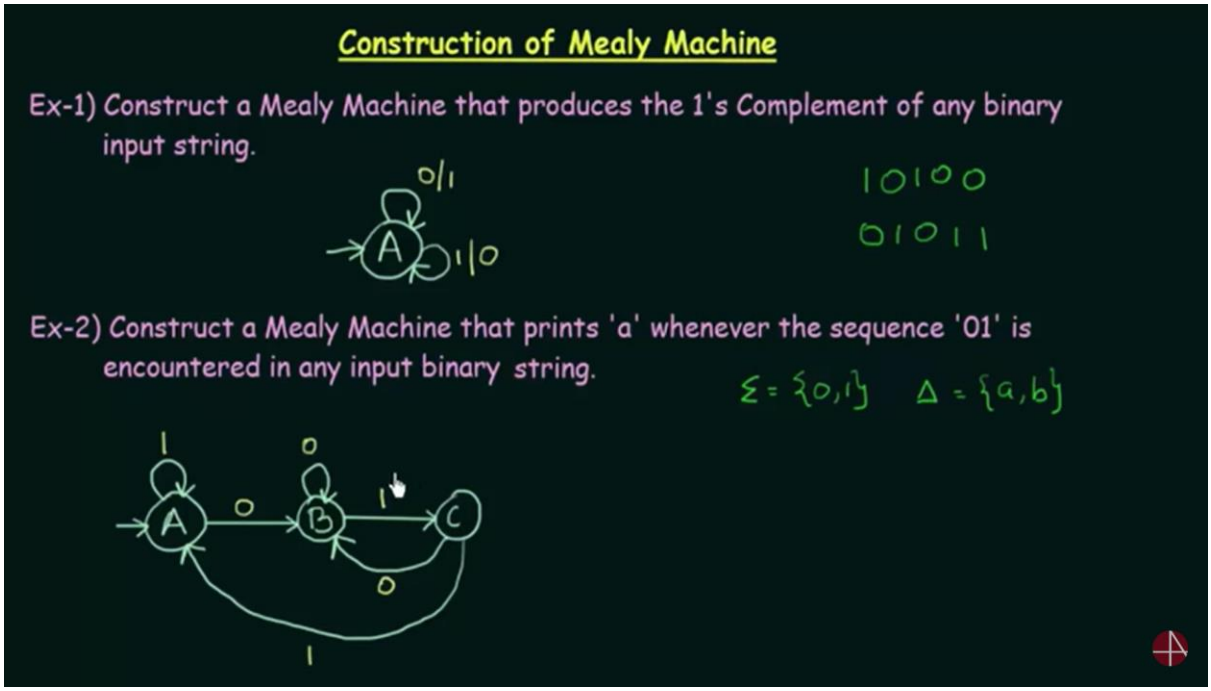


Finite Automata with outputs

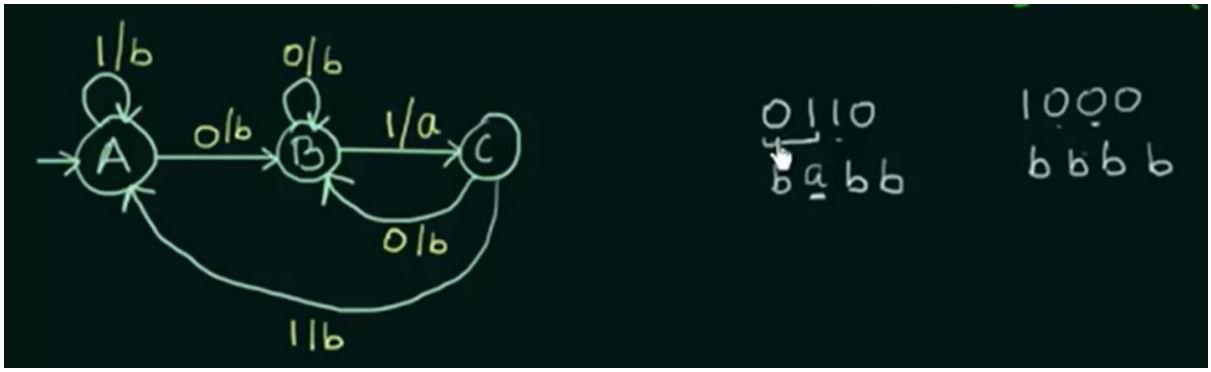




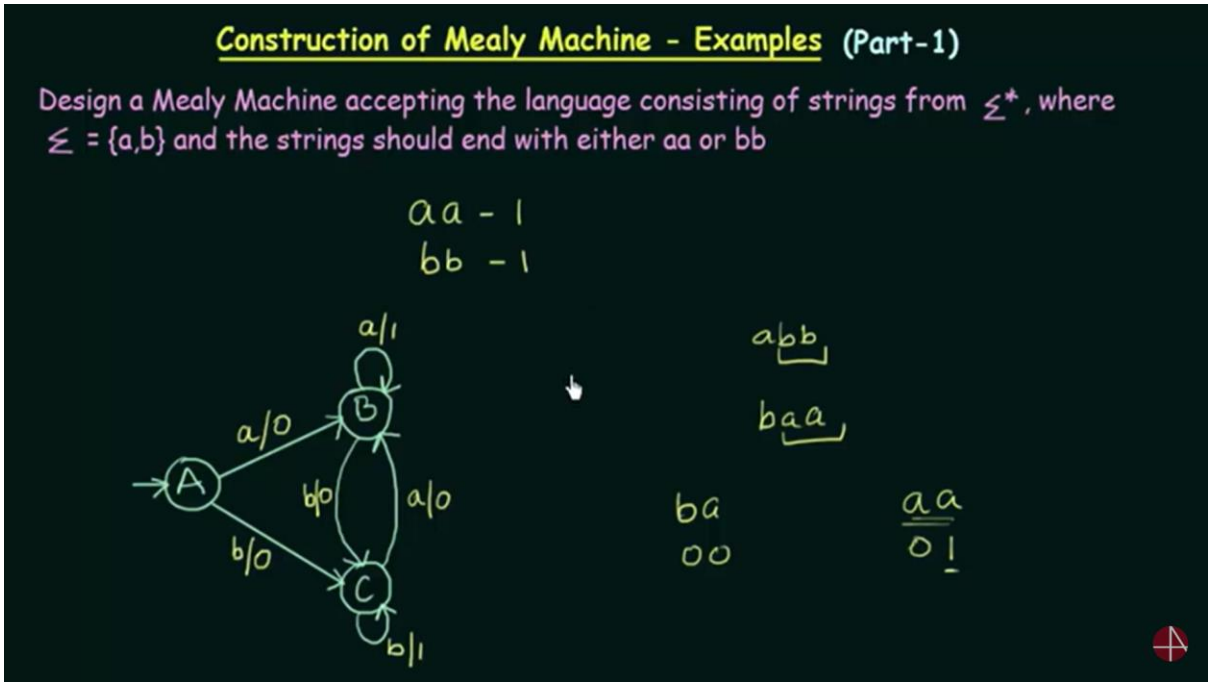
Construction of Mealy Machine

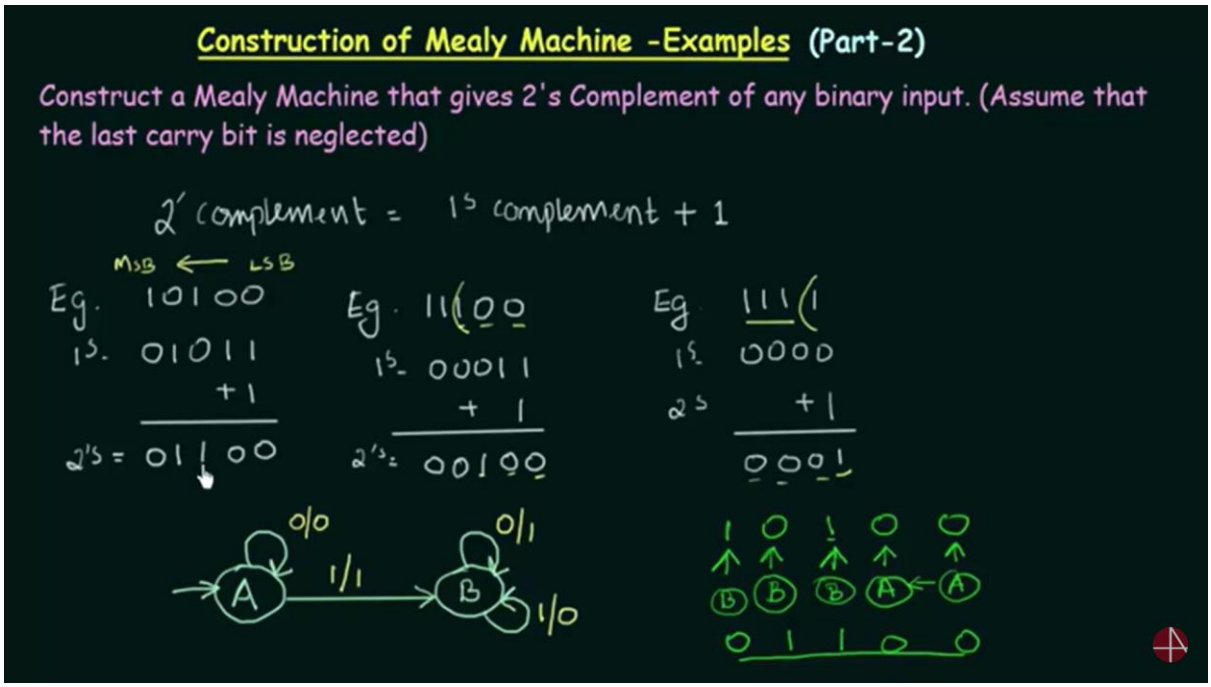


In the 2nd part first make the DFA and then convert it to a Mealy Machine



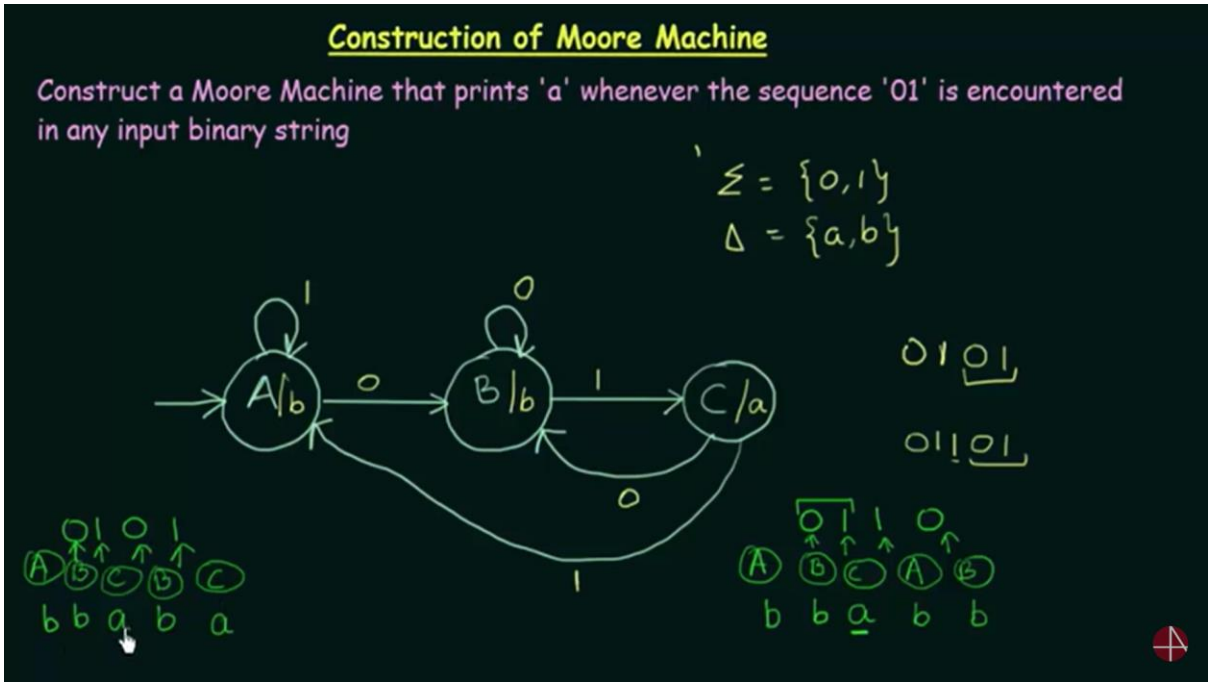
Example 1



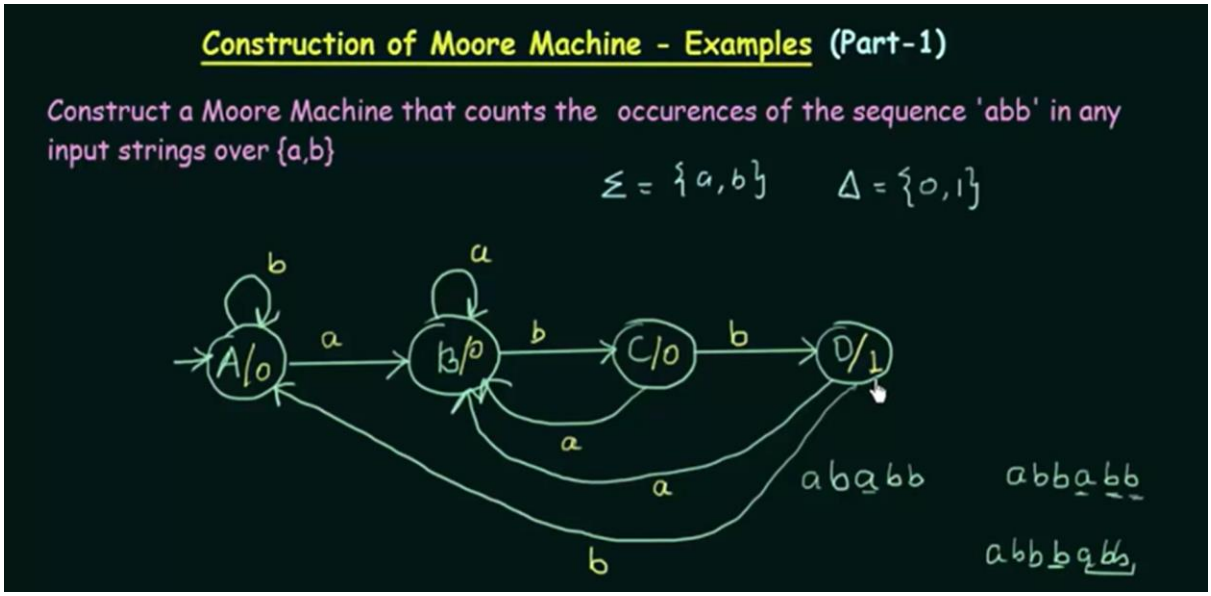


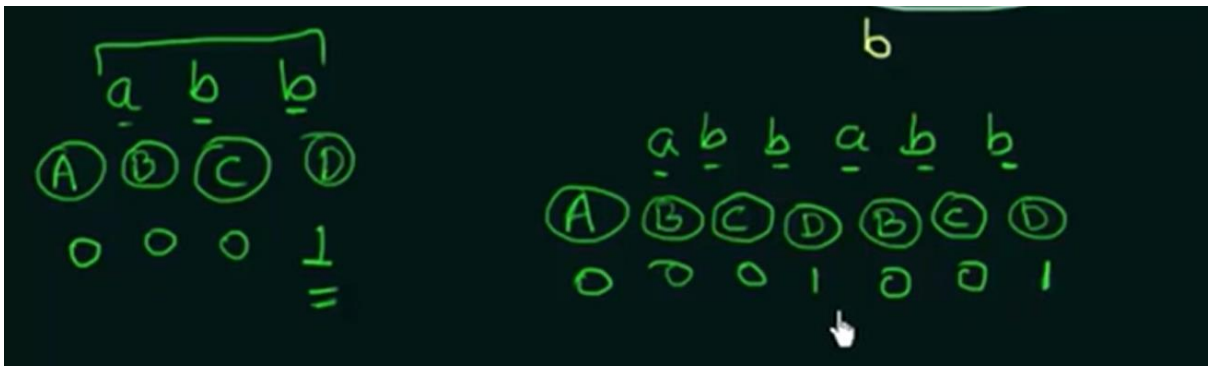
For finding two's complement, go from LSB to MSB and keep the digits same before encountering the first one and after the first one take1's complement of all digits.

Construction of Moore Machine


Construction of Moore Machine
Construct a Moore Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string
Σ = {0,1}
Δ = {a,b}
A/b
B/b
C/a
0
1
0 1 0 1
0 1 1 0 1
b b a b a
b b a b b

Example 1


Construction of Moore Machine - Examples (Part-1)
Construct a Moore Machine that counts the occurences of the sequence 'abb' in any input strings over {a,b}
Σ = {a,b}
Δ = {0,1}
A/0
B/0
C/0
D/1
a b a b b
a b b a b b
a b b b a b b


a b b
A B C D
0 0 0 1
b
a b b a b b
A B C D B C D
0 0 0 1 0 0 1

Construction of Moore Machine- Examples (Part-2)

For the following Moore Machine the input alphabet is $\Sigma = \{a,b\}$ and the output alphabet is $\Delta = \{0,1\}$. Run the following input sequences and find the repective outputs:
(i) aabab (ii) abbb (iii) ababb

States	a	b	Outputs
$\rightarrow q_0$	q_1	q_2	0
q_1	q_2	q_3	0
q_2	q_3	q_4	1
q_3	q_4	q_4	0
q_4	q_0	q_0	0

i) a a b a b

q_0 q_1 q_2 q_4 q_0 q_2

0 0 1 0 0 1

ii) a b b b

q_0 q_1 q_3 q_4 q_0

0 0 0 0 0

iii) a b a b b

q_0 q_1 q_3 q_4 q_0 q_2

0 0 0 0 0 1

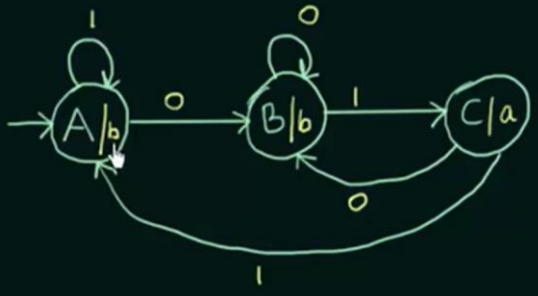
Conversion of Moore Machine to Mealy Machine

Conversion of Moore Machine to Mealy Machine

Construct a Moore Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string and then CONVERT IT TO ITS EQUIVALENT MEALY MACHINE

$\Sigma = \{0,1\}$ $\Delta = \{a,b\}$

Moore Machine $\Longleftrightarrow$ Mealy Machine



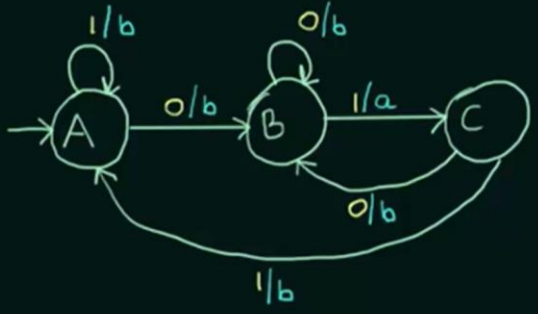
State	0	1	Output
$\rightarrow$ A	B	A	b
B	B	C	b
C	B	A	a

Conversion of Moore Machine to Mealy Machine

Construct a Moore Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string and then CONVERT IT TO ITS EQUIVALENT MEALY MACHINE

$\Sigma = \{0, 1\}$ $\Delta = \{a, b\}$

Moore Machine $\Longleftrightarrow$ Mealy Machine

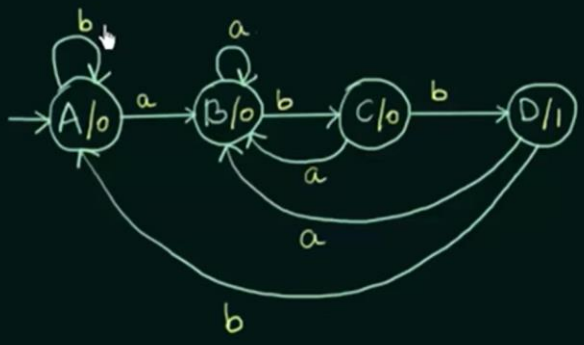


State	0	1
→ A	B, b	A, b
B	B, b	C, a
C	B, b	A, b

Conversion of Moore Machine to Mealy Machine - Examples (Part-1)

The given Moore Machine counts the occurences of the sequence 'abb' in any input binary strings over {a,b}. CONVERT IT TO ITS EQUIVALENT MEALY MACHINE

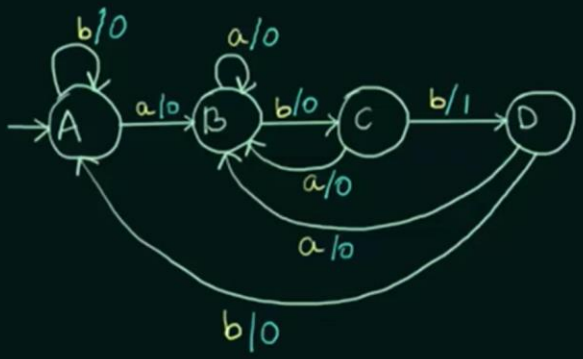
$\Sigma = \{a, b\}$ $\Delta = \{0, 1\}$



Conversion of Moore Machine to Mealy Machine - Examples (Part-1)

The given Moore Machine counts the occurences of the sequence 'abb' in any input binary strings over {a,b}. CONVERT IT TO ITS EQUIVALENT MEALY MACHINE

$\Sigma = \{a, b\}$ $\Delta = \{0, 1\}$



State	a	b
→A	B, 0	A, 0
B	B, 0	C, 0
C	B, 0	D, 1
D	B, 0	A, 0

Conversion of Moore Machine to Mealy Machine- Examples (Part-2)

Convert the given Moore Machine to its equivalent Mealy Machine.

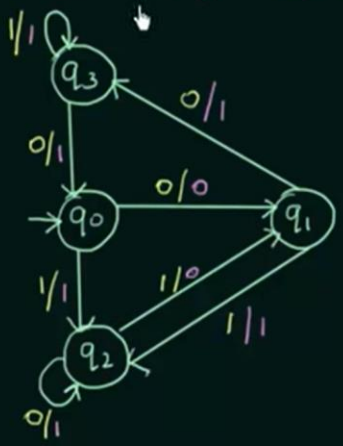
Moore

State	0	1	Output
→q_0	q_1	q_2	1
q_1	q_3	q_2	0
q_2	q_2	q_1	1
q_3	q_0	q_3	1

Mealy

State	0	1
→q_0	q_1, 0	q_2, 1
q_1	q_3, 1	q_2, 1
q_2	q_2, 1	q_1, 0
q_3	q_0, 1	q_3, 1

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



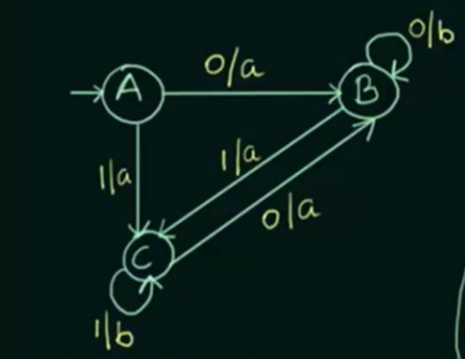
Conversion of Mealy Machine to Moore Machine

Conversion of Mealy Machine to Moore Machine

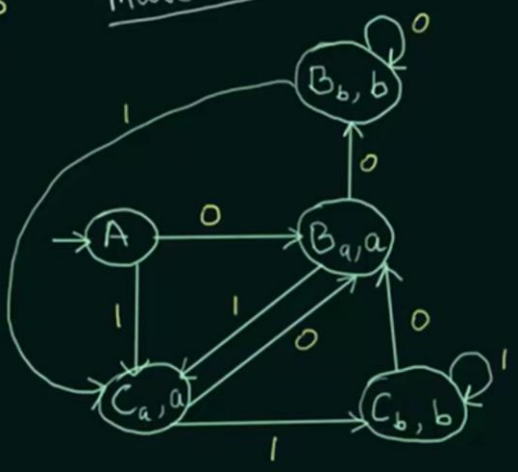
Convert the following Mealy Machine to its equivalent Moore Machine

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

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

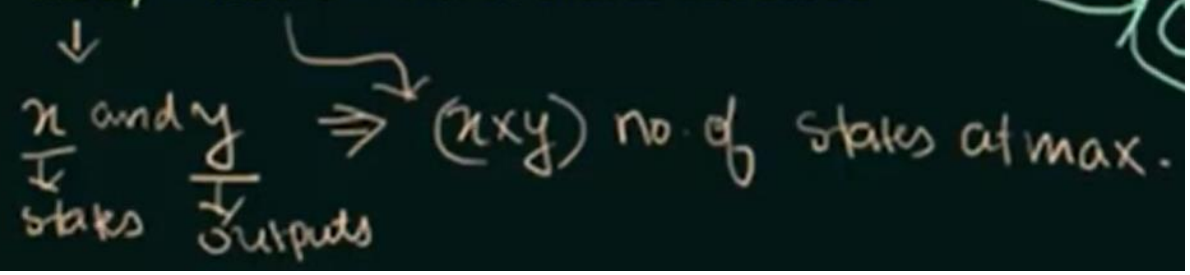


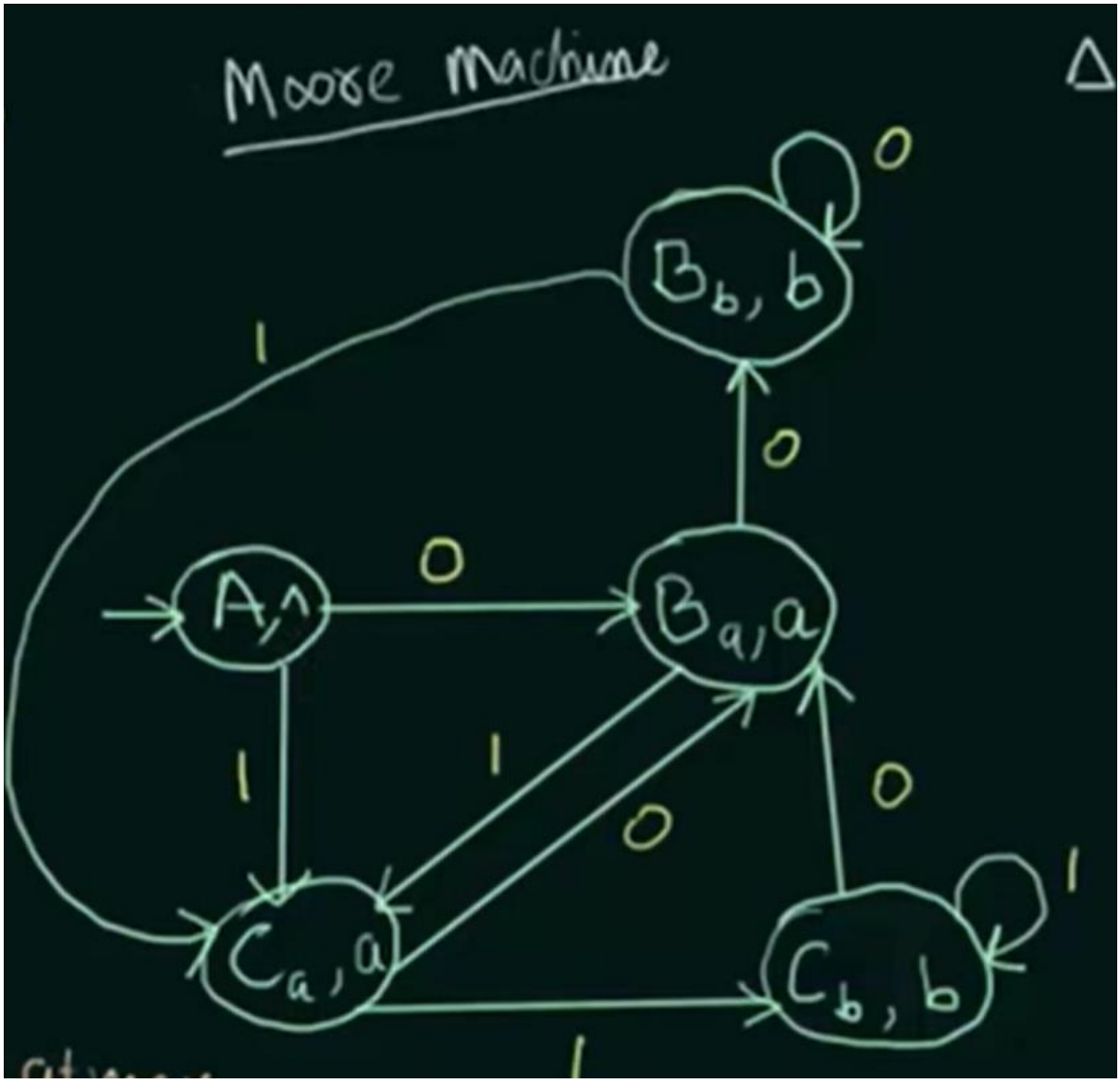
Moore machine



Moore -> Mealy => No. of states were same

Mealy -> Moore => No. of states increased




Moore machine
B_b, b
A, λ
B_a, a
C_a, a
C_b, b
0
1

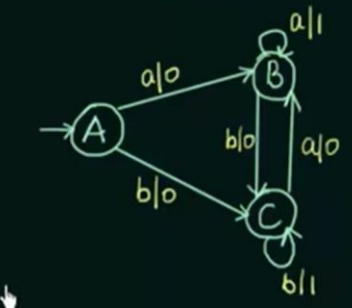
Conversion of Mealy Machine to Moore Machine - Examples (Part-1)

Given below is a Mealy Machine that prints '1' whenever the sequence 'aa' or 'bb' is encountered in any input binary string from Σ^* where $\Sigma = \{a,b\}$.

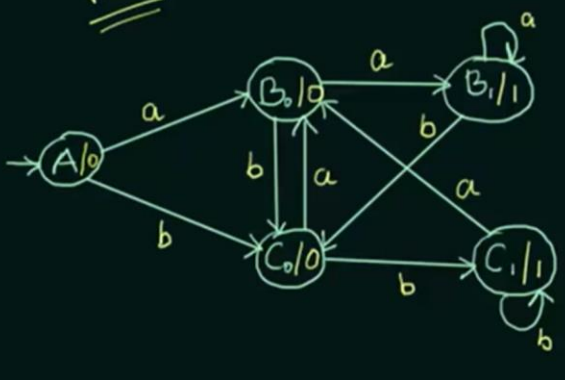
DESIGN THE EQUIVALENT MOORE MACHINE FOR IT.

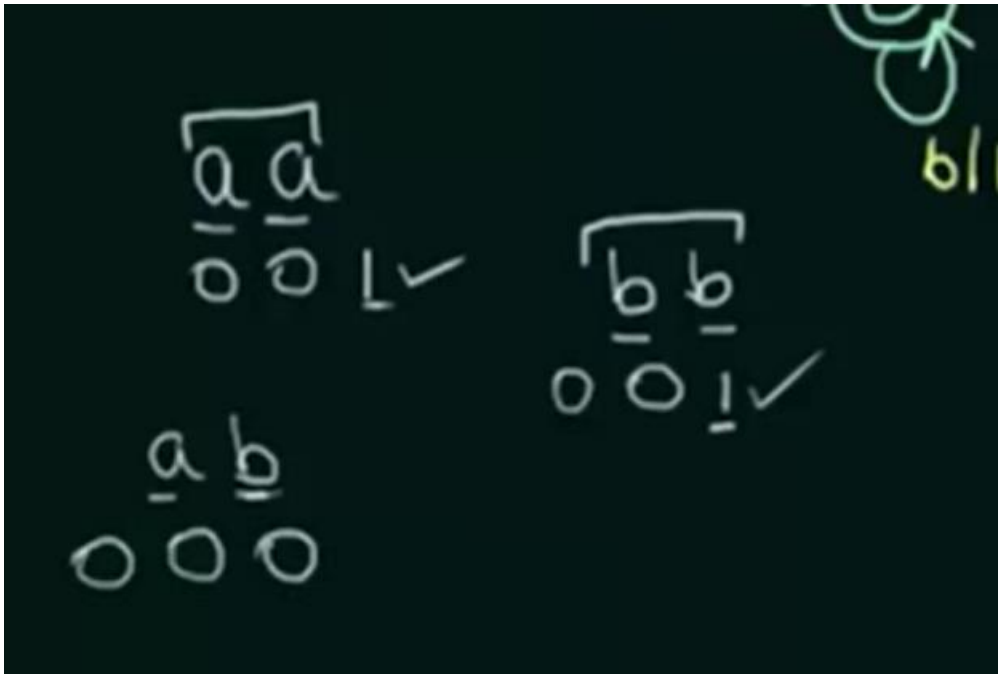
$\Sigma = \{a,b\}$

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



Moore

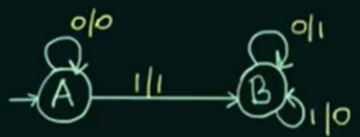


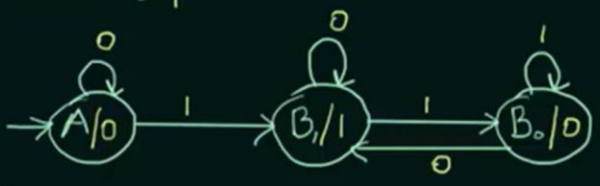


Conversion of Mealy Machine to Moore Machine- Examples (Part-2)

Convert the given Mealy Machine that give the 2's complement of any binary input to its equivalent MOORE MACHINE.

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

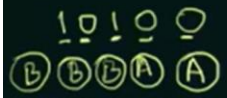




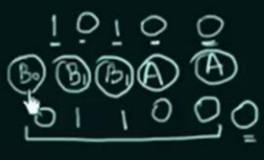
Moore

Eg. 10100
1's. 01011
+ 1
01100

Mealy:



Moore:



Conversion of Mealy Machine to Moore Machine -Examples (Part-3)

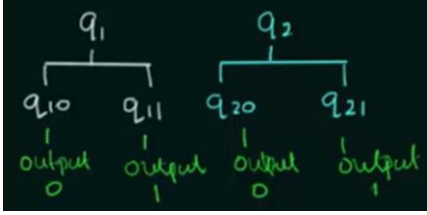
Using Transition Table

Convert the given Mealy Machine to its equivalent Moore Machine

State	a	b
→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

Moore

State	a	b	Output
→q_0	q_3	q_{11}	1
q_{10}	q_0	q_3	0
q_{11}	q_0	q_3	1
q_{20}	q_{21}	q_{20}	0
q_{21}	q_{21}	q_{20}	1
q_3	q_{10}	q_0	0



Kleene closure

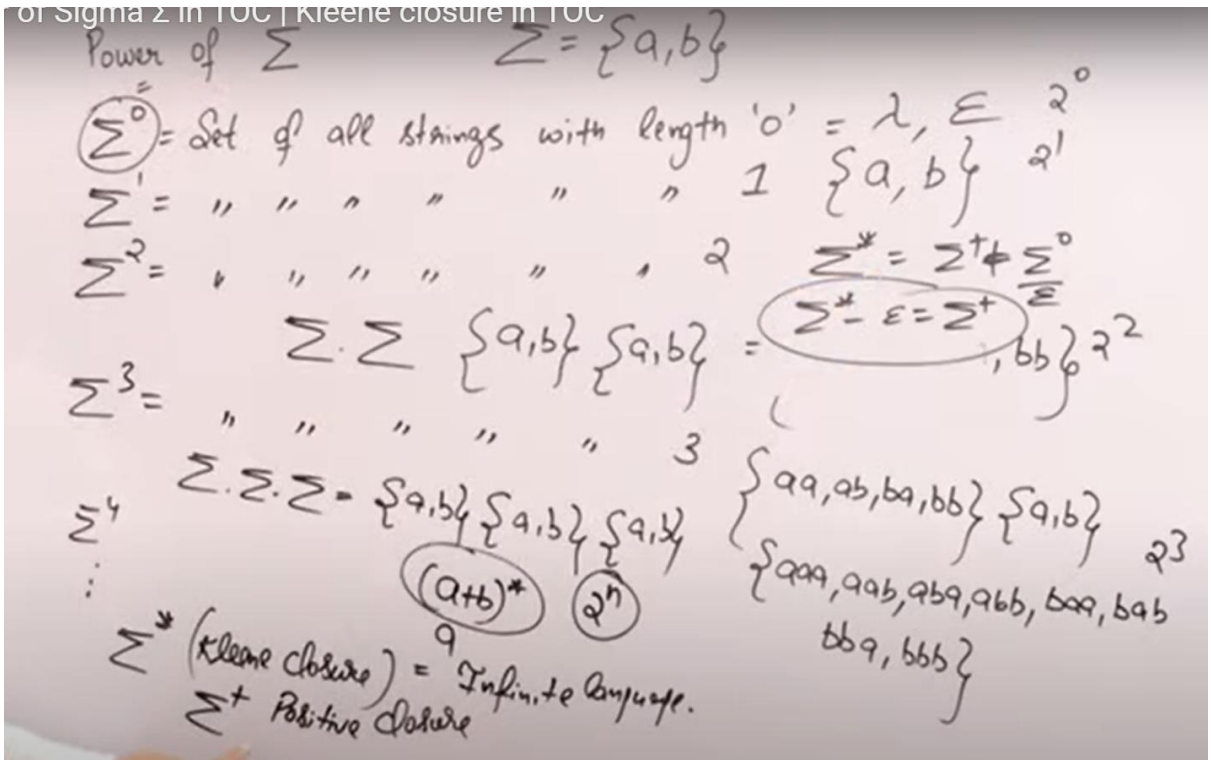
Kleene's Theorem

➢Kleene's Theorem has three parts:

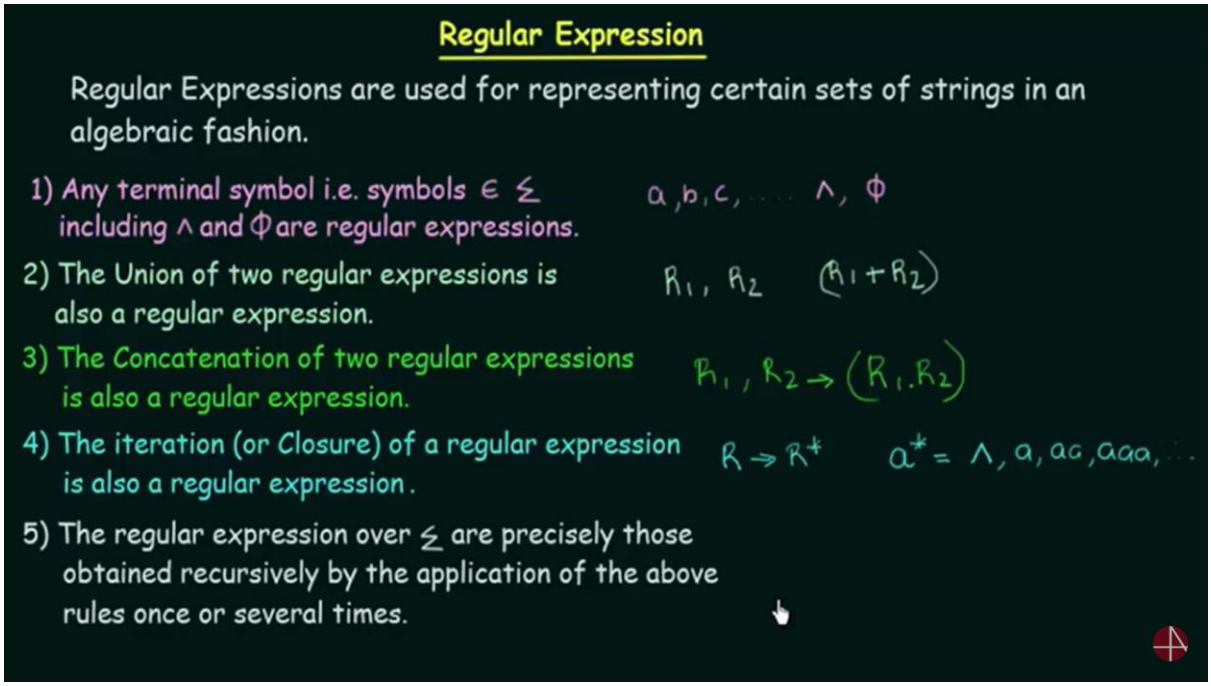
1) If language can be accepted by FA (Finite Automata) then it can also be accepted by TG (Transition Graph).

2) If language can be accepted by TG (Transition Graph) then it can also be expressed by RE (Regular Expression).

3) If language can be expressed by RE (Regular Expression) then it can be accepted by FA (Finite Automata)

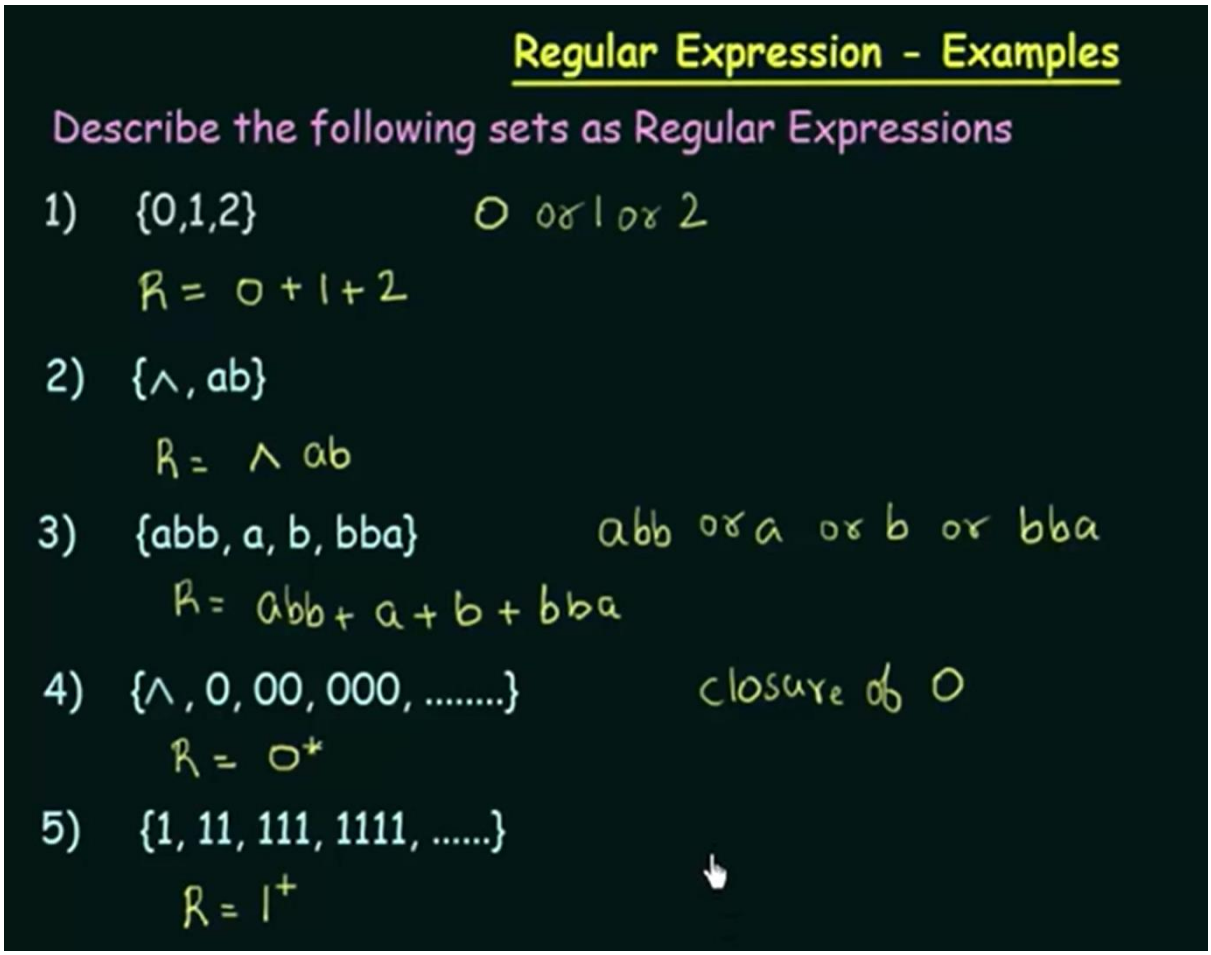


Regular Expressions

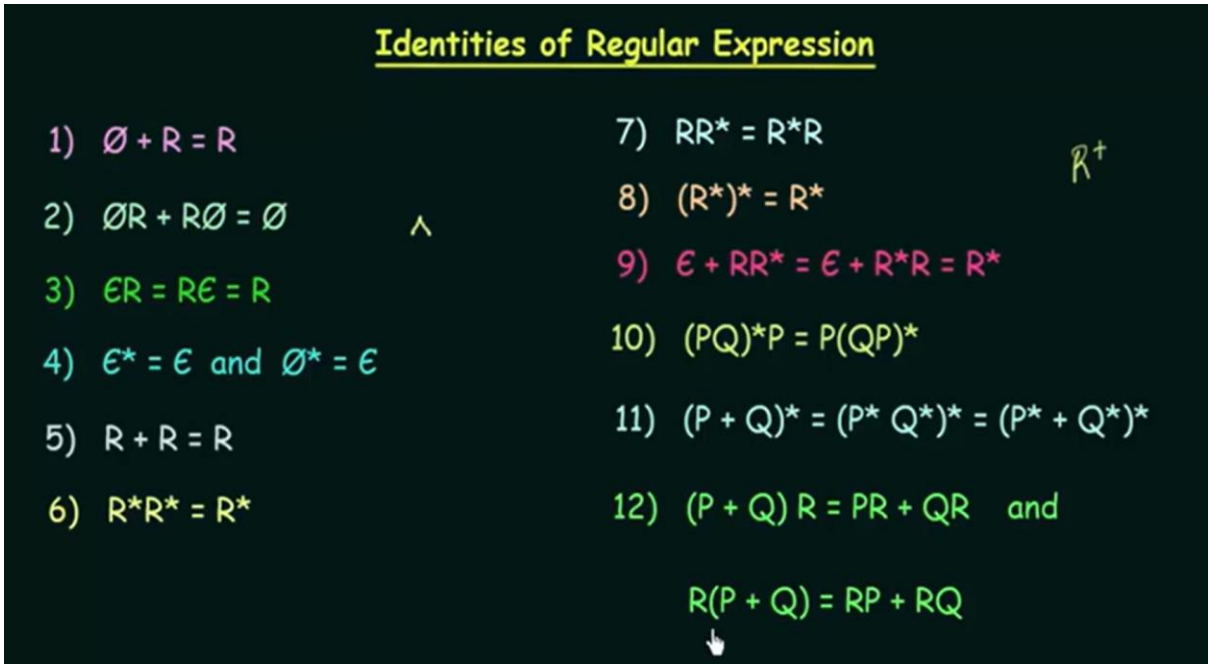


Properties of Regular Expressions:

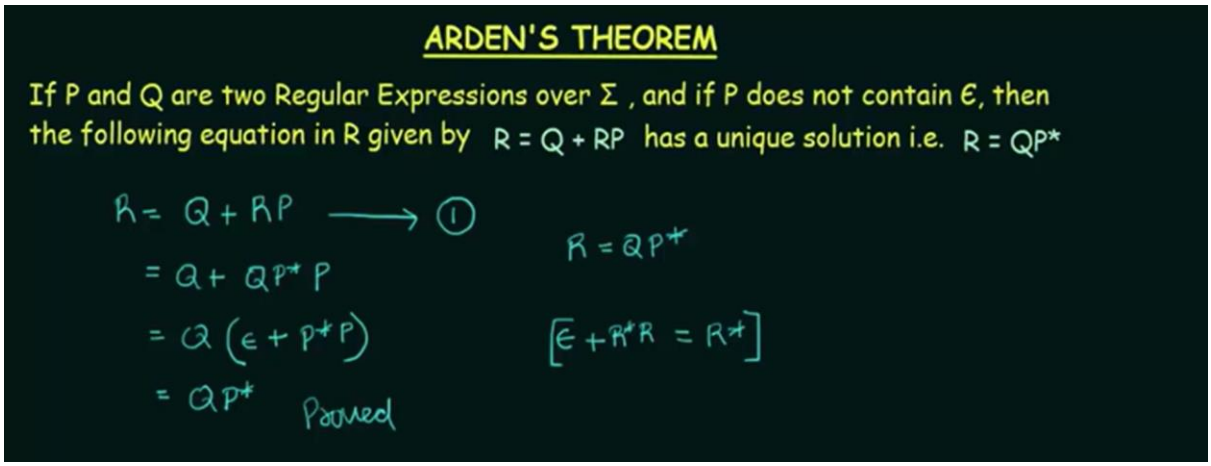
https://www.geeksforgeeks.org/properties-of-regular-expressions/



Identities of Regular Expressions(properties)



Arden's Theorem



$R = Q + RP$

$= Q + [Q + RP]P$

$= Q + QP + RP^2$

$= Q + QP + [Q + RP]P^2$

$= Q + QP + QP^2 + RP^3$

$\vdots$

$= Q + QP + QP^2 + \ldots QP^n + RP^{n+1}$

$[R = QP^*]$

$= Q + QP + QP^2 + \ldots QP^n + QP^* P^{n+1}$

$= Q[\epsilon + P + P^2 + \ldots P^n + P^* P^{n+1}$

$R = QP^*$

An Example Proof using Identities of Regular Expressions

Prove that (1+00*1) + (1+00*1) (0+10*1)* (0+10*1) is equal to 0*1 (0+10*1)*

$LHS = (1+00^*1) + (1+00^*1)(0+10^*1)^*(0+10^*1)$

$= (1+00^*1)[\epsilon + (0+10^*1)^*(0+10^*1)]$ $\quad \epsilon + R^*R = R^*$

$= (1+00^*1)(0+10^*1)^*$

$= (\epsilon . 1+00^*1)(0+10^*1)^*$ $\quad \epsilon . R = R$

$= (\epsilon+00^*)1(0+10^*1)^*$

$= 0^*1(0+10^*1)^*$

Designing Regular Expressions - Examples (Part-1)

Design Regular Expression for the following languages over {a,b}

1) Language accepting strings of length exactly 2
2) Language accepting strings of length atleast 2
3) Language accepting strings of length atmost 2

Soln

1) $L_1 = \{aa, ab, ba, bb\}$

$R = aa + ab + ba + bb$

$= a(a+b) + b(a+b)$

$= (a+b)(a+b)$

2) $L_1 = \{aa, ab, ba, bb, aaa, \ldots\}$

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

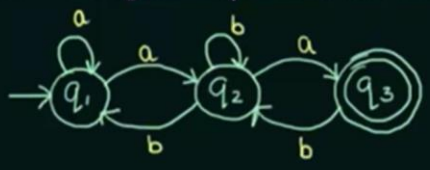
3) $L_1 = \{\epsilon, a, b, aa, ab, ba, bb\}$

$R = \epsilon + a + b + aa + ab + ba + bb$

$= (\epsilon + a + b)(\epsilon + a + b)$

Designing Regular Expression - Examples (Part-2)

Find the Regular Expression for the following NFA



$q_3 = q_2 a \rightarrow (1)$

$q_2 = q_1 a + q_2 b + q_3 b \rightarrow (2)$

$q_1 = \epsilon + q_1 a + q_2 b \rightarrow (3)$

(1) $\rightarrow$ $q_3 = q_2 a$

$= (q_1 a + q_2 b + q_3 b) a$

$= q_1 aa + q_2 ba + q_3 ba \rightarrow (4)$

(2) $\rightarrow$ $q_2 = q_1 a + q_2 b + q_3 b$ Putting value of q_3 from (1)

$= q_1 a + q_2 b + (q_2 a) b$

$= q_1 a + q_2 b + q_2 ab$

$q_2 = q_1 a + q_2 (b + ab)$

$q_2 = R \quad Q \quad R \quad P$

$R = Q + RP$ Arden's Theorem

$R = QP^*$

$q_2 = (q_1 a)(b + ab)^* \rightarrow (5)$

(3) ⇒ $q_1 = \epsilon + q_1a + q_2b$

Putting value of q_2 from (5)

$q_1 = \epsilon + q_1a + ((q_1a)(b+ab)^*)b$

$R = Q + RP$

$R = QP^*$

$q_1 = \epsilon + q_1(a + a(b+ab)^*)b$

(R = Q, R = R, P = $a + a(b+ab)^*$)

$\epsilon \cdot R = R$

$q_1 = \epsilon((a+a(b+ab)^*)b)^*$

$q_1 = (a+a(b+ab)^*b)^*$ → (6)

Final state (q_3)

$q_3 = q_2a$

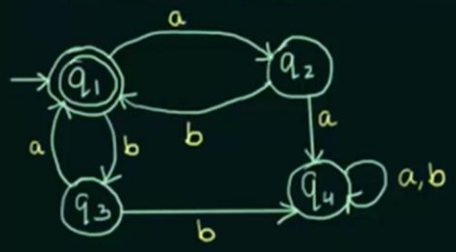
$= q_1a(b+ab)^*a$ Putting value of q_2 from (5)

$q_3 = (a+a(b+ab)^*b)^*\ a(b+ab)^*a$ Putting value of q_1 from (6)

= Required Regular Expression for the given NFA

Designing Regular Expression - Examples (Part-3)

Find the Regular Expression for the following DFA



$q_1 = \epsilon + q_2b + q_3a$ → (I)

$q_2 = q_1a$ → (II)

$q_3 = q_1b$ → (III)

$q_4 = q_2a + q_3b + q_4a + q_4b$ → (IV)

(I) $\rightarrow$ $q_1 = \epsilon + q_2 b + q_3 a$

Putting values of q_2 and q_3 from (II) and (III)

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

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

R Q R P

$R = Q + RP$

$R = QP^*$ Arden's Theorem

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

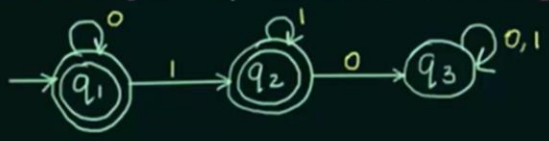
$\epsilon R = R$

$q_1 = (ab+ba)^*$

Designing Regular Expression - Examples (Part-4)

(When there are Multiple Final States)

Find the Regular Expression for the following DFA



$q_1 = \epsilon + q_1 0 \rightarrow$ (I)

$q_2 = q_1 1 + q_2 1 \rightarrow$ (II)

$q_3 = q_2 0 + q_3 0 + q_3 1 \rightarrow$ (III)

Final State q_1

(I) $\rightarrow$ $q_1 = \epsilon + q_1 0$

R Q R P

$R = Q + RP$

$R = QP^*$ Arden's Theorem

$q_1 = \epsilon . 0^*$

$\epsilon R = R$

$q_1 = 0^*$

$q_1 = 0^* \rightarrow$ (IV)

Final State q_2

(II) $\rightarrow$ $q_2 = q_1 1 + q_2 1$

$q_2 = 0^* 1 + q_2 1$ Putting value of q_1 from (IV)

R Q R P

$R = Q + RP$

$R = QP^*$

$q_2 = 0^* 1 (1)^*$

R = union of both Final states

$= 0^* + 0^* 1 1^*$

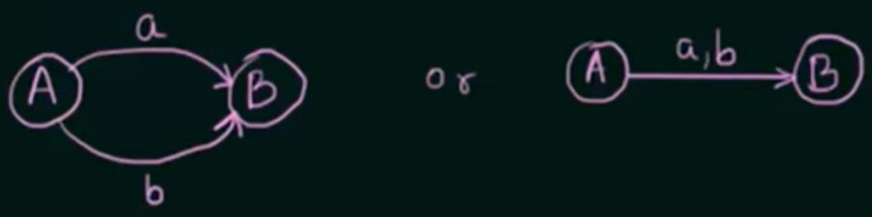
$= 0^* (\epsilon + 1 1^*)$ $\quad \epsilon + RR^* = R^*$

$= 0^* 1^*$

↳ Regular Expression

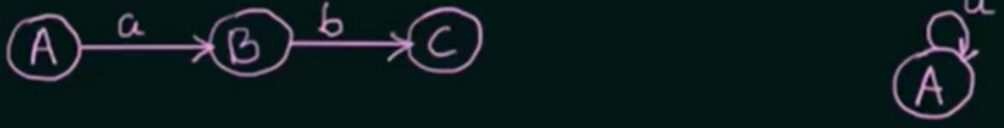
Conversion of Regular Expression to Finite Automata

$(a+b)$



$(a.b)$

a^*

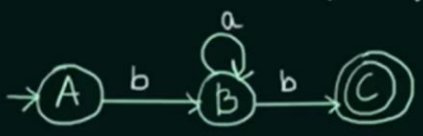


Conversion of Regular Expression to Finite Automata - Examples (Part-1)

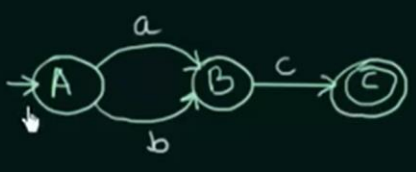
Convert the following Regular Expressions to their equivalent Finite Automata:

1) b a* b
2) (a+b) c
3) a (bc)*

1) ba^*b — bb, bab, baab, ...

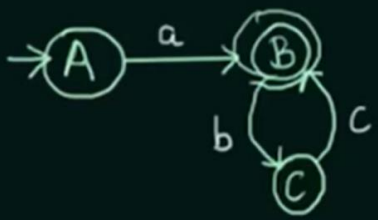


2) $(a+b)c$



ac
bc

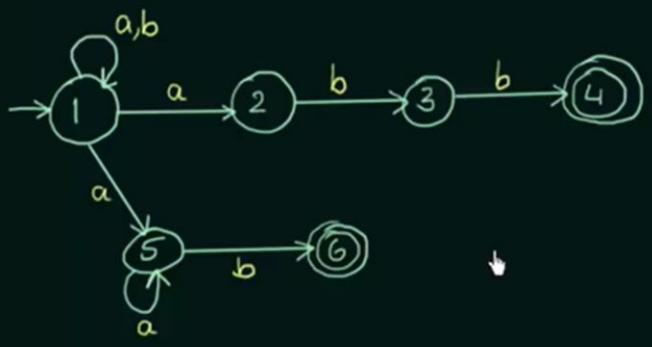
3) $a(bc)^*$ — a, abc, abcbc, abcbcbc



Conversion of Regular Expression to Finite Automata - Examples (Part-2)

Convert the following Regular Expression to its equivalent Finite Automata:

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



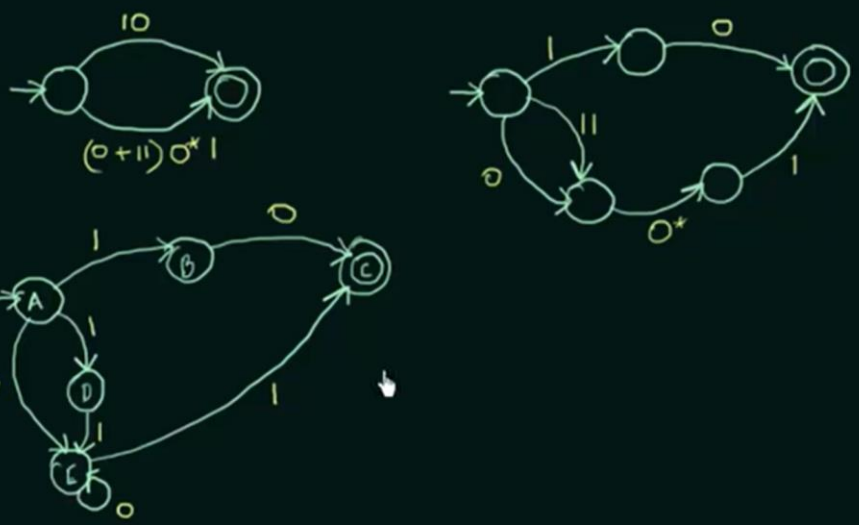
$a^+ = \{a, aa, aaa, \ldots\}$

$a^* = \{\epsilon, a, aa, \ldots\}$

Conversion of Regular Expression to Finite Automata - Examples (Part-3)

Convert the following Regular Expression to its equivalent Finite Automata:

10 + (0 + 11) 0*1

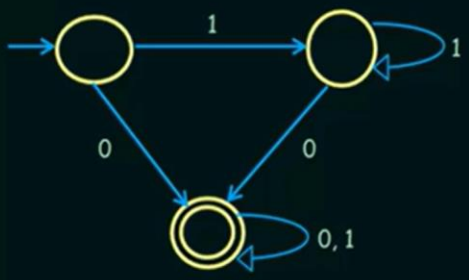


Regular Languages and Finite Automata

(Solved Problem - 1)

GATE 2013

Consider the DFA A given below:

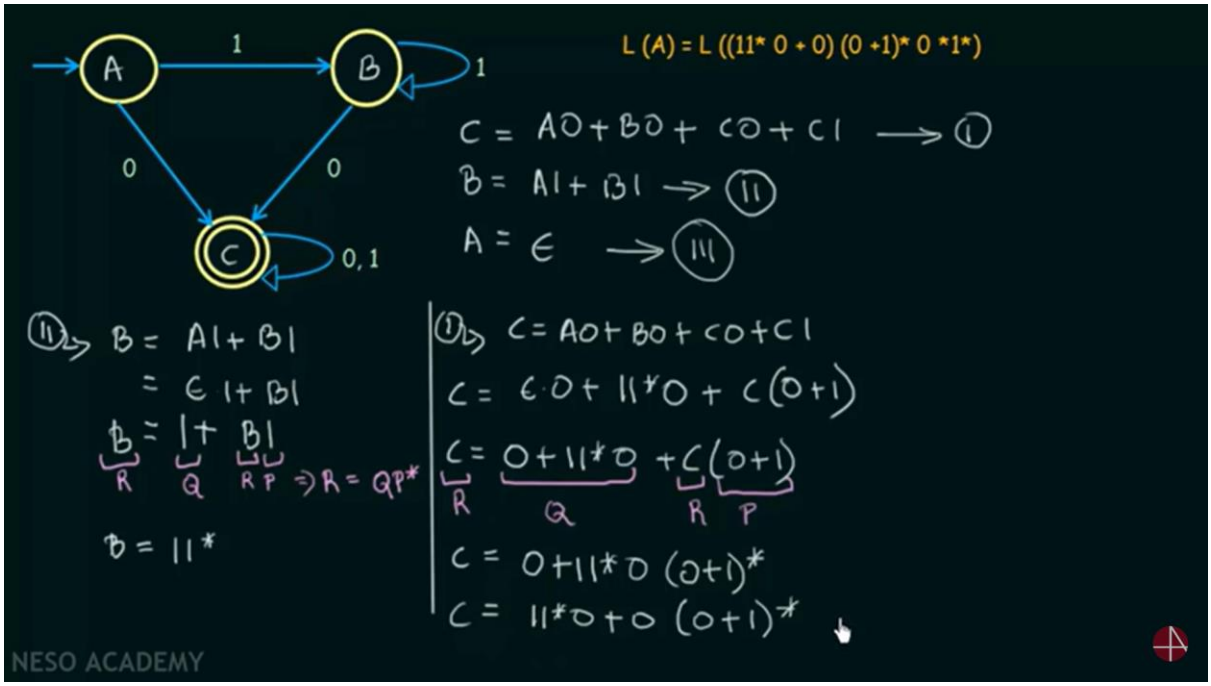


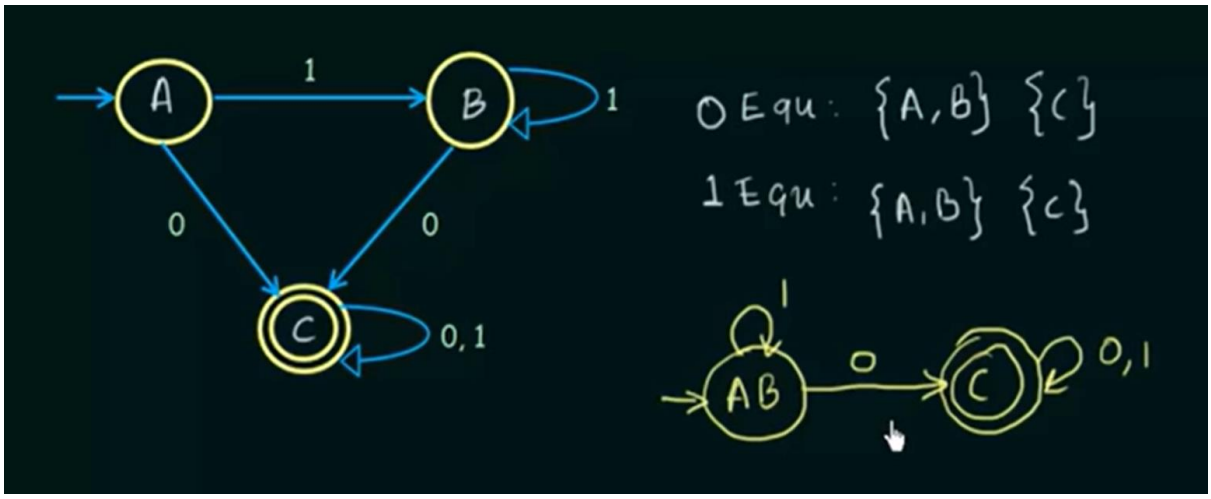
Which of the following are FALSE?

1. Complement of L (A) is context-free
2. L (A) = L ((11* 0 + 0) (0 +1)* 0 *1*)
3. For the language accepted by A, A is the minimal DFA
4. A accepts all strings over {0, 1} of length at least 2

(A) 1 and 3 only
(B) 2 and 4 only
(C) 2 and 3 only
(D) 3 and 4 only

NESO ACADEMY


A
B
C
1
1
0
0
0, 1
L (A) = L ((11* 0 + 0) (0 +1)* 0 *1*)
C = A0 + B0 + C0 + C1 → (I)
B = A1 + B1 → (II)
A = ∈ → (III)
(II) → B = A1 + B1
= ∈ 1 + B1
B = 1 + B1
R Q R P ⇒ R = QP*
B = 11*
(I) → C = A0 + B0 + C0 + C1
C = ∈0 + 11*0 + C(0+1)
C = 0 + 11*0 + C(0+1)
R Q R P
C = 0 + 11*0 (0+1)*
C = 11*0 + 0 (0+1)*
NESO ACADEMY

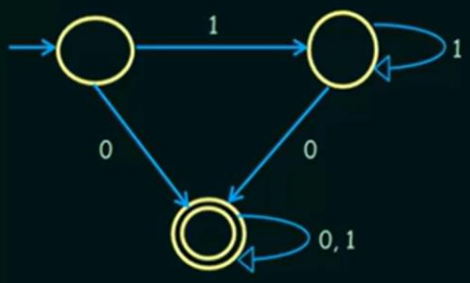

A
B
C
1
1
0
0
0, 1
0 Equ: {A, B} {C}
1 Equ: {A, B} {C}
AB
C
1
0
0, 1

Regular Languages and Finite Automata

(Solved Problem - 1)

GATE 2013

Consider the DFA A given below:



Which of the following are FALSE?

1. Complement of L (A) is context-free ✓

2. L (A) = L ((11* 0 + 0) (0 +1)* 0 *1*) ✓

3. For the language accepted by A, A is the minimal DFA ✗

4. A accepts all strings over {0, 1} of length at least 2 ✗

(A) 1 and 3 only (B) 2 and 4 only
(C) 2 and 3 only (D) 3 and 4 only

NESO ACADEMY

Regular Languages and Finite Automata

(Solved Problem - 2)

GATE 2013

Consider the languages $L_1 = \Phi$ and $L_2 = \{a\}$. Which one of the following represents $L_1L_2^* \cup L_1^*$?

(A) $\{\epsilon\}$ ✓
(B) Φ
(C) a^*
(D) $\{\epsilon, a\}$

$L_1 L_2^* \cup L_1^*$

$\Phi . a^* \cup L_1^*$

$\Phi \cup \Phi^*$

$\Phi \cup \epsilon$

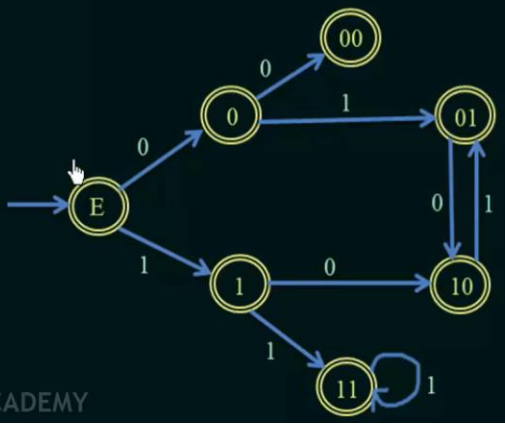
ϵ

Regular Languages and Finite Automata

(Solved Problem - 3)

GATE 2012

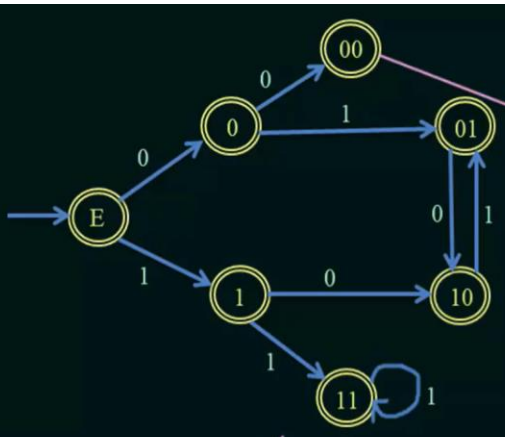
Consider the set of strings on {0, 1} in which, every substring of 3 symbols has at most two zeros. For example, 001110 and 011001 are in the language, but 100010 is not. All strings of length less than 3 are also in the language. A partially completed DFA that accepts this language is shown below.



The missing arcs in the DFA are:

(A) A
(B) B
(C) C
(D) D

NESO ACADEMY



The missing arcs in the DFA are:

(A) A
(B) B
(C) C
(D) D

(A)

	00	01	10	11	q
00	1	0			
01				1	
10	0				
11			0		

(B)

	00	01	10	11	q
00		0			1
01		1			
10				0	
11		0			

(C)

	00	01	10	11	q
00		1			0
01		1			
10			0		
11		0			

(D)

	00	01	10	11	q
00		1			0
01				1	
10	0				
11			0		

NESO ACADEMY

Regular Languages and Finite Automata

(Solved Problem - 4)

GATE 2009

Which one of the following languages over the alphabet {0, 1} is described by the regular expression: **(0+1)* 0 (0+1)* 0 (0+1)*** ?

(A) The set of all strings containing the substring 00. ✗

(B) The set of all strings containing at most two 0's. ✗

(C) The set of all strings containing at least two 0's. ✓

(D) The set of all strings that begin and end with either 0 or 1. ✗

Atmost - Maximum - Not more than 2 zeros

0000 - 4

Atleast - Minimum - Not less than 2

0 - - - 0

1 - - - 1

00

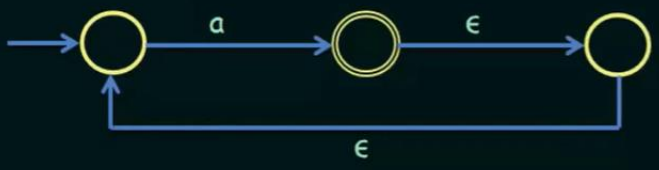
NESO ACADEMY

Regular Languages and Finite Automata

(Solved Problem - 5)

What is the complement of the language accepted by the NFA shown below?

GATE 2012



(A) Φ

(B) {ϵ} ✓

(C) a*

(D) {a , ϵ }

a ✓

a a ✓

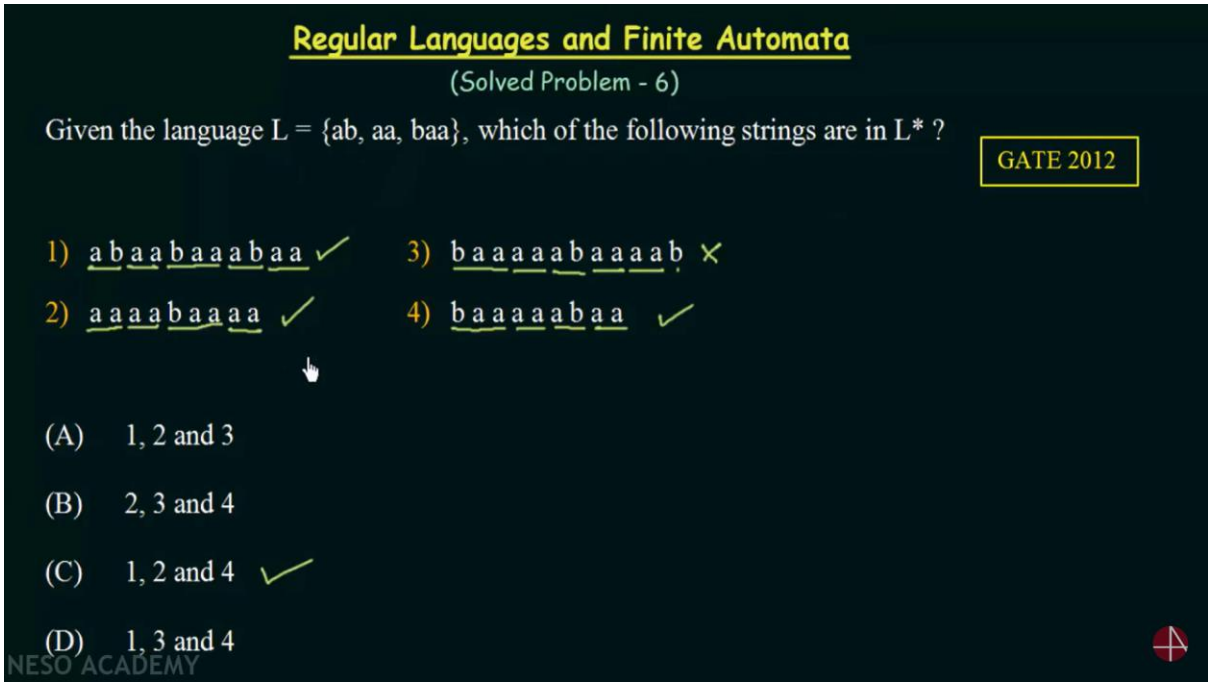
a a a ✓

a a a a ✓

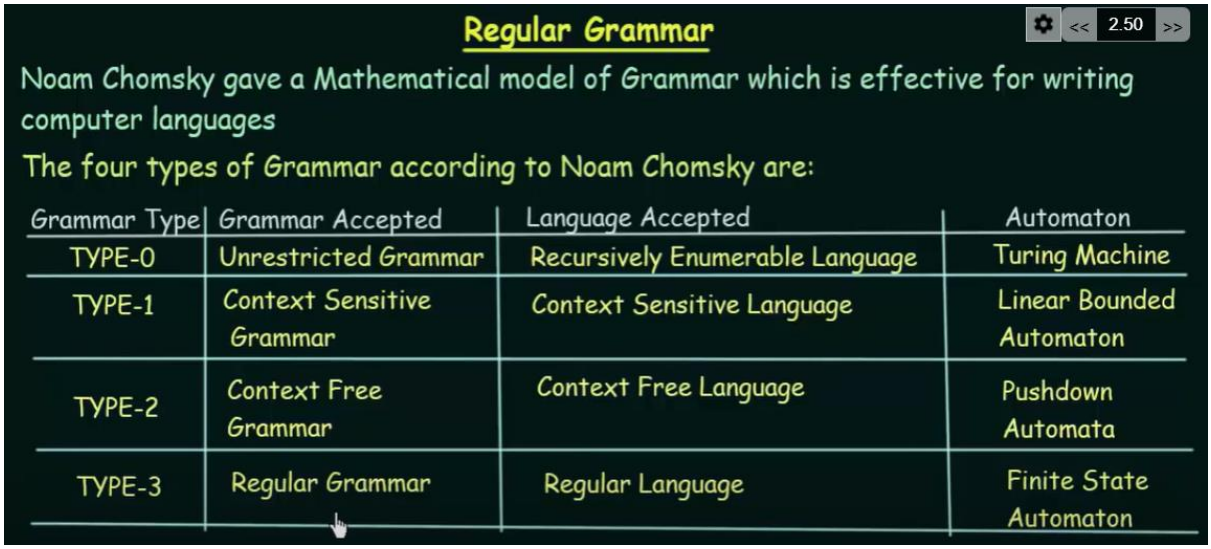
a^+ ✓

a^* ✗

ϵ



Regular Grammar



Grammar:

A Grammar 'G' can be formally described using 4 tuples as $G = (V,T,S,P)$ where,

- V = Set of Variables or Non-Terminal Symbols
- T = Set of Terminal Symbols
- S = Start Symbol
- P = Production rules for Terminals and Non-Terminals

  A production rules has the form $\alpha \longrightarrow \beta$ where α and β are strings on V U T and atleast one symbol of α belongs to V.

Example: $G = (\{S,A,B\}, \{a,b\}, S, \{S \rightarrow AB,\ A \rightarrow a,\ B \rightarrow b\})$

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

$T = \{a, b\}$

$S = S$

$P = S \rightarrow AB,\ A \rightarrow a,\ B \rightarrow b$

Eg: $S \rightarrow AB$
$\rightarrow aB$
$\rightarrow ab$

Regular Grammar:

Regular Grammar can be divided into two types:

Right Linear Grammar	Left Linear Grammar
A grammar is said to be Right Linear if all productions are of the form $A \rightarrow xB$ $A \rightarrow x$ where $A,B \in V$ and $x \in T$	A grammar is said to be Left Linear if all productions are of the form $A \rightarrow Bx$ $A \rightarrow x$ where $A,B \in V$ and $x \in T$

Eg: $S \rightarrow abS \mid b$ - Right linear

$S \rightarrow Sbb \mid b$ - Left linear

Derivations from a Grammar

The set of all strings that can be derived from a Grammar is said to be the LANGUAGE generated from that Grammar

Example 1: Consider the Grammar $G1 = (\{S,A\}, \{a,b\}, S, \{S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow \epsilon\})$

$S \rightarrow aAb$ [by $S \rightarrow aAb$]
$\rightarrow aaAbb$ [by $aA \rightarrow aaAb$]
$\rightarrow aaaAbbb$ [by $aA \rightarrow aaAb$]
$\rightarrow aaabbb$ [by $A \rightarrow \epsilon$]

Example 2: $G2 = (\{S,A,B\}, \{a,b\}, S, \{S \rightarrow AB, A \rightarrow a, B \rightarrow b\})$

$S \rightarrow AB$
$\rightarrow ab$

$L(G2) = \{ab\}$

Example 3: $G3 = (\{S,A,B\}, \{a,b\}, S, \{S \rightarrow AB, A \rightarrow aA|a, B \rightarrow bB|b\})$

$S \rightarrow AB$
$\rightarrow ab$

$S \rightarrow AB$
$\rightarrow aAbB$
$\rightarrow aabb$

$S \rightarrow AB$
$\rightarrow aAb$
$\rightarrow aaab$

$S \rightarrow AB$
$\rightarrow abB$
$\rightarrow abb$

$L(G3) = \{ab, a^2b^2, a^2b, ab^2, \ldots\}$

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

Context Free Language

Context Free Language

In formal language theory, a Context Free Language is a language generated by some Context Free Grammar.

The set of all CFL is identical to the set of languages accepted by Pushdown Automata.

Context Free Grammar is defined by 4 tuples as $G = \{ V, \Sigma, S, P \}$ where

- V = Set of Variables or Non-Terminal Symbols
- Σ = Set of Terminal Symbols
- S = Start Symbol
- P = Production Rule

Context Free Grammar has Production Rule of the form

$A \rightarrow a$

where, $a = \{V \cup \Sigma\}^*$ and $A \in V$

Example: For generating a language that generates equal number of a's and b's in the form $a^n b^n$, the Context Free Grammar wil be defined as

$G = \{ (S,A), (a,b), (S \rightarrow aAb | A \rightarrow aAb | \in) \}$

$S \rightarrow aAb$

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

$\rightarrow aaaAbbb$ (")

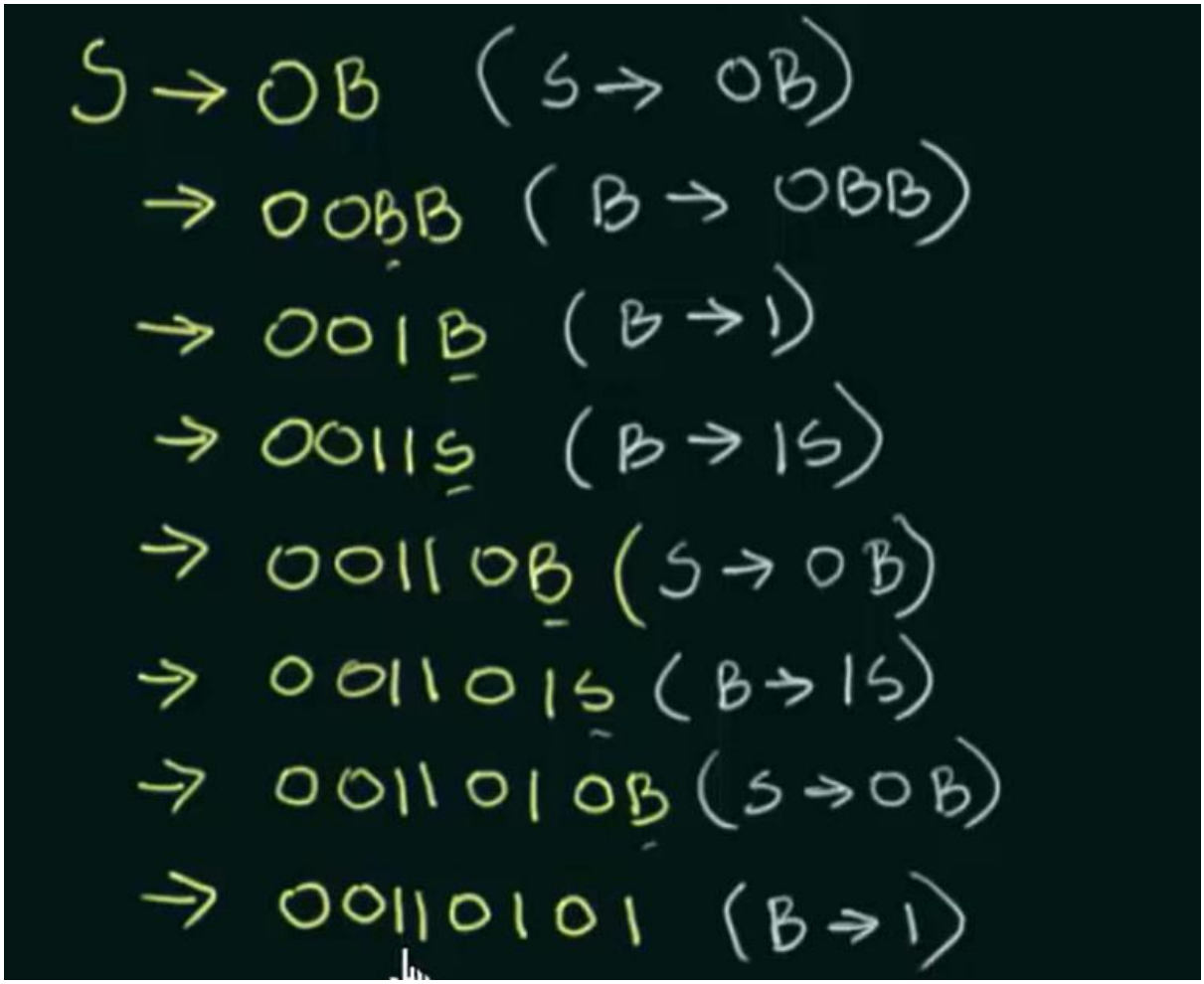
$\rightarrow aaabbb$ (by $A \rightarrow \in$)

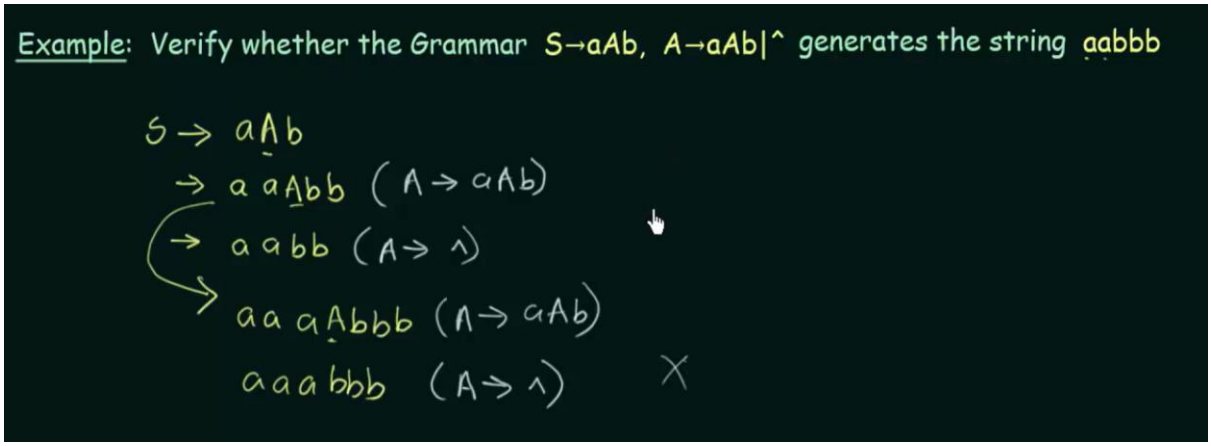
$\rightarrow a^3 b^3 \Rightarrow a^n b^n$

Method to find whether a String belongs to a Grammar or not

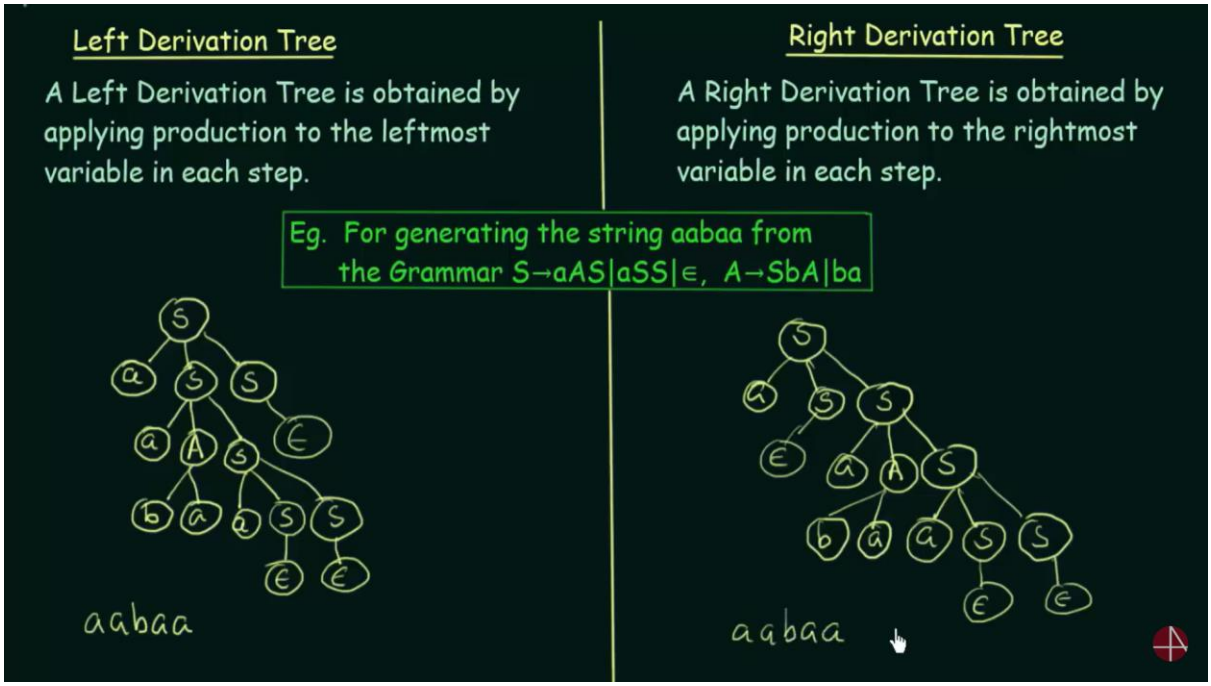
1) Start with the Start Symbol and choose the closest production that matches to the given string.
2) Replace the Variables with its most appropriate production. Repeat the process until the string is generated or until no other productions are left.

Example: Verify whether the Grammar $S \rightarrow 0B|1A$, $A \rightarrow 0|0S|1AA|\hat{}$, $B \rightarrow 1|1S|0BB$ generates the string 00110101

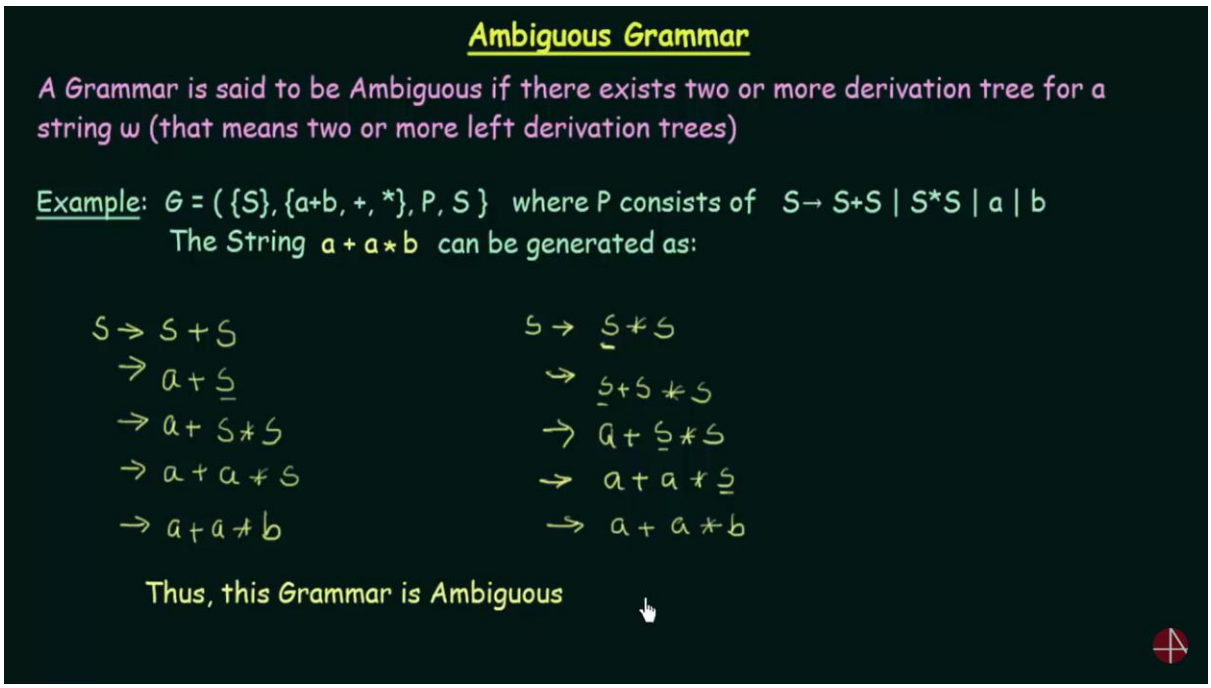




Derivation Tree



Ambiguous Grammar



Reduction of Context Free Grammar

Simplification of Context Free Grammar

Reduction of CFG

In CFG, sometimes all the production rules and symbols are not needed for the derivation of strings. Besides this, there may also be some NULL Productions and UNIT Productions. Elimination of these productions and symbols is called Simplification of CFG.

Simplification consists of the following steps:

1) Reduction of CFG 2) Removal of Unit Productions 3) Removal of Null Productions

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

Derivation Procedure:

Step 1: Include all Symbols W_1 ,that derives some terminal and initialize i = 1

Step 2: Include symbols W_{i+1} , that derives W_i

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

Step 4: Include all production rules that have W_i in it

Phase 2: Derivation of an equivalent grammar G'' , from the CFG, G', such that each symbol appears in a sentential form

Derivation Procudure:

Step 1: Include the Start Symbol in Y_1 and initialize i =1

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

Step 3: Increment i and repeat Step 2, until $Y_{i+1} = Y_i$

Example: Find a reduced grammar equivalent to the grammar G, having production rules
P: S→AC|B, A→a, C→c|BC, E→aA|e

Phase 1

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

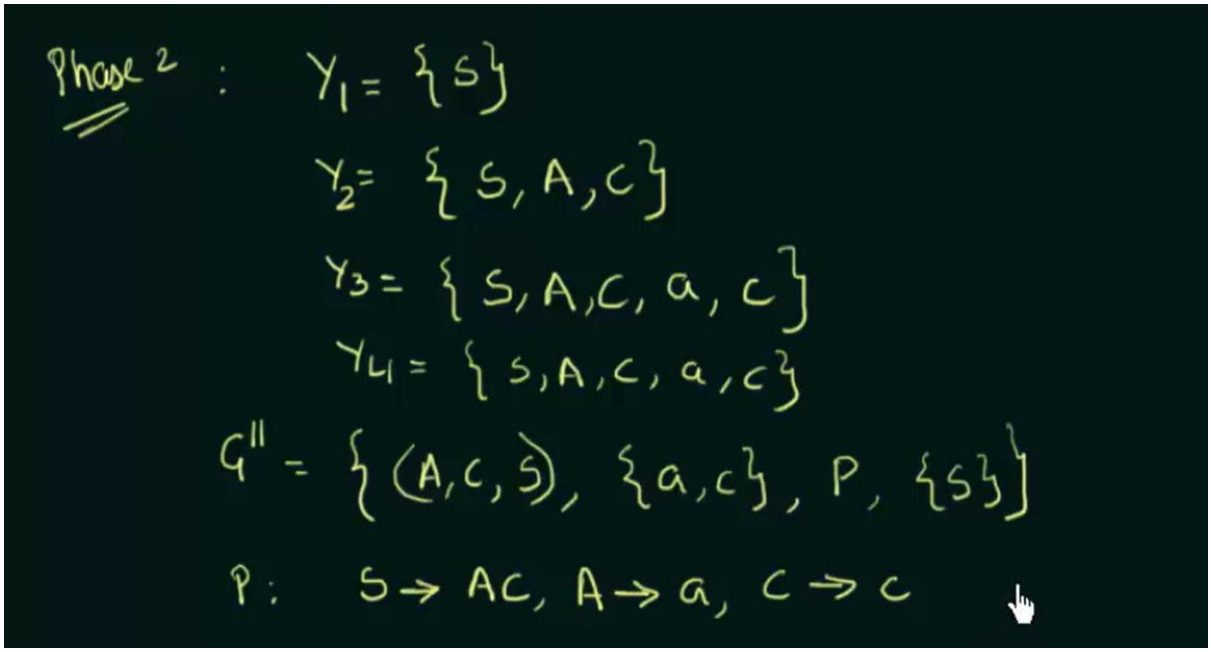
$W_1 = \{A, C, E\}$

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

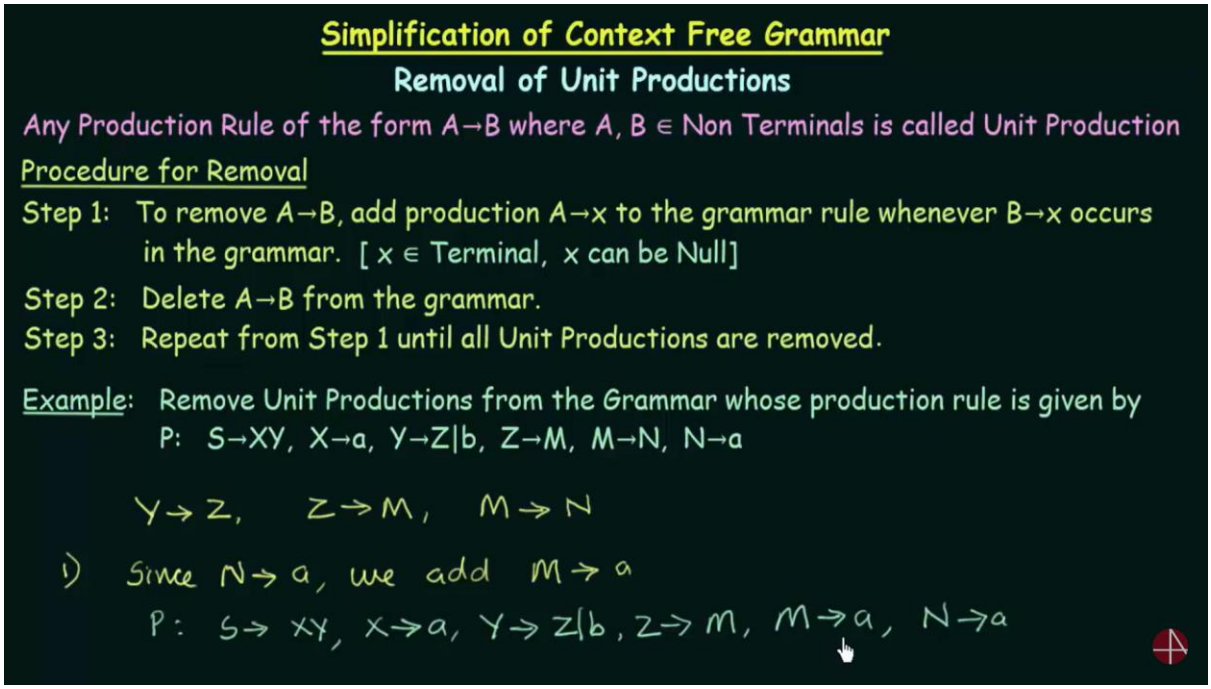
$W_3 = \{A, C, E, S\}$

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

P: $S \to AC,\ A \to a,\ C \to c,\ E \to aA|e$



Removing UNIT Productions



2) Since $M \to a$, we add $Z \to a$

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

3) Since $Z \to a$, we add $Y \to a$

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

Remove the Unreachable symbols

P: $S \to XY,\ X \to a,\ Y \to a|b$

Removal of Null Productions

Simplification of Context Free Grammar

Removal of Null Productions

In a CFG, a Non-Terminal Symbol 'A' is a nullable variable if there is a production $A \to \epsilon$ or there is a derivation that starts at 'A' and leads to ϵ. (Like $A \to \ldots\ldots\ldots\ldots \to \epsilon$)

Procedure for Removal:

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

Step2: Replace each occurences of 'A' in each of these productions with ϵ

Step 3: Add the resultant productions to the Grammar

Example: Remove Null Productions from the following Grammar

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

Example: Remove Null Productions from the following Grammar

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

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

1) To eliminate $A \to \epsilon$

$S \to ABAC$

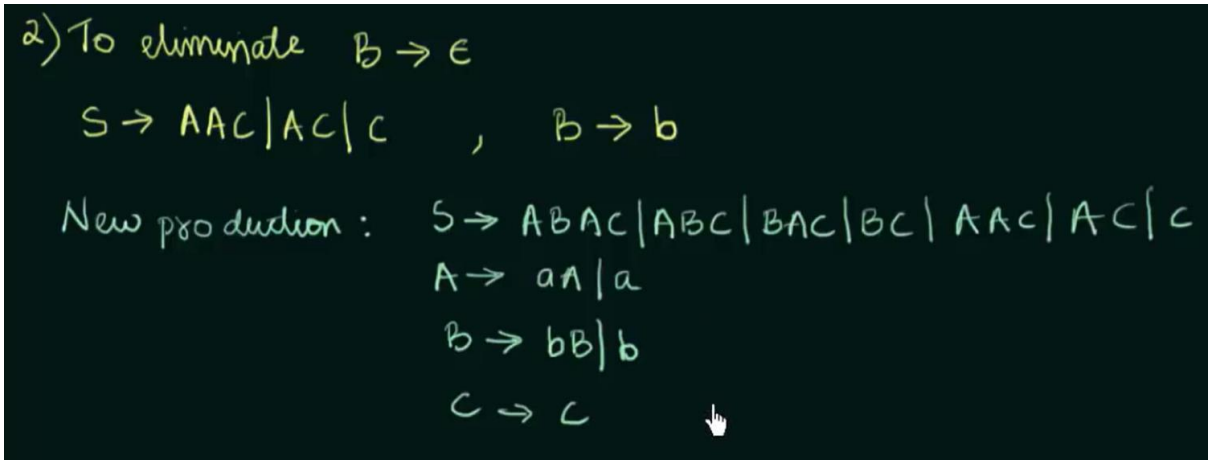
$S \to ABC|BAC|BC$

$A \to aA$

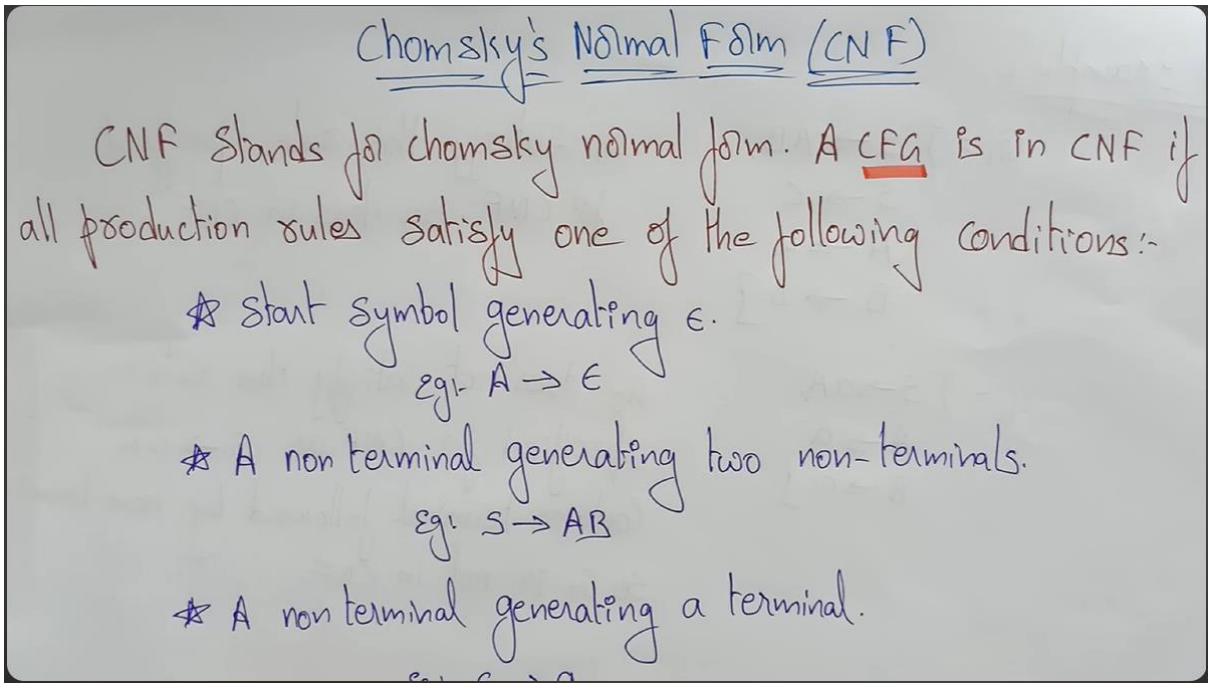
$A \to a$

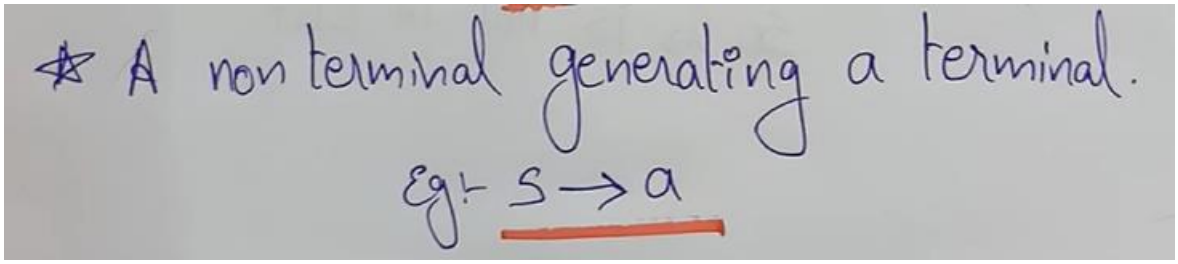
New production: $S \to ABAC|ABC|BAC|BC$

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



Chomsky Normal Form





Chomsky Normal Form

In Chomsky Normal Form (CNF) we have a restriction on the length of RHS; which is; elements in RHS should either be two variables or a Terminal.

A CFG is in Chomsky Normal Form if the productions are in the following forms:

$A \rightarrow a$

$A \rightarrow BC$

where A, B and C are non-terminals and a is a terminal

Steps to convert a given CFG to Chomsky Normal Form:

Step 1: If the Start Symbol S occurs on some right side, create a new Start Symbol S' and a new Production $S' \rightarrow S$.

Step 2: Remove Null Productions. (Using the Null Production Removal discussed in previous Lecture)

Step 3: Remove Unit Productions. (Using the Unit Production Removal discussed in previous Lecture)

Step 4: Replace each Production $A \rightarrow B_1 \ldots\ldots B_n$ where $n > 2$, with $A \rightarrow B_1 C$ where $C \rightarrow B_2 \ldots\ldots B_n$ Repeat this step for all Productions having two or more Symbols on the right side.

Step 5: If the right side of any Production is in the form $A \rightarrow aB$ where 'a' is a terminal and A and B are non-terminals, then the Production is replaced by $A \rightarrow XB$ and $X \rightarrow a$. Repeat this step for every Production which is of the form $A \rightarrow aB$

Conversion of CFG to Chomsky Normal Form

Convert the following CFG to CNF: P: $S \rightarrow ASA \mid aB,\ A \rightarrow B \mid S,\ B \rightarrow b \mid \in$

1) Since S appears in RHS, we add a new State S' and $S' \rightarrow S$ is added to the production

P: $S' \rightarrow S,\ S \rightarrow ASA \mid aB,\ A \rightarrow B \mid S,\ B \rightarrow b \mid \in$

2) Remove the Null Productions: $B \rightarrow \in$ and $A \rightarrow \in$:

After Removing $B \rightarrow \in$: P: $S' \rightarrow S,\ S \rightarrow ASA \mid aB \mid a,\ A \rightarrow B \mid S \mid \in,\ B \rightarrow b$

After Removing $A \rightarrow \in$: P: $S' \rightarrow S,\ S \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S,\ A \rightarrow B \mid S,\ B \rightarrow b$

3) Remove the Unit Productions: $S \to S$, $S' \to S$, $A \to B$ and $A \to S$:

After Removing $S \to S$: P: $S' \to S$, $S \to ASA|aB|a|AS|SA$, $A \to B|S$, $B \to b$

After Removing $S' \to S$: P: $S' \to ASA|aB|a|AS|SA$,
$S \to ASA|aB|a|AS|SA$,
$A \to B|S$, $B \to b$

After Removing $A \to B$: P: $S' \to ASA|aB|a|AS|SA$,
$S \to ASA|aB|a|AS|SA$,
$A \to b|S$, $B \to b$

After Removing $A \to S$: P: $S' \to ASA|aB|a|AS|SA$,
$S \to ASA|aB|a|AS|SA$,
$A \to b|ASA|aB|a|AS|SA$,
$B \to b$

4) Now find out the productions that has more than TWO variables in RHS

$S' \to ASA$, $S \to ASA$ and $A \to ASA$

After removing these, we get: P: $S' \to AX|aB|a|AS|SA$,
$S \to AX|aB|a|AS|SA$,
$A \to b|AX|aB|a|AS|SA$,
$B \to b$,
$X \to SA$

5) Now change the productions $S' \to aB$, $S \to aB$ and $A \to aB$

Finally we get: P: $S' \to AX|YB|a|AS|SA$,
$S \to AX|YB|a|AS|SA$,
$A \to b|AX|YB|a|AS|SA$,
$B \to b$,
$X \to SA$,
$Y \to a$

which is the required Chomsky Normal Form for the given CFG

Steps for converting CFG into CNF

Step 1 : Eliminate start symbol from the RHS.
If the start symbol T is at the right-hand side of any production, create a production as :

$$S_1 \rightarrow S$$

where S_1 is the new start symbol.

Step 2 : In the grammar, remove the null, unit and useless productions.

Step 3 : Eliminate terminals from the RHS of the production if the exists with other non-terminals or terminals.

Step 4 : Eliminate RHS with more than two non-terminals.

eg: $S \rightarrow ABS$ can be decomposed as :

CNF

$S \rightarrow RS$
$R \rightarrow AB$

$S \rightarrow ABS$

$S \rightarrow AR$
$R \rightarrow BS$

Example : Convert the given CFG to CNF,
Consider the given grammar G_1 as

$S \rightarrow a \mid aA \mid B$
$A \rightarrow aBB \mid \epsilon$
$B \rightarrow Aa \mid b$

rules CN

$S \rightarrow \epsilon$
$A \rightarrow AB$
$A \rightarrow a$

Solution :

Step 1 :

$S_1 \rightarrow S$ // new production.
$S \rightarrow a \mid aA \mid B$
$A \rightarrow aBB \mid \epsilon$
$B \rightarrow Aa \mid b$

Step 2 :- $A \to \epsilon$ null production, we have to remove ϵ.

$S_1 \to S$

$S \to a \mid aA \mid B$

$A \to aBB$

$B \to Aa \mid b \mid a$

$A \to \epsilon$

replace A with ϵ & write that remaining

Simplify CFG

Simp

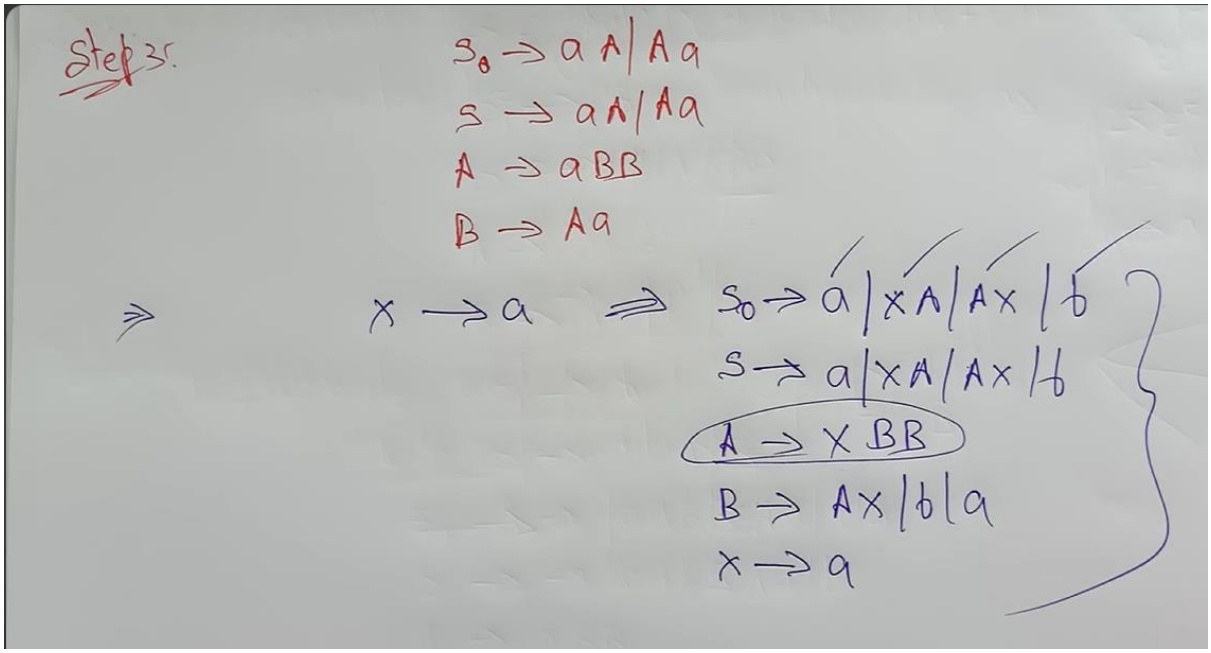
To eliminate unit production, replace B with B production rule.

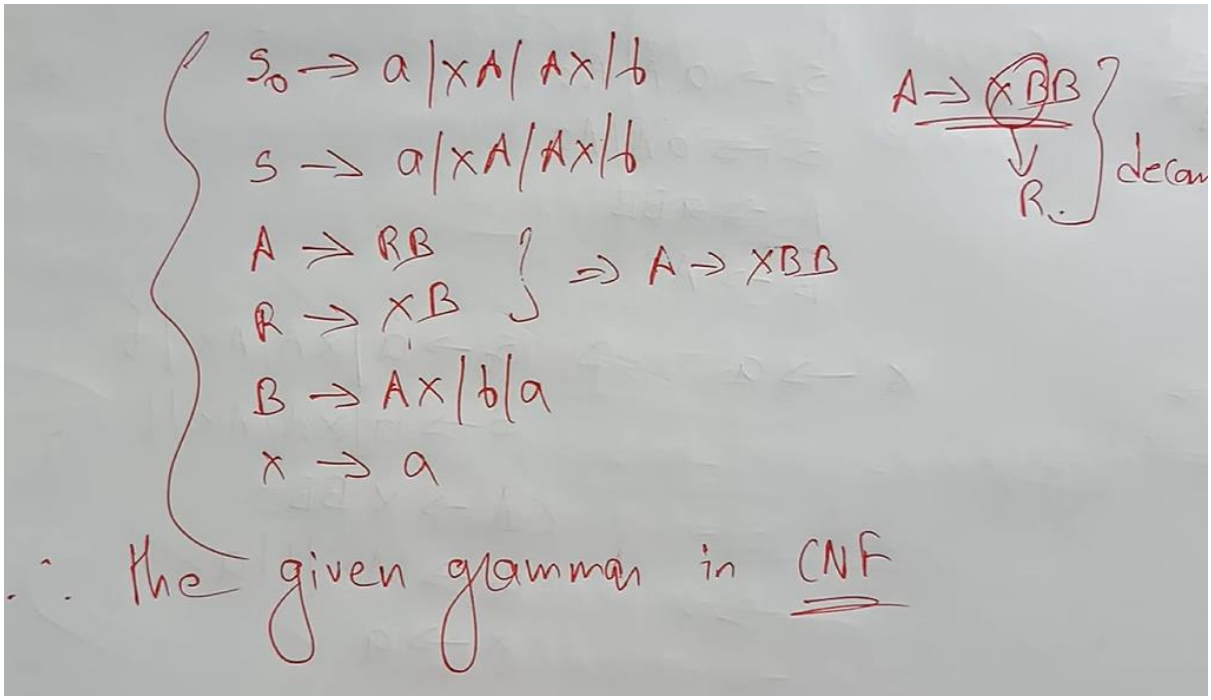
$S_1 \to a \mid aA \mid Aa \mid bb$

$S \to a \mid aA \mid Aa \mid b$

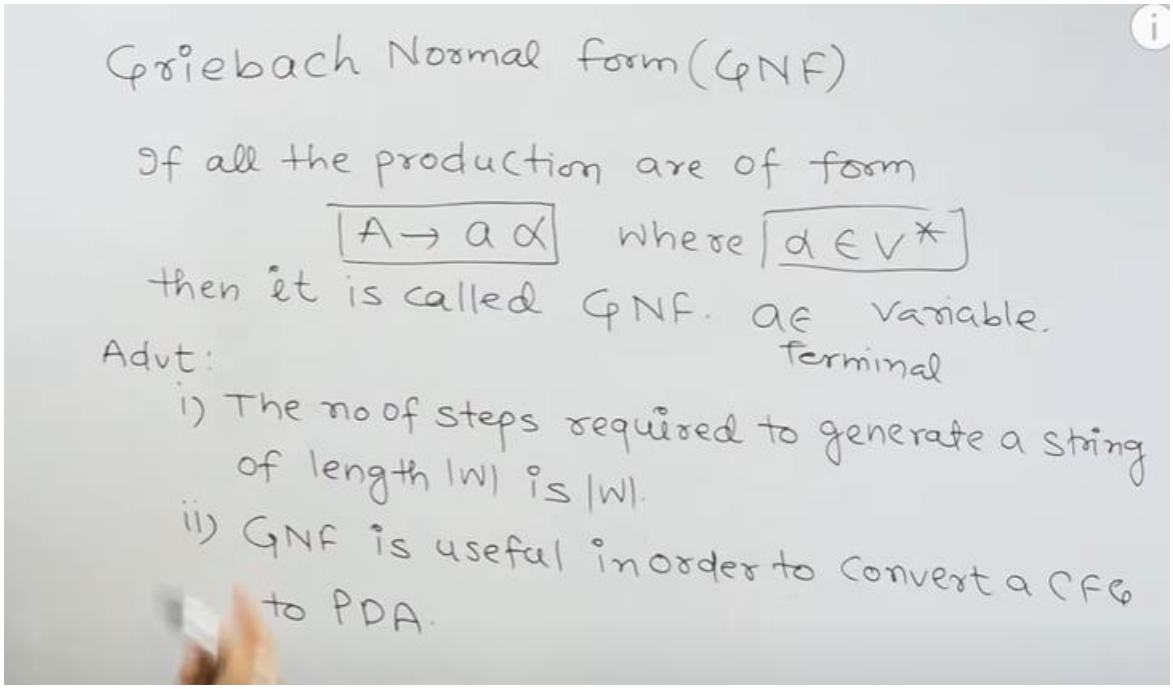
$A \to aBB$

$B \to Aa \mid b \mid a$





Greibach Normal Form



Greibach Normal Form

A CFG is in Greibach Normal Form if the productions are in the following forms:

$$A \rightarrow b$$
$$A \rightarrow bC_1C_2 \ldots\ldots C_n$$

where $A, C_1, \ldots\ldots, C_n$ are Non-Terminals and b is a Terminal

Steps to convert a given CFG to GNF:

Step 1: Check if the given CFG has any Unit Productions or Null Productions and Remove if there are any (using the Unit & Null Productions removal techniques discussed in the previous lecture)

Step 2: Check whether the CFG is already in Chomsky Normal Form (CNF) and convert it to CNF if it is not. (using the CFG to CNF conversion technique discussed in the previous lecture)

Step 3: Change the names of the Non-Terminal Symbols into some A_i in ascending order of i

Steps to convert a given CFG to GNF:

Step 1: Check if the given CFG has any Unit Productions or Null Productions and Remove if there are any (using the Unit & Null Productions removal techniques discussed in the previous lecture)

Step 2: Check whether the CFG is already in Chomsky Normal Form (CNF) and convert it to CNF if it is not. (using the CFG to CNF conversion technique discussed in the previous lecture)

Step 3: Change the names of the Non-Terminal Symbols into some A_i in ascending order of i

Example:
$S \rightarrow CA \mid BB$
$B \rightarrow b \mid SB$
$C \rightarrow b$
$A \rightarrow a$

Replace:
S with A_1
C with A_2
A with A_3
B with A_4

We get:
$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$
$A_4 \rightarrow b \mid A_1 A_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Step 4: Alter the rules so that the Non -Terminals are in ascending order, such that, If the Production is of the form $A_i \rightarrow A_j x$, then, $i < j$ and should never be $i \geqslant j$

$A_4 \rightarrow b \mid A_1 A_4$

$A_4 \rightarrow b \mid A_2 A_3 A_4 \mid A_4 A_4 A_4$

$A_4 \rightarrow b \mid b A_3 A_4 \mid A_4 A_4 A_4$

↓

Left Recursion

Step 5: Remove Left Recursion

Introduce a New Variable to remove the Left Recursion

$A_4 \rightarrow b \mid b\,A_3\,A_4 \mid A_4\,A_4\,A_4$

$Z \rightarrow A_4\,A_4\,Z \mid A_4\,A_4$

$A_4 \rightarrow b \mid bA_3\,A_4 \mid bZ \mid bA_3\,A_4\,Z$

Now the grammar is:

$A_1 \rightarrow A_2\,A_3 \mid A_4\,A_4$
$A_4 \rightarrow b \mid b\,A_3\,A_4 \mid bZ \mid b\,A_3\,A_4\,Z$
$Z \rightarrow A_4A_4 \mid A_4\,A_4\,Z$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

$\mid bA_3\,A_4 \mid bZ \mid bA_3\,A_4\,Z$

Now the grammar is:

$A_1 \rightarrow A_2\,A_3 \mid A_4\,A_4$
$A_4 \rightarrow b \mid b\,A_3\,A_4 \mid bZ \mid b\,A_3\,A_4\,Z$
$Z \rightarrow A_4A_4 \mid A_4\,A_4\,Z$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

$A_1 \rightarrow bA_3 \mid bA_4 \mid bA_3\,A_4\,A_4 \mid bZ\,A_4 \mid bA_3\,A_4Z\,A_4$

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$Z \rightarrow bA_4 \mid bA_3A_4\,A_4 \mid bZA_4 \mid bA_3\,A_4Z\,A_4 \mid$
$bA_4Z \mid bA_3A_4A_4Z \mid bZA_4Z \mid bA_3\,A_4\,Z\,A_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

$S \rightarrow abSb \mid aa$

$S \rightarrow AB$

$A \rightarrow aA \mid bB \mid b$

$B \rightarrow b$

$A \rightarrow a$

$A \rightarrow aV^*$

$S \rightarrow AB$

$A \rightarrow aA \mid bB \mid b$

$B \rightarrow b$

$A \rightarrow a$

$A \rightarrow aV^*$

$S \rightarrow aAB \mid bBB \mid bB$

$A \rightarrow aA \mid bB \mid b$

$B \rightarrow b$

Steps for Converting CFG into GNF

Step1 : Convert the grammar into CNF

If the given grammar is not in CNF, Convert it into CNF.

Step2 : If the grammar exists left recursion, eliminate it.

If the given grammar has left recursion eliminate it.

Step3 : In the grammar, Convert the given production rule into GNF form.

If any production rule in the grammar is not in

Example :-

$S \rightarrow XB \mid AA$

$A \rightarrow a \mid SA$

$B \rightarrow b$

$X \rightarrow a$

$X \rightarrow a$

on :- The given grammar G is already in CNF & there is no left recursion.

The production rule $A \rightarrow SA$ is not in GNF,

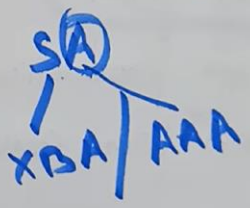
So we substitute $S \rightarrow XB \mid AA$ in the production rule $A \rightarrow SA$ as:

$S \rightarrow XB \mid AA$

$A \rightarrow a \mid XBA \mid AAA$

$B \rightarrow b$

$X \rightarrow a$



The production rule $S \rightarrow XB$ & $B \rightarrow XBA$ is not in GNF,

So we substitute $X \rightarrow a$ in the production rule as:

$S \rightarrow aB \mid AA$

$A \rightarrow a \mid aBA \mid AAA$

$B \rightarrow b$

$X \rightarrow a$

left recursion

We will remove ~~null~~ ~~production~~ left recursion ($A \to AAA$) we get

{ Introduce new variable to remove left recursion one 'A' in production is replace with new variable & without new variable

$S \to aB \mid AA$

$A \to ac \mid aBAC \mid a \mid aBA$

$C \to AAC \mid AA$

$B \to b$

$x \to a$

$AAA \to C$

$C \to AAC \mid AA$

---

e production rule $S \to AA$, $C \to AA$ is not in GNF,

so we substitute $A \to ac \mid aBAC \mid a \mid aBA$ in $S \to AA$ as:

$C \to AA$

$S \to aB \mid acA \mid aBACA \mid aA \mid aBAA$

$A \to ac \mid aBAc \mid a \mid aBA$

$C \to AAC$

$C \to acA \mid aBACA \mid aA \mid aBAA$

$B \to b$

$x \to a$

The production rule $C \to AAC$ is not in GNF, so we Substitute $A \to ac \mid aBAc \mid a \mid aBA$ in production rule $C \to AAC$ as:

The production rule $C \to AAC$ is not in GNF, so we Substitute $A \to ac \mid aBAc \mid a \mid aBA$ in production rule $C \to AAC$ as: $C \to acAC \mid aBAcAC \mid aAC \mid aBAAC$

converting CFG to GNF example-1 | TOC | Lec-73 | Bhanu Priya

$A \Rightarrow a$

$A \rightarrow aBcd \ldots$

$S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$

$A \rightarrow ac \mid aBAc \mid a \mid aBA$

$C \rightarrow aCAc \mid aBACAC \mid aAC \mid aBAAc$

$C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$

$B \rightarrow b$

$X \rightarrow a$

ence this is the GNF for the Grammar G.

---

mple

$S \rightarrow CA \mid BB$

$B \rightarrow b \mid SB$

$C \rightarrow b$

$A \rightarrow a$

CFG ⇓ CNF ⇒ GNF

CFG ⇒ GNF

⇓

$A \rightarrow b$ (a)

$A \rightarrow bC_1C_2 \ldots C_n$

$A \rightarrow B$

Step 1: there are no unit production

Step 2: Their production is already CNF

Step 3: Convert CNF into GNF

replace S with A_1,
C with A_2
A with A_3
B with A_4

Step 4 :- If the production is of the form $A_i \rightarrow A_j X$ then $i < j$ & should never be $i \geq j$ X

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

$A_4 \rightarrow b \mid A_1 A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

$A_4 \rightarrow A_1 A_4$

$\Rightarrow A_4 \rightarrow b \mid A_2 A_3 A_4 \mid A_4 A_4 A_4$

$\Rightarrow A_4 \rightarrow b \mid b A_3 A_4 \mid A_4 A_4 A_4$

↓

left recursia

Step 5 :- Introduce new variable to remove the left recursia.

$Z \rightarrow A_4 A_4 Z \mid A_4 A_4$

$\Rightarrow A_4 \rightarrow b \mid b A_3 A_4 \mid b Z \mid b A_3 A_4 Z \mid$

$AA\underline{A}$

↓

$AAC \mid AA$

$A_4 \rightarrow b \mid bA_3A_4 \mid A_4A$

$AA\underline{A}$

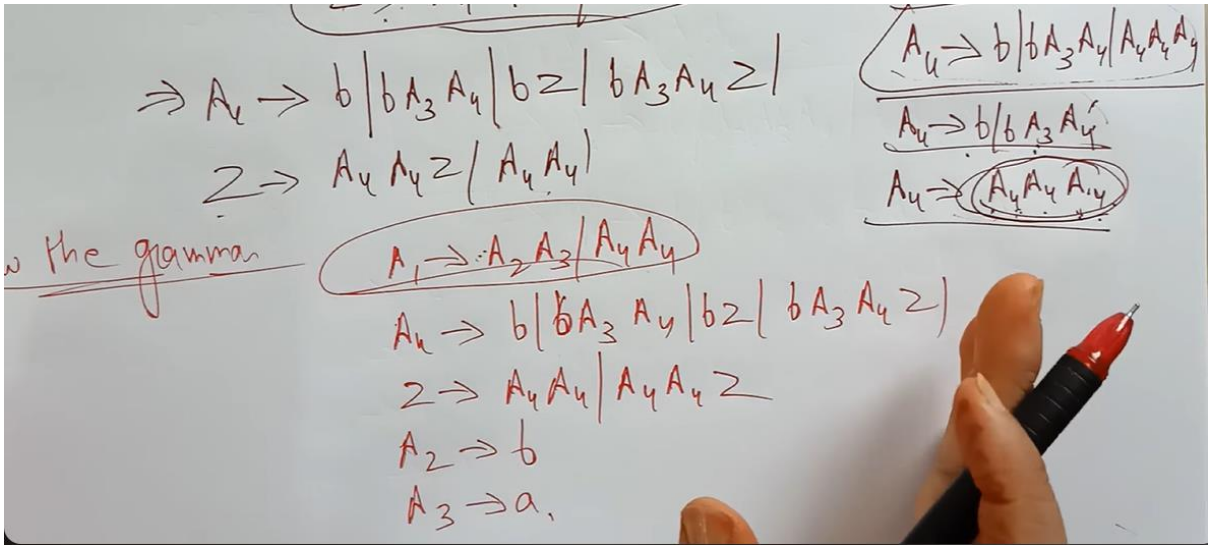
↓

$AAC \mid AA$

$A_4 \rightarrow b \mid bA_3A_4 \mid A_4A_4A_4$

$A_4 \rightarrow b \mid bA_3A_4$

$A_4 \rightarrow A_4A_4A_4.$



$A_1 \to bA_3 \mid bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4$

$A_4 \to b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$Z \to bA_4 \mid bA_3A_4 \mid bZA_4 \mid$

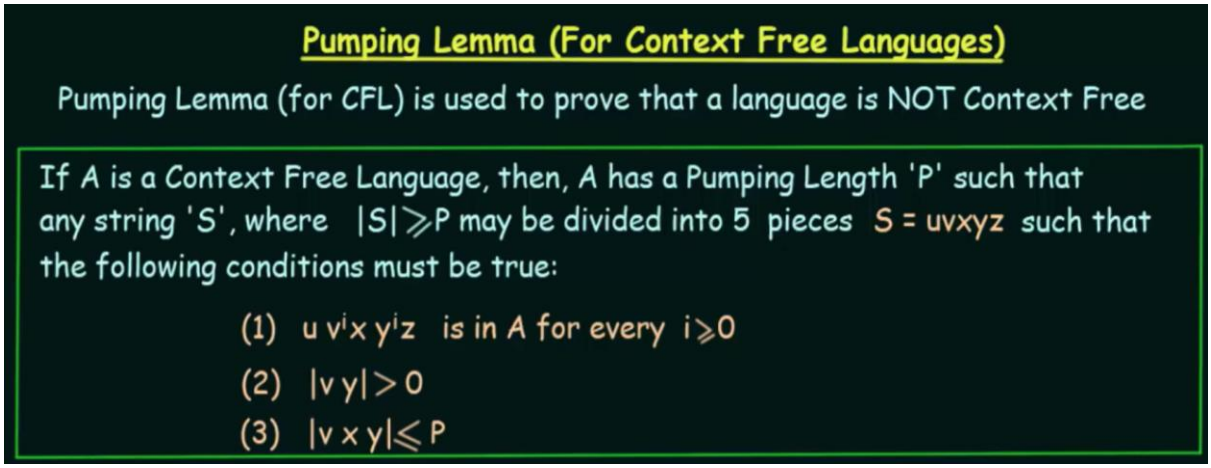
$bA_3A_4ZA_4 \mid$

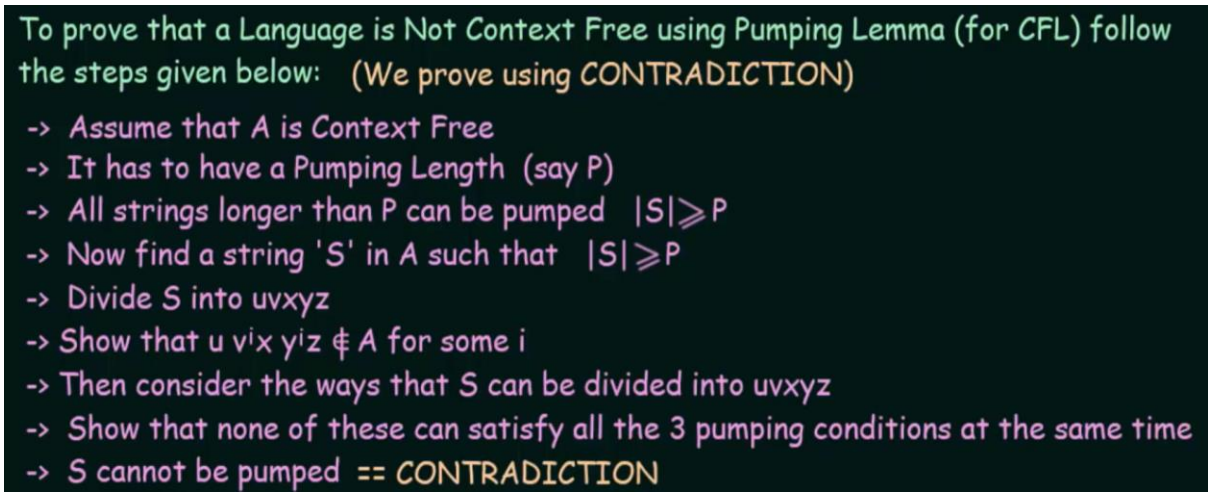
$bA_4Z \mid bA_3A_4A_4Z \mid bZA_4Z \mid bA_3A_4ZA_4Z$

$A_2 \to b$

$A_3 \to a$

Pumping Lemma for Context free languages (CFL)





Pumping Lemma (for Context Free Languages) - Example (Part-1)

Show that $L = \{ a^N b^N c^N \mid N \geq 0 \}$ is Not Context Free

-> Assume that L is Context Free
-> L must have a pumping length (say P)
-> Now we take a string S such that $S = a^P b^P c^P$
-> We divide S into parts u v x y z

Eg. $P = 4$ So, $S = a^4 b^4 c^4$

Case I : v and y each contain only one type of symbol

a a a a b b b b c c c c
u v x y z

$uv^i xy^i z$ $(i = 2)$

$uv^2 xy^2 z$

a a a a a a b b b b c c c c c

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

Case II : Either v or y has more than one kind of symbols

a a a a b b b b c c c c
u v x y z

$uv^i xy^i z$ $(i = 2)$

$uv^2 xy^2 z$

$a^N b^N c^N$

a a a a b b a a b b b b c c c c $\notin L$

Pumping Lemma (for Context Free Languages) - Example (Part-2)

Show that $L = \{ ww \mid w \in \{0,1\}^* \}$ is NOT Context Free

-> Assume that L is Context Free
-> L must have a pumping length (say P)
-> Now we take a string S such that $S = 0^P 1^P 0^P 1^P$
-> We divide S into parts u v x y z

Case 1: vxy does not straddle a boundary

0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1
u v x y z

Eg. $P = 5$ So, $S = 0^5 1^5 0^5 1^5$

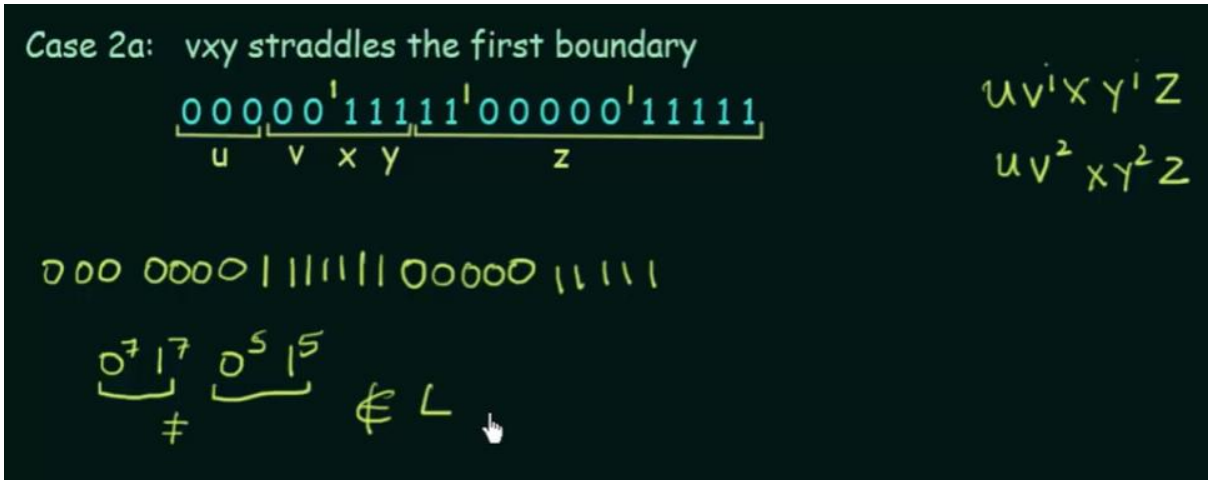
$uv^i xy^i z$

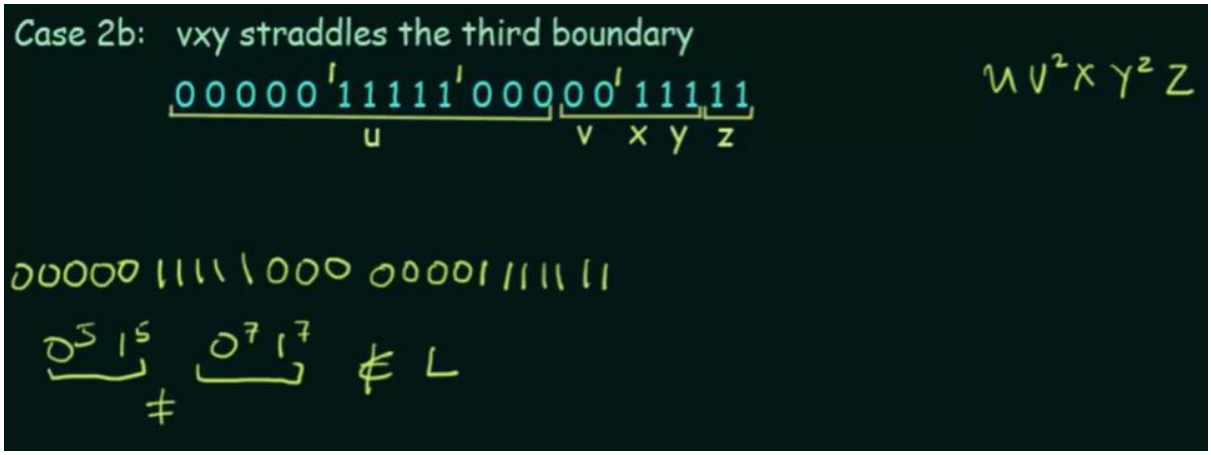
$uv^2 xy^2 z$

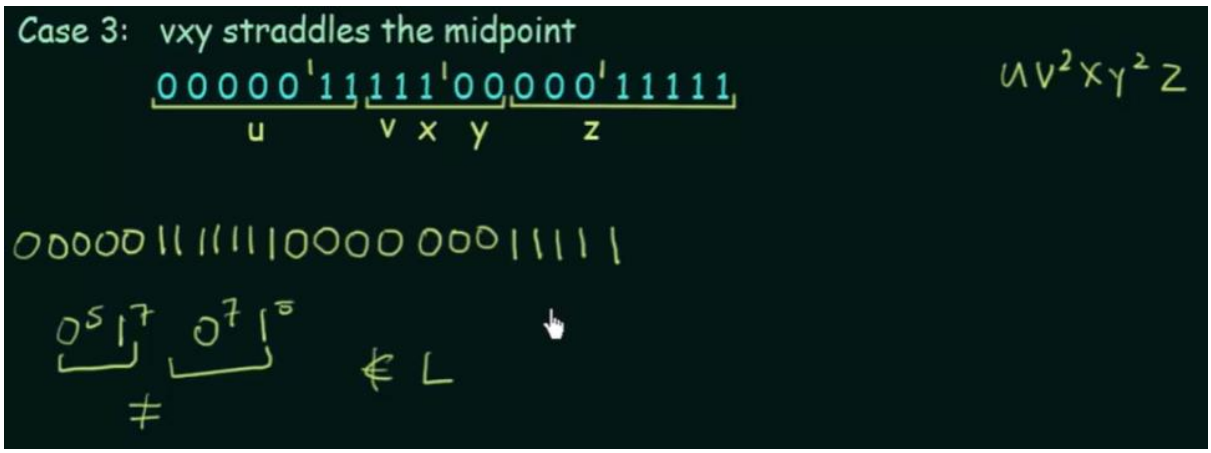
0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1

$0^5 1^7 0^5 1^5 \notin L$

$\neq$







Pushdown Automata

Pushdown Automata (Introduction)

A Pushdown Automata (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 a very limited memory but PDA has more memory
- -> PDA = Finite State Machine + A Stack

A stack is a way we arrange elements one on top of another

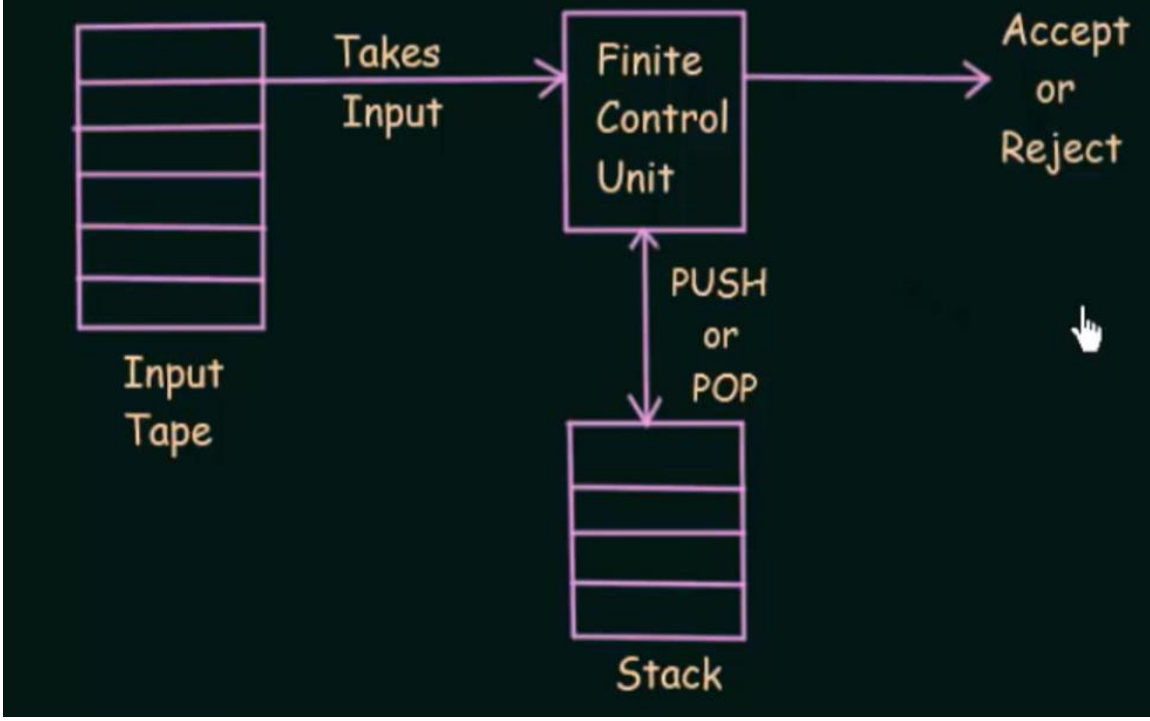
A stack does two basic operations:

PUSH: A new element is added at the Top of the stack

POP: The Top element of the stack is read and removed

A Pushdown Automata has 3 components:

1) An input tape
2) A Finite Control Unit
3) A Stack with infinite size



Pushdown Automata (Formal Definition)

A Pushdown Automata is formally defined by 7 Tuples as shown 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 = The Start State
- z_0 = The Start Stack Symbol
- F = The set of Final / Accepting States

δ takes as argument a triple $\delta(q, a, X)$ where:

(i) q is a State in Q

(ii) a is either an Input Symbol in Σ or $a = \in$

(iii) X is a Stack Symbol, that is a member of Γ

The output of δ is finite set of pairs (p, γ) where:

p is a new state

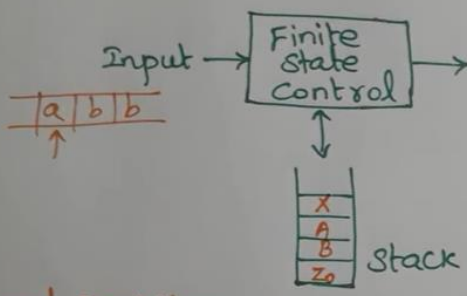
γ is a string of stack symbols that replaces X at the top of the stack

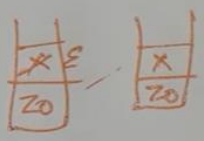
Eg. If $\gamma = \in$ then the stack is popped

If $\gamma = X$ then the stack is unchanged

If $\gamma = YZ$ then X is replaced by Z and Y is pushed onto the stack

ε-NFA + stack.





Formal Definition

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

Q - Finite set of states

Σ - Input alphabet

Γ - stack alphabet

q_0 - start state

start stack symbol

a is in Σ or $a = \varepsilon$

F - Set of final states

δ - Transition Function

$\delta(q, a, X) = (p, \gamma)$

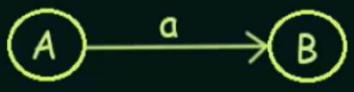
(i) $\gamma = \varepsilon$, POP

(ii) $\gamma = X$, No change

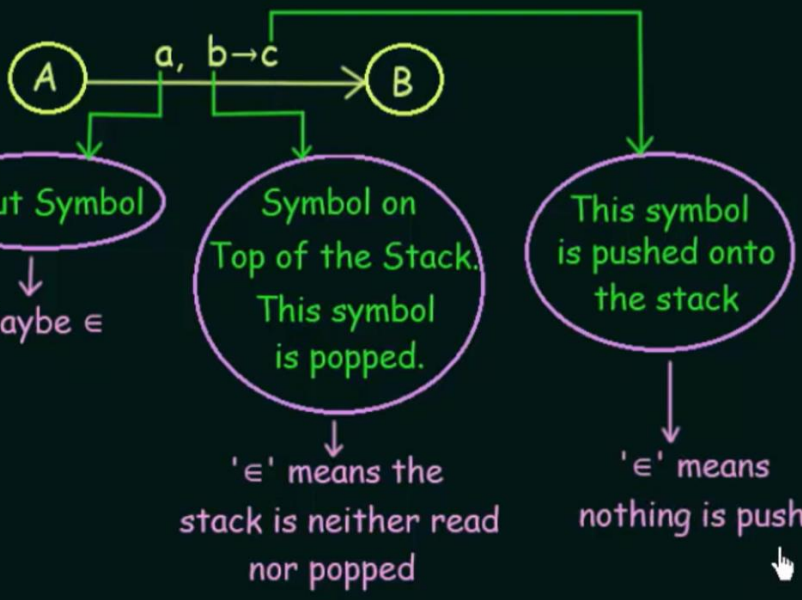
(iii) $\gamma = yz$, Push.

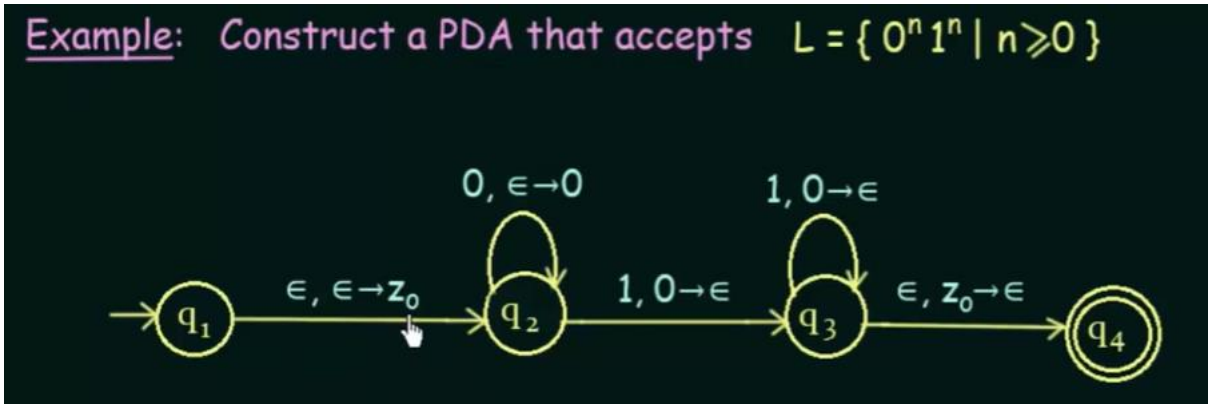
Pushdown Automata (Graphical Notation)

Finite State Machine

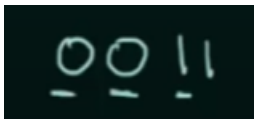


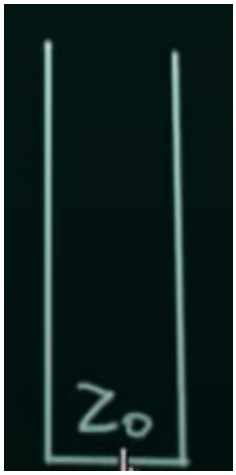
Pushdown Automata



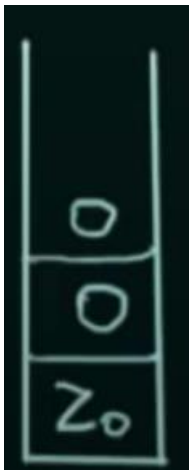


Stepwise

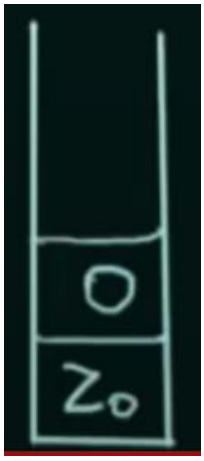




Zo or $ sign is used to denote the bottommost element of the stack.

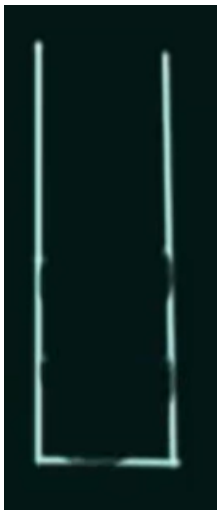


At q2 we get input as 2 zeroes and we push 2 0's into the stack according to the automaton.



After we get an input 1 at q2 we pop the zero and push nothing and go to q3.

At q3 we get input 1 and again pop the remaining 0

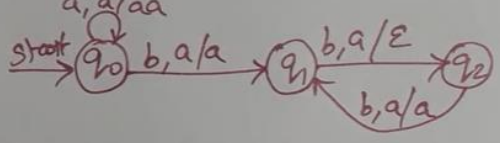


Now the next symbol is an epsilon symbol, and nothing is pushed into the stack and we pop the Zo out of the stack and reach the goal state.

We accept the NFA in cases when we either reach the final state or the stack is empty.

Give a PDA to accept the following language

$L = \{a^n b^{2n} \mid n \geq 1\}$ by final state.



$\{abb, aabbbb, \ldots\}$

aabbbbε

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

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

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

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

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

aabbbbε

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

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

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

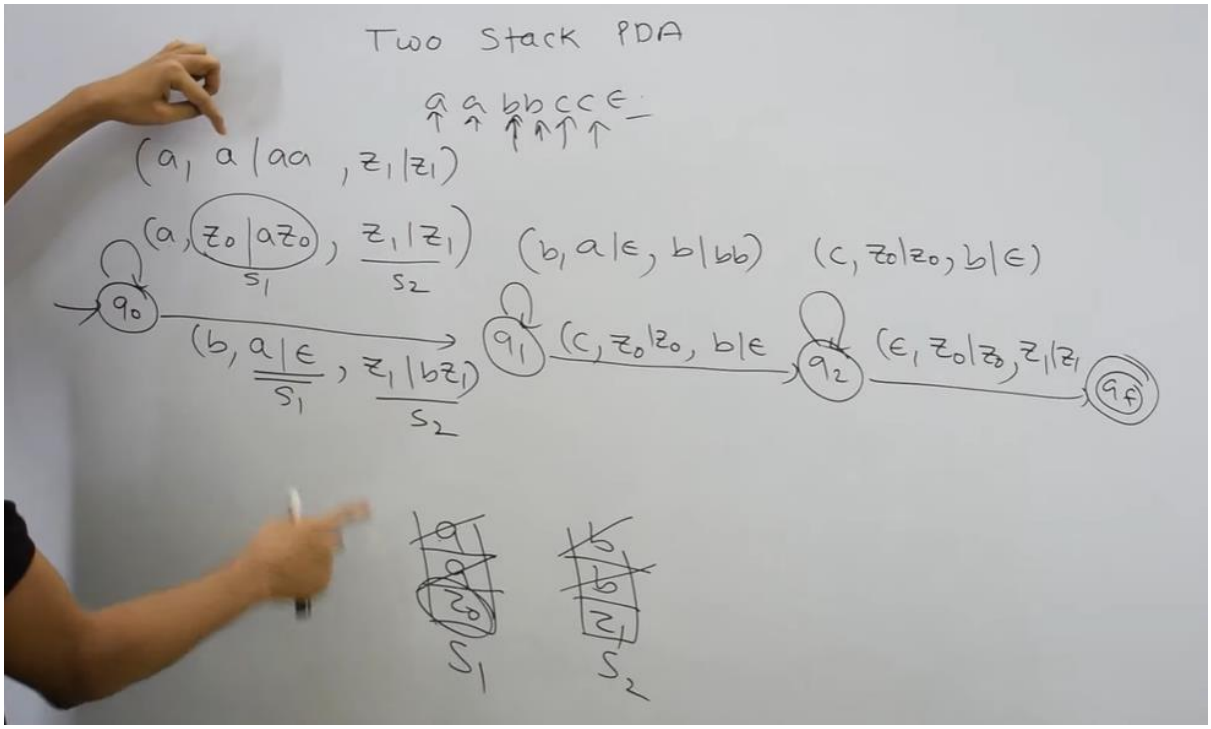
$\delta(q_1, b, a) = (q_2, \varepsilon)$

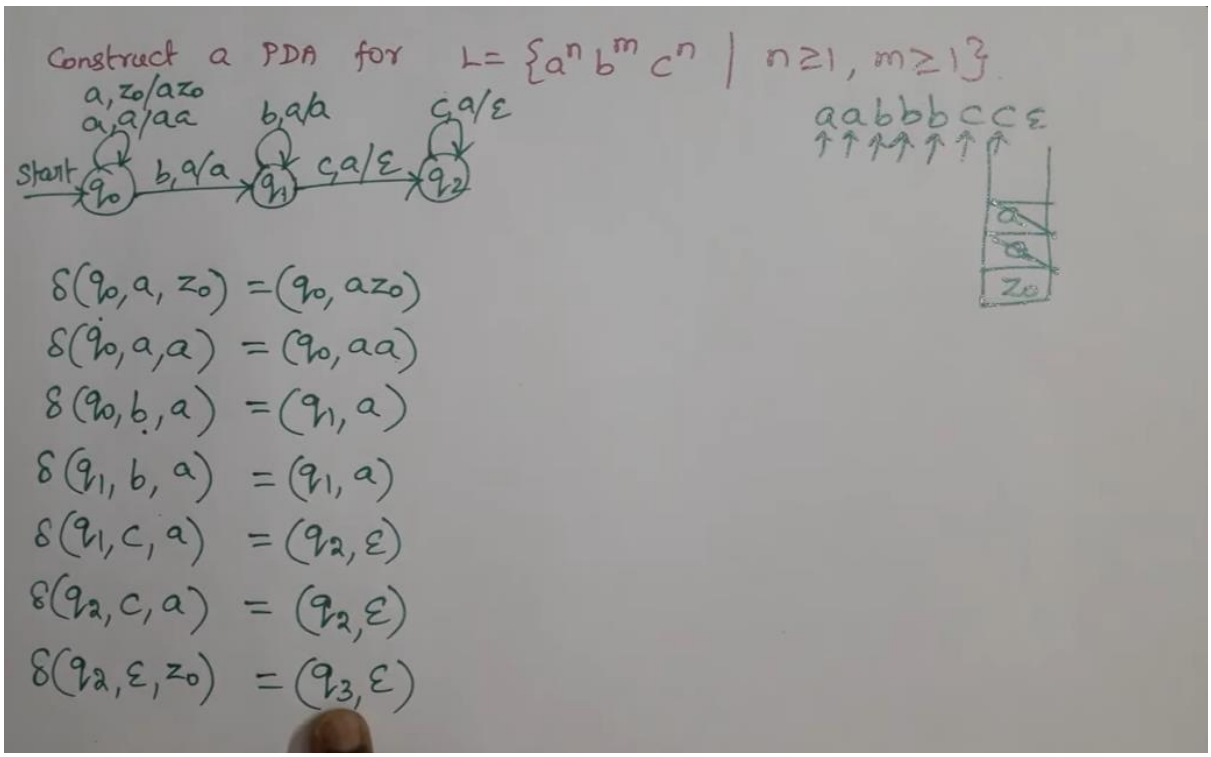
$\delta(q_2, b, a) = (q_1, a)$

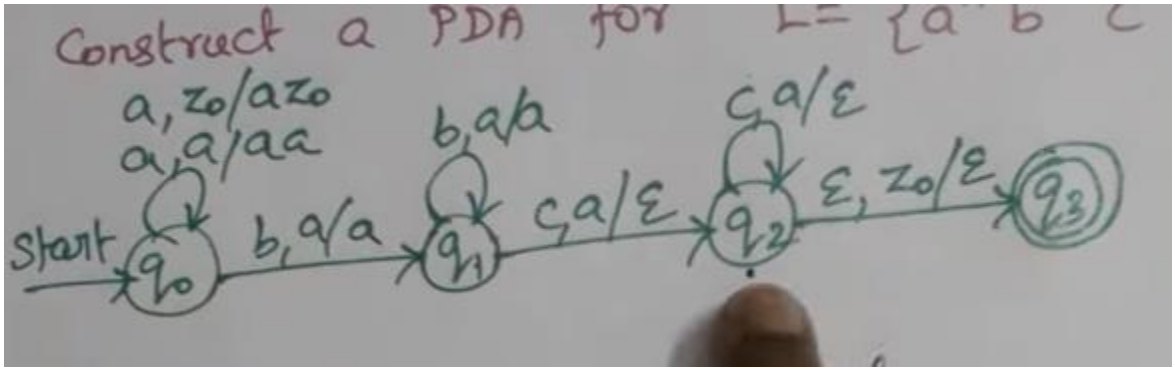
aabbbbε

$\delta(q_2, \varepsilon, z_0) = (q_3, z_0)$

PDA, $P = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \{q_3\}$


Two Stack PDA
a a b b c c ε
(a, a | aa, z1 | z1)
(a, z0 | az0, z1 | z1)
S1
S2
(b, a | ε, z1 | bz1)
(b, a | ε, b | bb)
(c, z0 | z0, b | ε)
(ε, z0 | z0, z1 | z1)
q0
q1
q2
qf
S1
S2


Construct a PDA for L = {a^n b^m c^n | n≥1, m≥1}
a, z0/az0
a, a/aa
b, a/a
c, a/ε
start
q0
q1
q2
b, a/a
c, a/ε
a a b b c c ε
z0
δ(q0, a, z0) = (q0, az0)
δ(q0, a, a) = (q0, aa)
δ(q0, b, a) = (q1, a)
δ(q1, b, a) = (q1, a)
δ(q1, c, a) = (q2, ε)
δ(q2, c, a) = (q2, ε)
δ(q2, ε, z0) = (q3, ε)



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

$\delta(q_1, c, a) = (q_2, \varepsilon)$

$\delta(q_2, c, a) = (q_2, \varepsilon)$

$\delta(q_2, \varepsilon, z_0) = (q_3, \varepsilon)$

PDA, $P = (\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \{a, z_0\}, \delta, q_0, z_0, \{q_3\})$

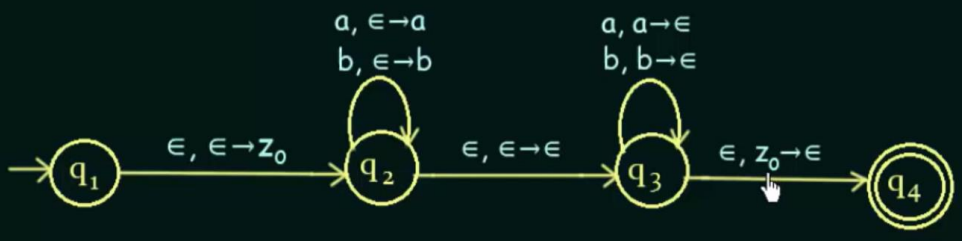
Pushdown Automata - Example (Even Palindrome) PART-1

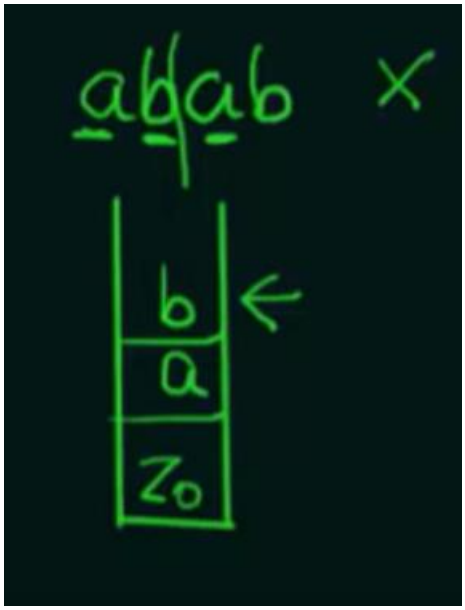
Construct a PDA that accepts Even Palindromes of the form

$L = \{ ww^R \mid w = (a+b)^+ \}$

PALINDROMES: A word or sequence that reads the same backwards as forwards.

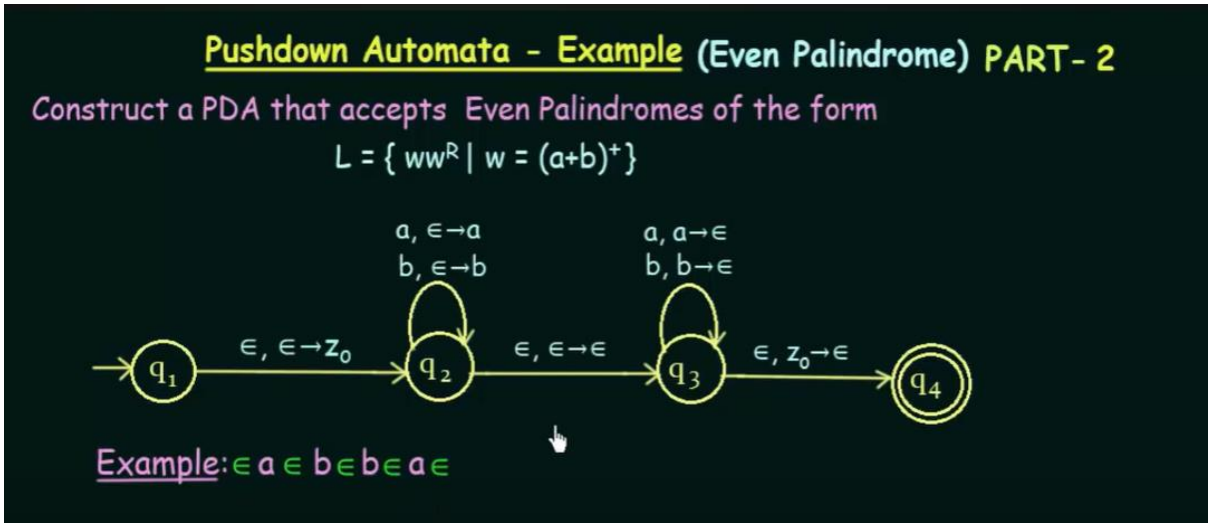
Examples: NOON
NO LEMON NO MELON
123321
abba
RACECAR

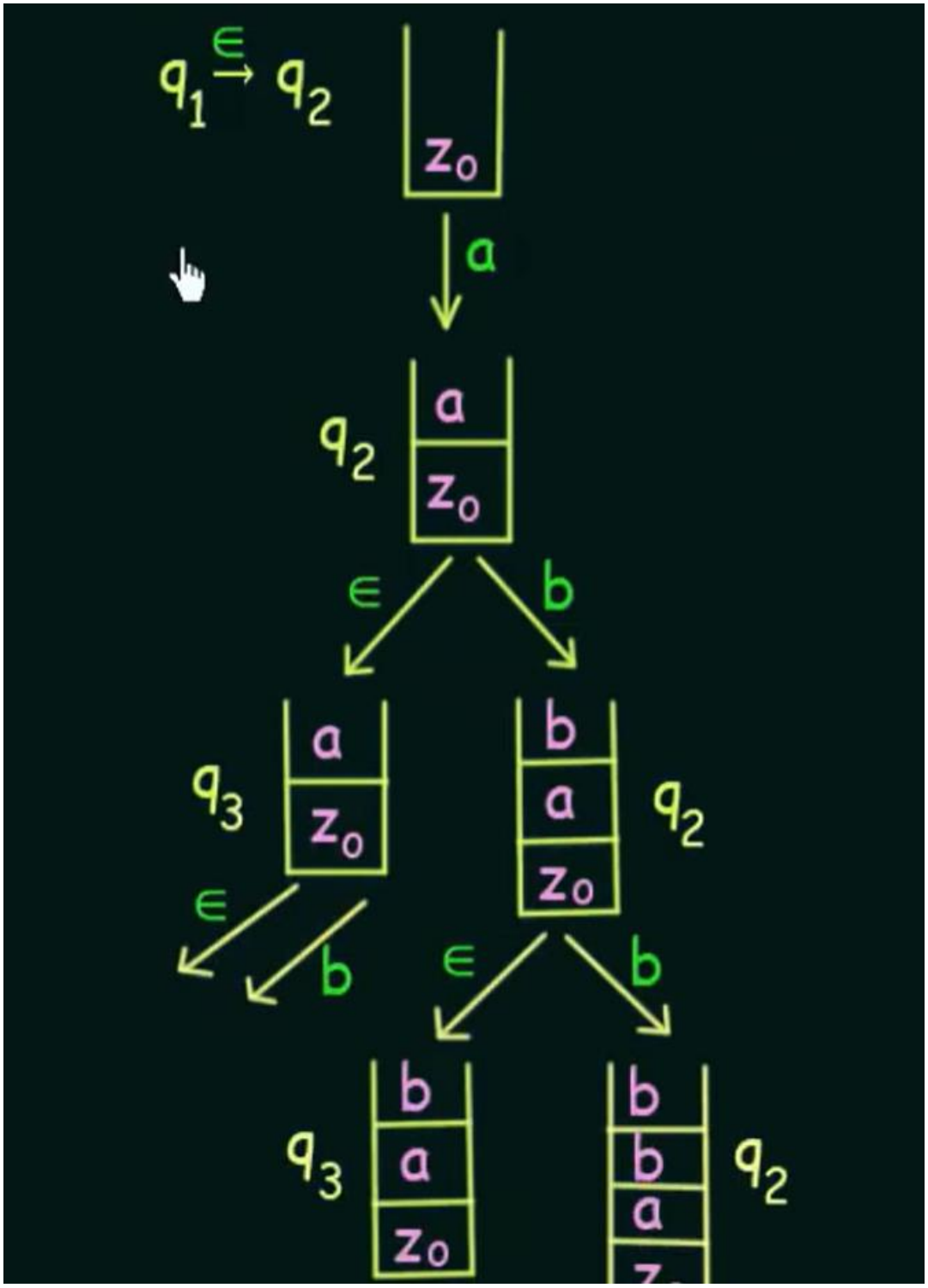


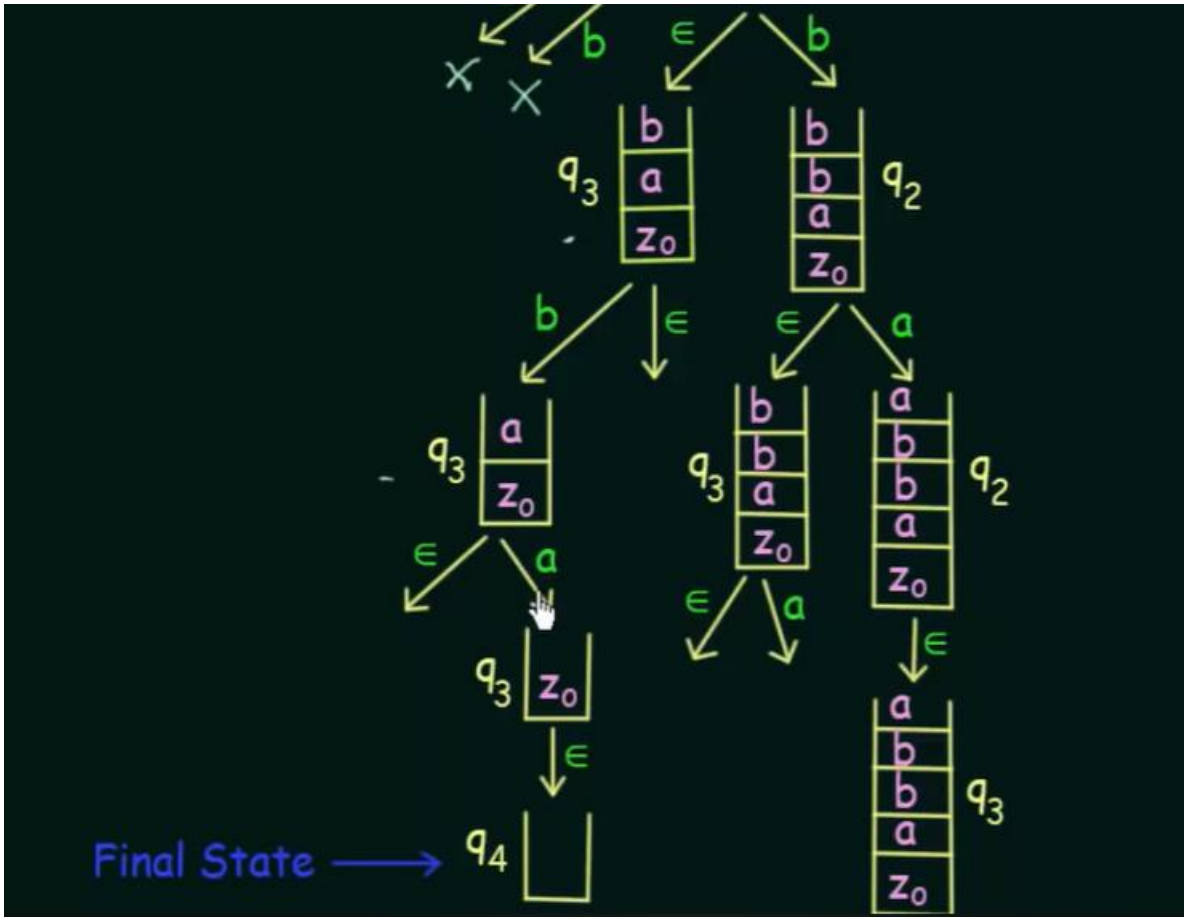


This is not a palindrome as after pushing a & b we can't pop them at step q3.

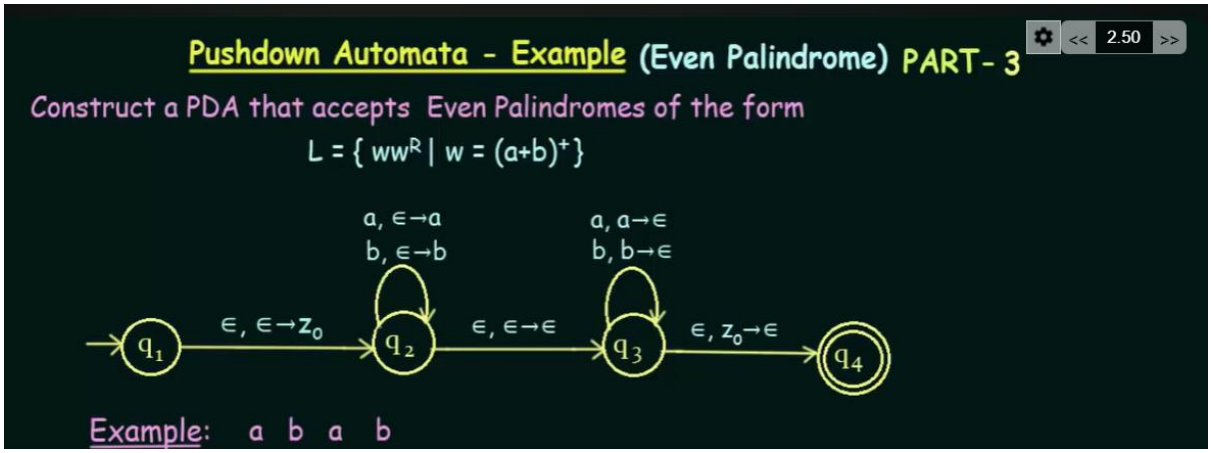
How to know when we have reached the mid of the string

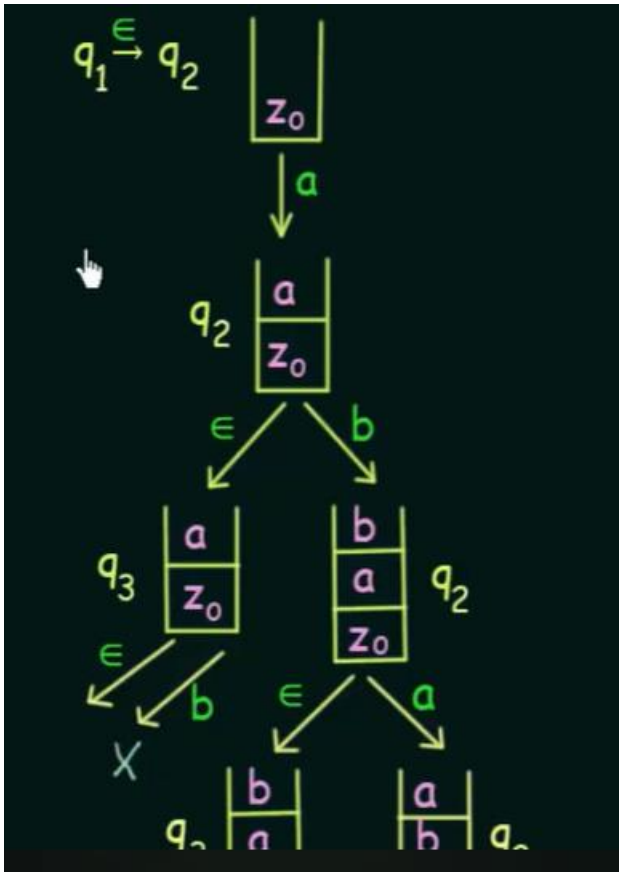


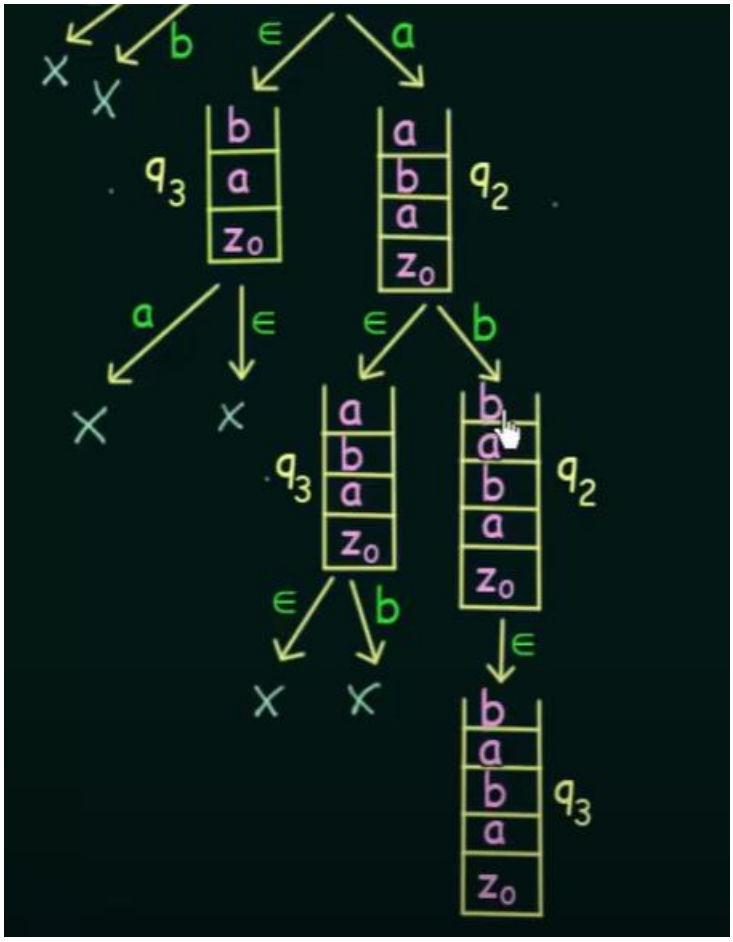

$q_1 \xrightarrow{\in} q_2$
z_0
a
q_2
a
z_0
∈
b
q_3
a
z_0
q_2
b
a
z_0
∈
b
∈
b
q_3
b
a
z_0
q_2
b
b
a



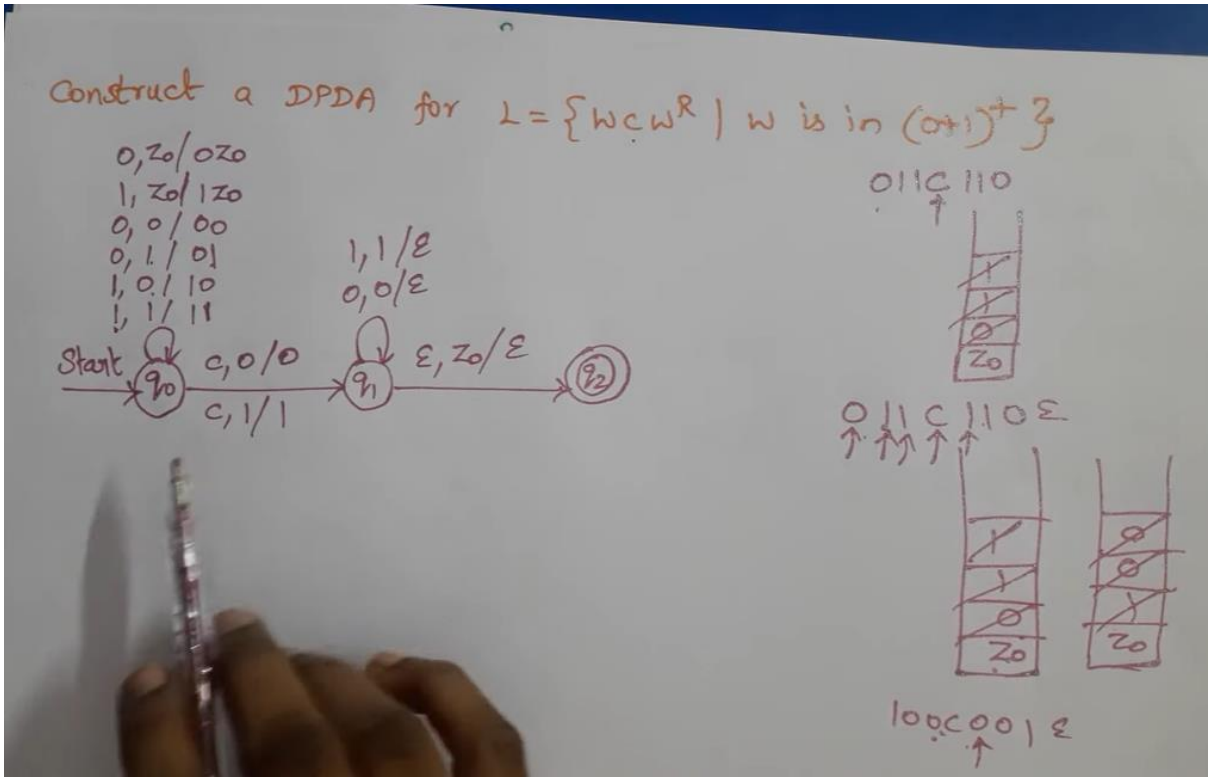
https://www.codingninjas.com/studio/library/pushdown-automata



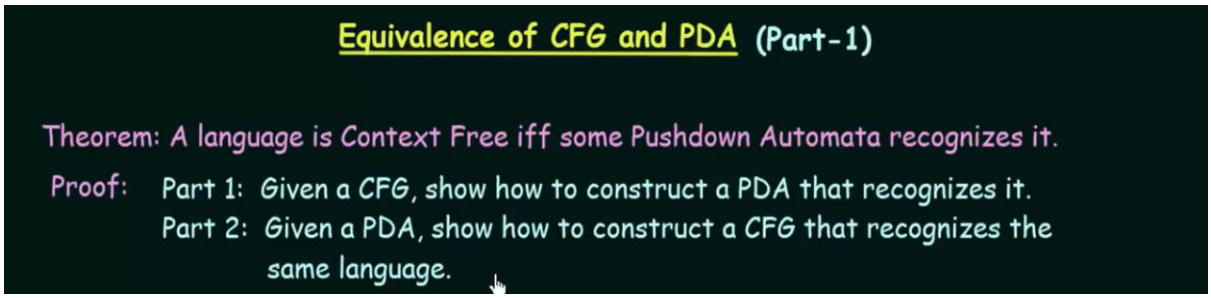

$q_1 \xrightarrow{\in} q_2$
z_0
a
q_2
a
z_0
∈
b
q_3
a
z_0
b
a
z_0
q_2
∈
b
∈
a
X
b
a


X
X
b
∈
a
q_3
b
a
z_0
a
b
a
z_0
q_2
a
∈
∈
b
X
X
q_3
a
b
a
z_0
b
a
b
a
z_0
q_2
∈
b
X
X
∈
b
a
b
a
z_0
q_3

Deterministic PDA



Equivalence of CFG and PDA



Equivalence of CFG and PDA (Part-1)

(From CFG to PDA)

Theorem: A language is Context Free iff some Pushdown Automata recognizes it.

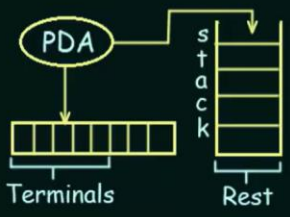
Proof: Part 1: Given a CFG, show how to construct a PDA that recognizes it.

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

Given a grammar

S→ BS|A
A→0A|∈
B→BB1|2

Find or build a PDA



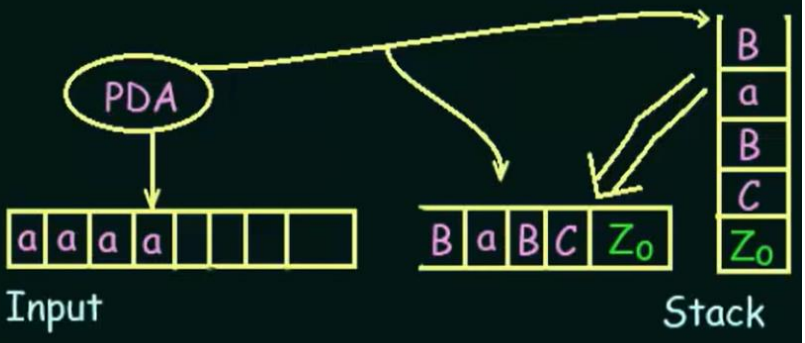
General Form:

aaaa BaBC

Terminals Rest

--> S
--> BS
--> BB1S
--> 2B1S
--> 221S
--> 221A
--> 221∈
--> 221

(Left most derivation)

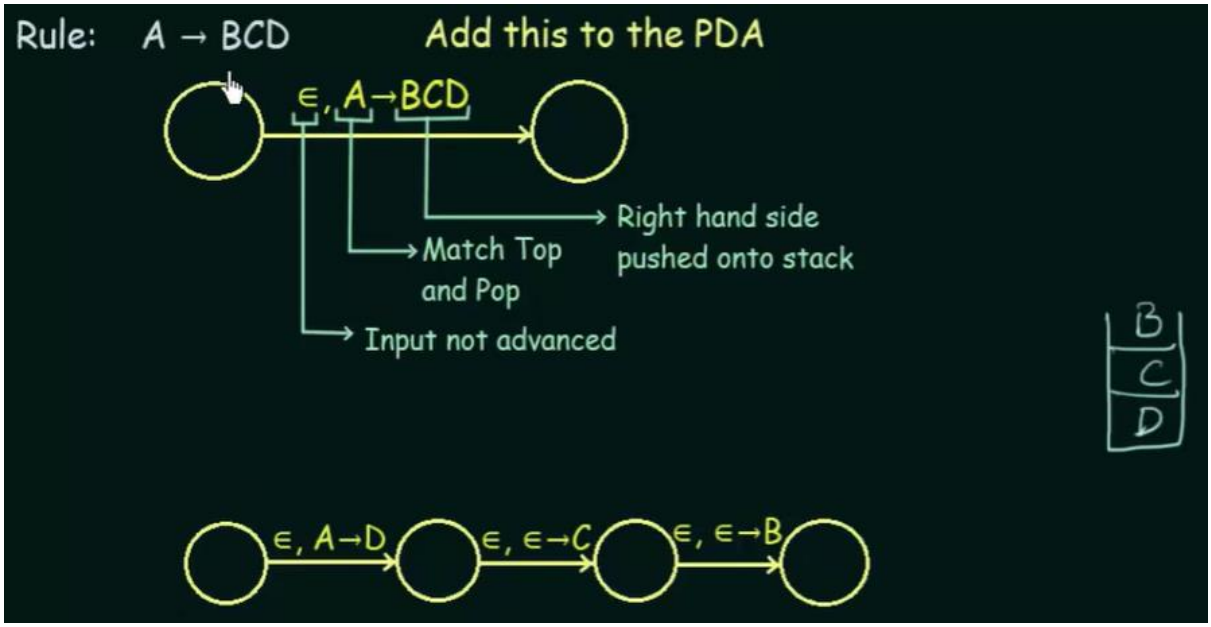


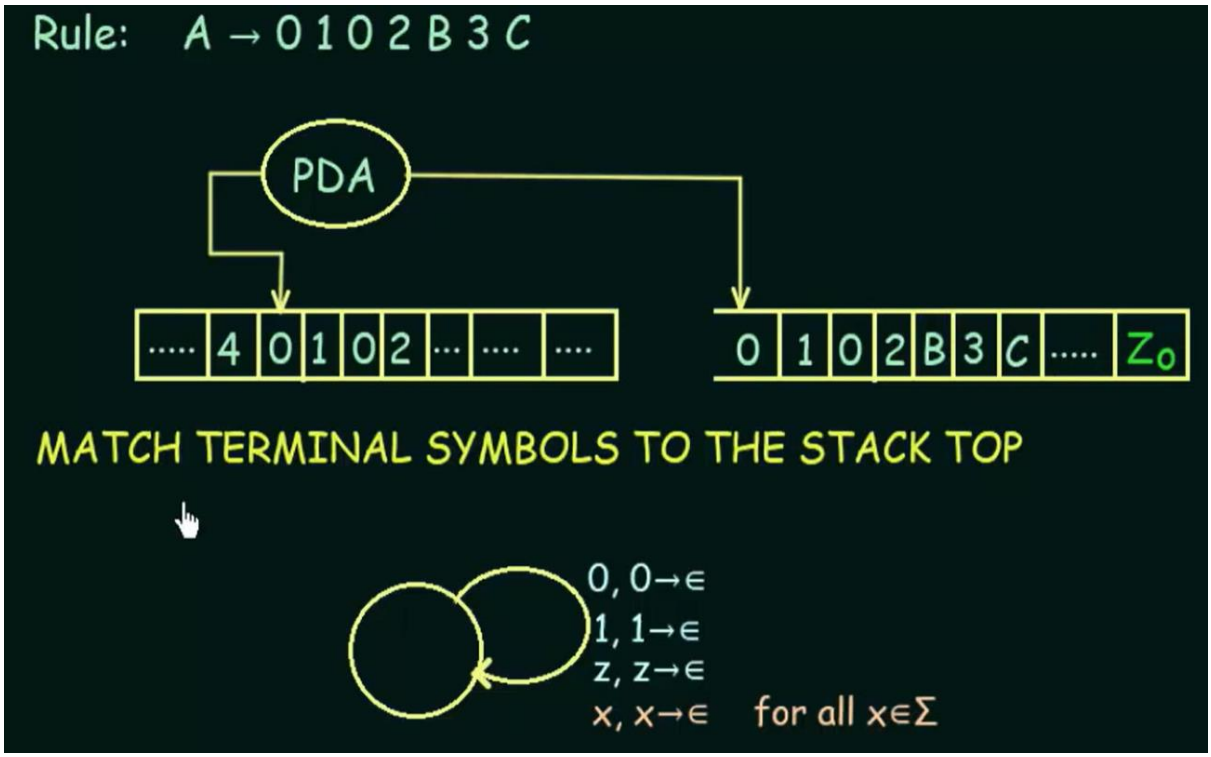
Left Most Derivation: S --> ..... a a a a B a B C --> .....

AT EACH STEP EXPAND LEFT-MOST DERIVATION

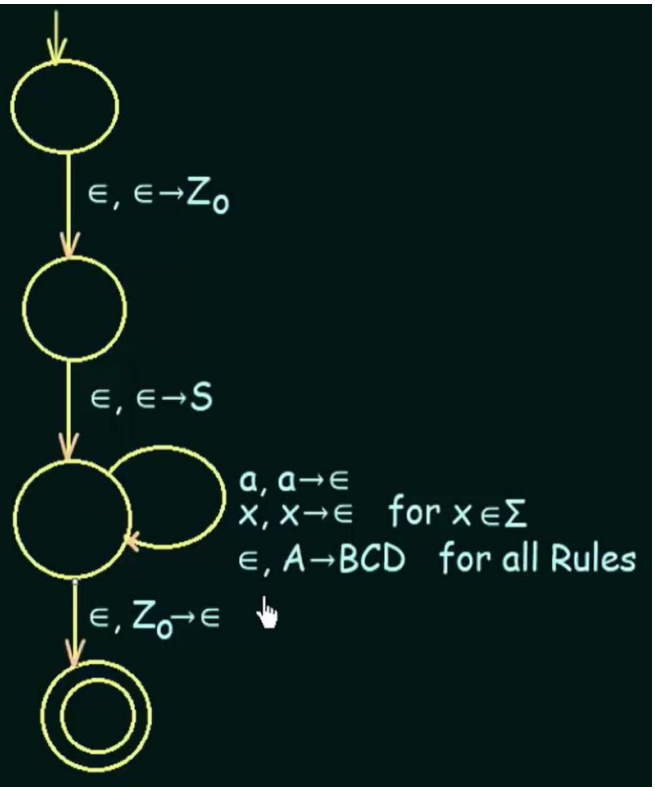
Eg. Rule: B→ASAxBA --> ..... a a a a A S A x B A a B C

→ Match Stack Top to a Rule
→ Pop Stack
→ Push Right Hand Side of Rule onto Stack


Rule: A → BCD
Add this to the PDA
∈, A→BCD
Right hand side pushed onto stack
Match Top and Pop
Input not advanced
B
C
D
∈, A→D
∈, ∈→C
∈, ∈→B


Rule: A → 0 1 0 2 B 3 C
PDA
..... 4 0 1 0 2 ... .
0 1 0 2 B 3 C Z0
MATCH TERMINAL SYMBOLS TO THE STACK TOP
0, 0→∈
1, 1→∈
z, z→∈
x, x→∈
for all x∈Σ

The Final PDA



Equivalence of CFG and PDA (Part-2a)

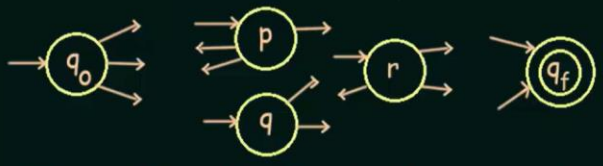
(From PDA to CFG)

Theorem: A language is Context Free iff some Pushdown Automata recognizes it.

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

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

Given a PDA --> Build a CFG from it



Step 1: Simplify the PDA

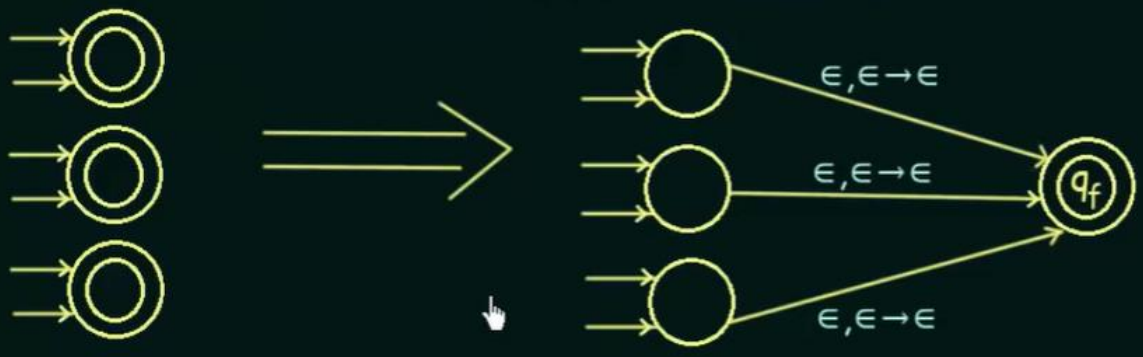
Step2: Build the CFG

There will be a Non-Terminal for every pair of states : A_{pq} , A_{qr} , A_{rq_0}, ...

The starting Non-Terminal will be : $A_{q_0q_f}$

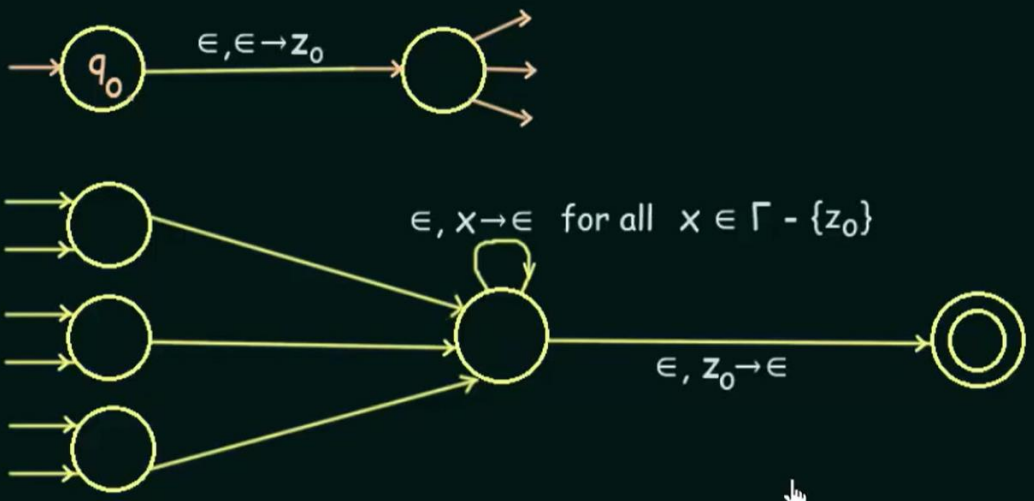
Simplifying the PDA:

1) The PDA should have only one final/accept state.

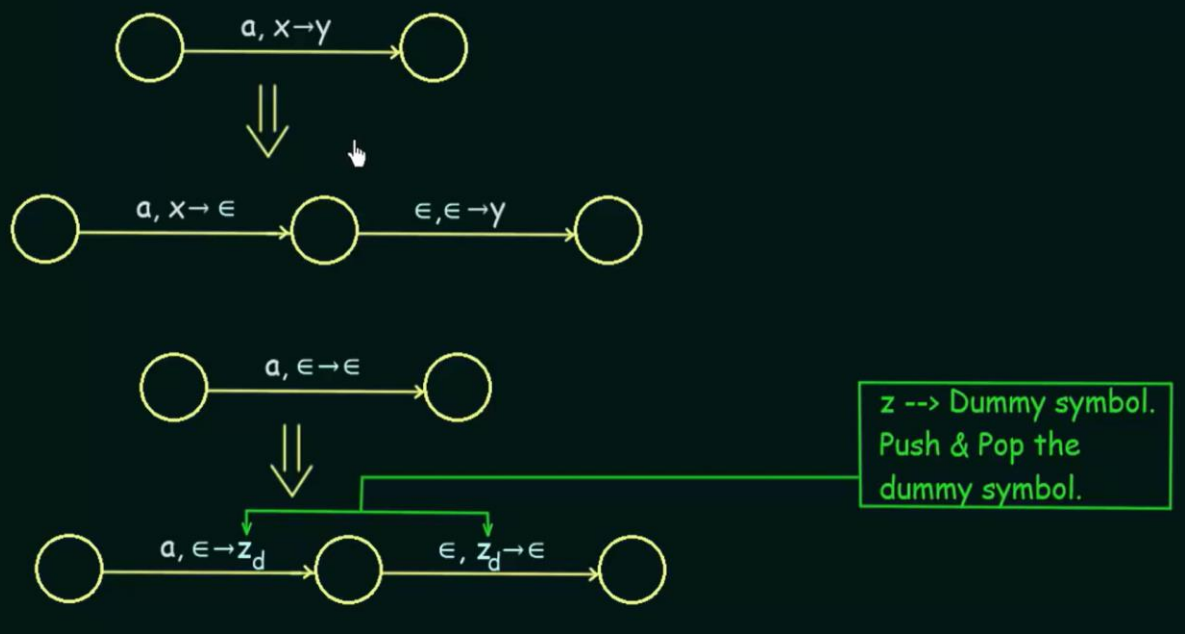


2) The PDA should empty its stack before accepting.

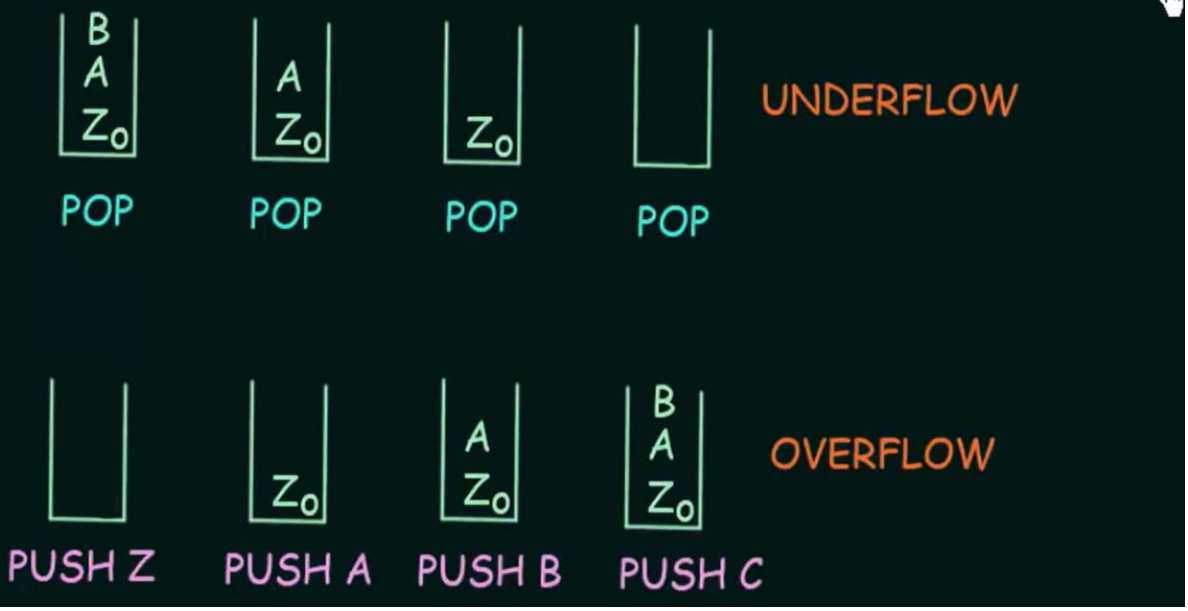
--> Create a new Start State q_0 which pushes z_0 to the stack.



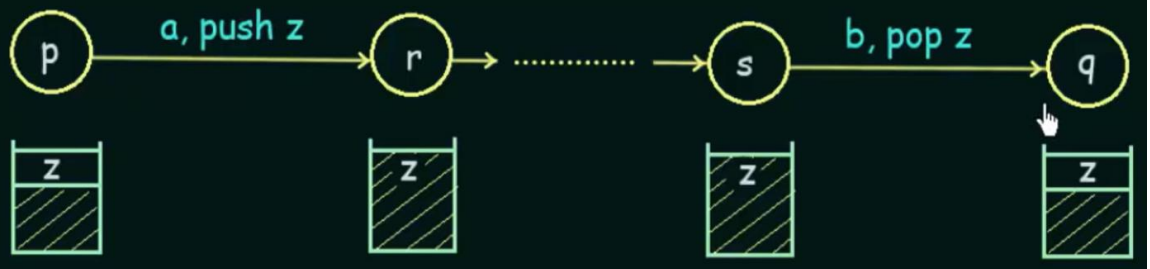
3) Make sure each transitions either Pushes or Pops but does not do both.



--> Start with an empty Stack and finish with an empty Stack



Case-1:



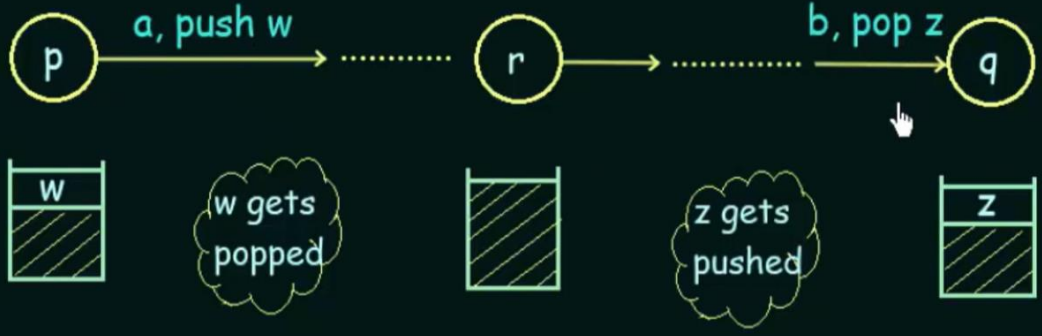
What strings can be generated by following this path ?

--> "a.....b"

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

--> This rule will generate exactly those strings.

Case-2:



What strings can be generated by following this path ?

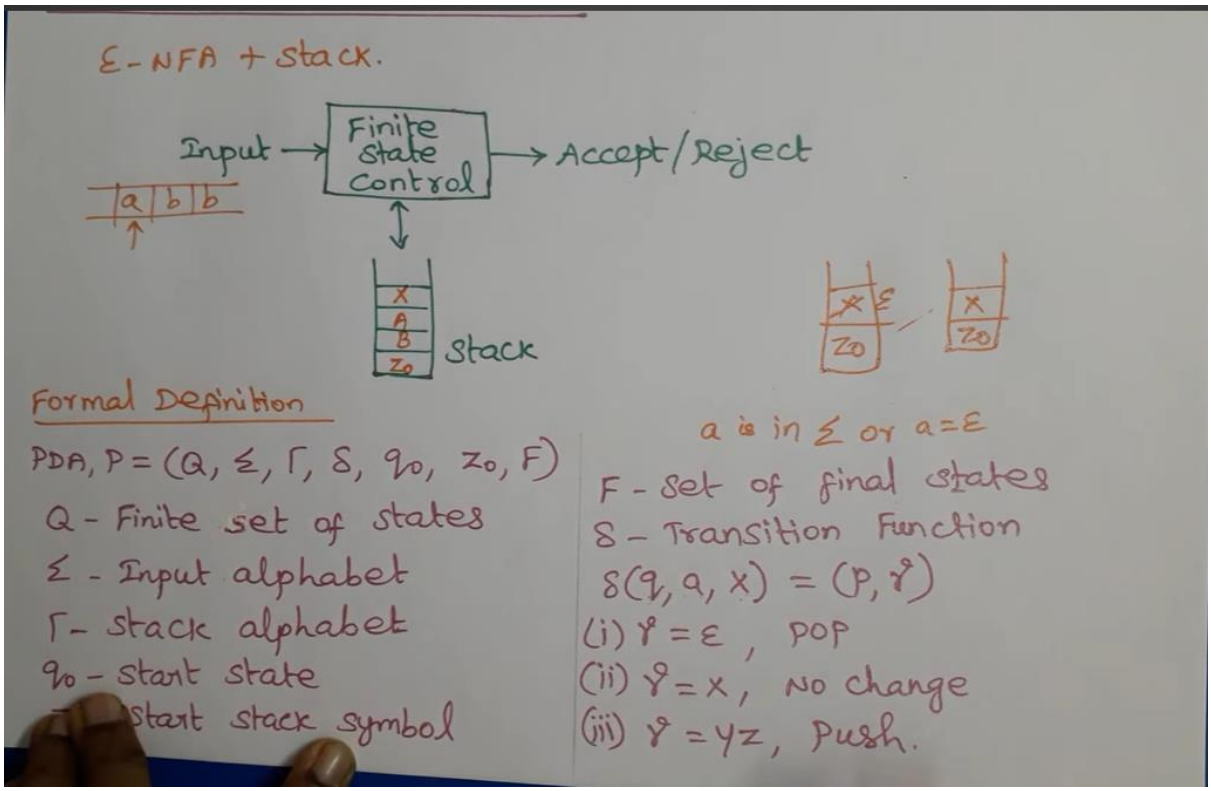
$A_{pq} \rightarrow A_{pr} A_{rq}$

--> This rule will generate exactly those strings.

--> To get from any state (p) to itself, $A_{pp} \rightarrow \epsilon$

--> If the PDA accepts some strings, then there is a way to go from (q_0) to (q_f) that does not modify the stack.

Our Start Non-Terminal is $A_{q_0 q_f}$



Conversion of CFG to PDA

Conversion of CFG to PDA.

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

Let $G = (V, T, R, S)$ be a CFG,

Construct PDA, accepts by empty stack.

PDA, $P = (\{q\}, T, VUT, \delta, q, S)$

Transition Function δ

1. For each variable A,

$\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \to \beta \text{ is in } R\}$

2. For each terminal a,

$\delta(q, a, a) = (q, \varepsilon)$

EXAMPLE

Construct a PDA for the following CFG & test whether "abbabb" is in $N(P)$.

$G = (\{S, A\}, \{a, b\}, R, S)$

$R = \{ S \to AA \mid a$
$A \to SA \mid b \}$

PDA, $P = (\{q\}, \{a, b\}, \{S, A, a, b\}, \delta, q, S)$

$\delta(q, \varepsilon, S) = \{(q, AA), (q, a)\}$

$\delta(q, \varepsilon, A) = \{(q, SA), (q, b)\}$

$\delta(q, a, a) = (q, \varepsilon)$

$\delta(q, b, b) = (q, \varepsilon)$

$(q, abbabb, S) \vdash (q, abbabb, AA)$

$\vdash (q, abbabb, SAA)$

$\vdash (q, abbabb, aAA)$

$\vdash (q, bbabb, AA)$

$\vdash (q, bbabb, bA)$

$\vdash (q, babb, A)$

$\vdash (q, babb, SA)$

$\vdash (q, babb, AAA)$

$\vdash (q, babb, bAA)$

$\vdash (q, abb, AA)$

$\vdash (q, abb, SAA)$

$\vdash (q, abb, aAA)$

$\vdash (q, bb, AA)$

$\vdash (q, bb, bA)$

$\vdash (q, b, A)$

$\vdash (q, b, b)$

$\vdash (q, \varepsilon, \varepsilon)$

"abbabb" is accepted by N(P)

Conversion of PDA to CFG

Conversion of PDA to CFG

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

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

Variables, V

1. special symbol S
2. $[pXq]$ where p, q are states in Q & X is in Γ

Productions, R

1. For all states p,

   $S \rightarrow [q_0 z_0 p]$

2. Let $\delta(q, a, X) = (r, Y_1 Y_2 \cdots Y_k)$

   $[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$

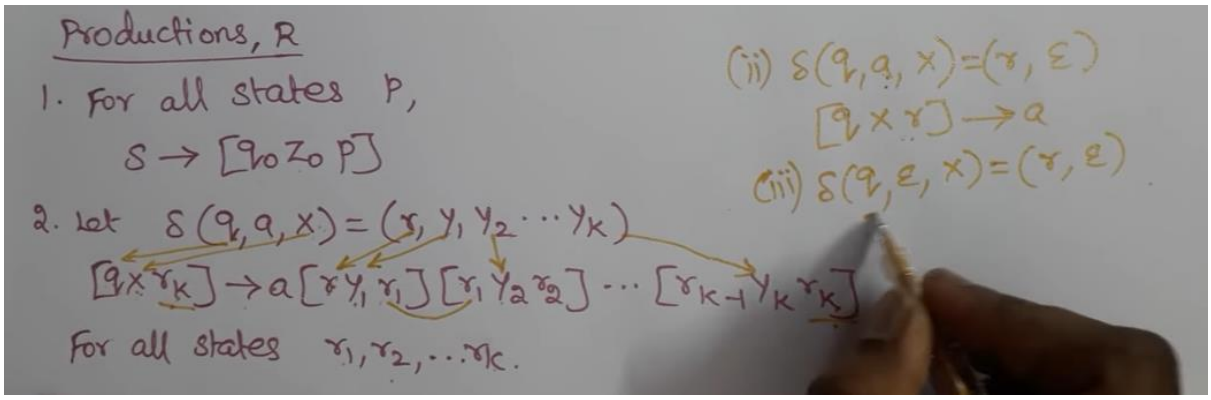
   For all states $r_1, r_2, \ldots$

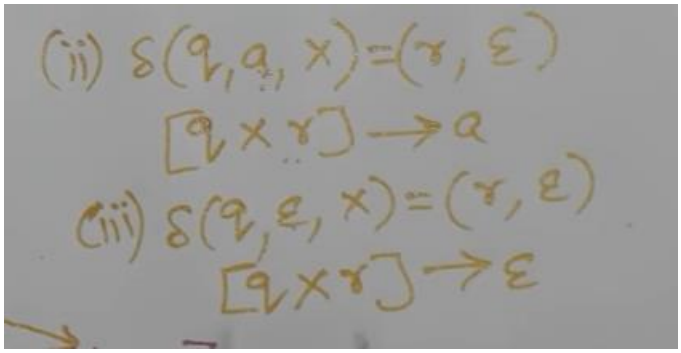
Variables, V

1. special symbol S
2. $[pXq]$ where p, q are states in Q & X is in Γ

Productions, R

1. For all states p,
   $S \rightarrow [q_0 Z_0 p]$
2. Let $\delta(q, a, X) = (r, Y_1 Y_2 \cdots Y_k)$
   $[qX r_k] \rightarrow a[r Y_1 r_1][r_1 Y_2 r_2] \cdots [r_{k-1} Y_k r_k]$
   For all states $r_1, r_2, \ldots r_k$.





Convert the following PDA into CFG.

$P = (\{q, p\}, \{0, 1\}, \{x, z\}, \delta, q, z)$

$\delta(q, 1, z) = (q, xz)$	$\delta(q, 0, x) = (p, x)$
$\delta(q, 1, x) = (q, xx)$	$\delta(p, 1, x) = (p, \varepsilon)$
$\delta(q, \varepsilon, x) = (q, \varepsilon)$	$\delta(p, 0, z) = (q, z)$

$V = \{S, [qxq], [qxp], [pxq], [pxp], [qzq], [qzp], [pzq], [pzp]\}$

i) $S \rightarrow [qzq]$

$S \rightarrow [qzp]$

ii) $\delta(q, 1, z) = (q, xz)$

$[qzq] \rightarrow 1[qxq][qzq]$

$[qzq] \rightarrow 1[qxp][pzq]$

$[qzp] \rightarrow 1[qxq][qzp]$

$[qzp] \rightarrow 1[qxp][pzp]$

iii) $\delta(q_1, 1, X) = (q_1, XX)$

$[q_1 X q_1] \rightarrow 1[q_1 X q_1][q_1 X q_1]$

$[q_1 X q_1] \rightarrow 1[q_1 X p_1][p_1 X q_1]$

$[q_1 X p_1] \rightarrow 1[q_1 X q_1][q_1 X p_1]$

$[q_1 X p_1] \rightarrow 1[q_1 X p_1][p_1 X p_1]$

iv) $\delta(q_1, \varepsilon, X) = (q_1, \varepsilon)$

$[q_1 X q_1] \rightarrow \varepsilon$

v) $\delta(q_1, 0, X) = (p_1, X)$

$[q_1 X q_1] \rightarrow 0[p_1 X q_1]$

$[q_1 X p_1] \rightarrow 0[p_1 X p_1]$

vi) $\delta(p_1, 1, X) = (p_1, \varepsilon)$

$[p_1 X p_1] \rightarrow 1$

vi) $\delta(P, 1, X) = (P, \varepsilon)$

$[PXP] \rightarrow 1$

vii) $\delta(P, 0, Z) = (q, Z)$

$[PZq] \rightarrow 0[qZq]$

$[PZP] \rightarrow 0[qZP]$

PDA to CFG Conversion

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

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

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

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

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

$\delta(q_1, n, z_0) = (q_1, n)$

(FG = (V T P S)

① $S \rightarrow (q_0, z_0, q_0)$

② $S \rightarrow (q_0, z_0, q_1)$

$\delta(q_0, a, z_0) = (q_0, az_0)$ $\{Q^2 = 4$

$[q_0, z_0, q_0] = a[q_0, a, q_0][q_0, z_0, q_0]$

$[q_0, z_0, q_0] = a[q_0, a, q_1][q_1, z_0, q_0]$

$[q_0, z_0, q_1] = a[q_0, a, q_0][q_0, z_0, q_1]$

$[q_0, z_0, q_1] = a[q_0, a, q_1][q_1, z_0, q_1]$

$\delta(q_0, b, a) = (q_1, a)$ { $2^1 = 2$

$[q_0, a, q_0] = b\,[q_1, a, q_0]$

$[q_0, a, q_1] = b\,[q_1, a, q_1]$

$\delta(q_1, a, a) = (q_1, \epsilon)$ { $2^\epsilon = 0$

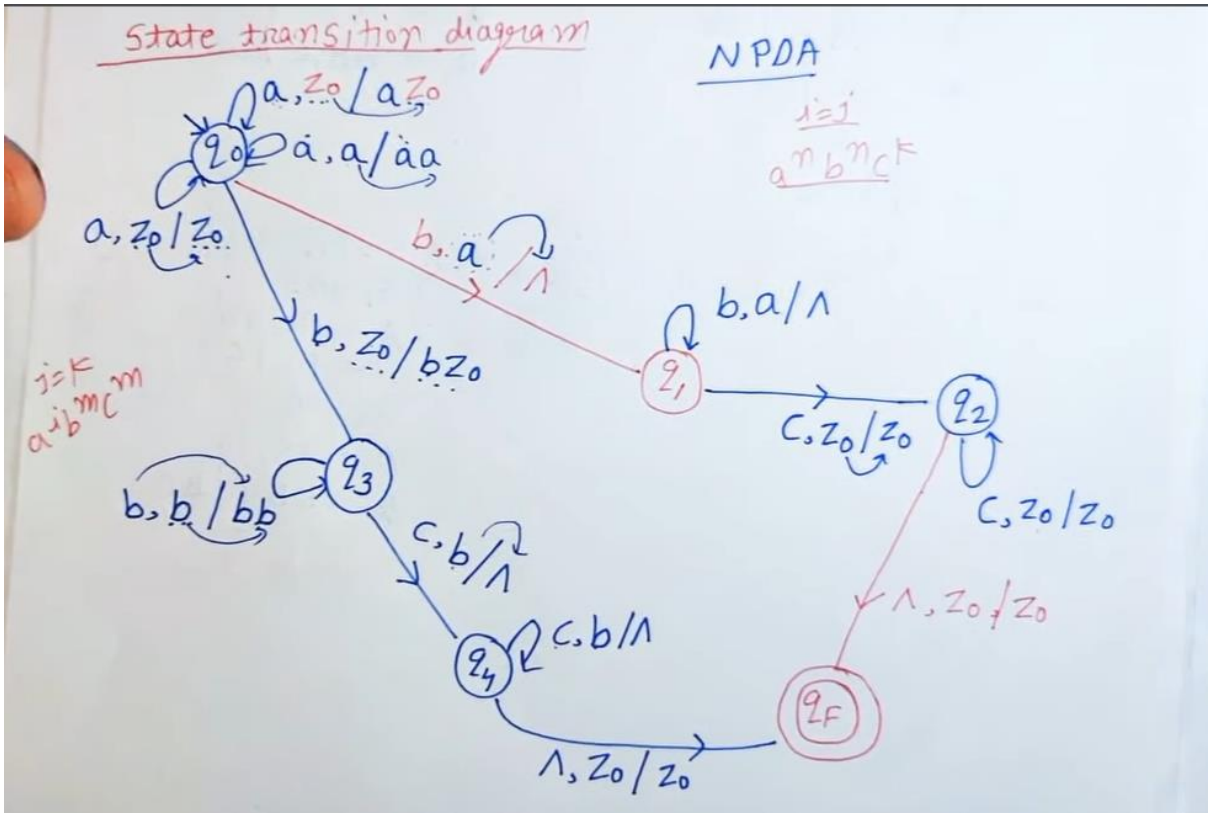
$[q_1, a, q_0] = a$

KV

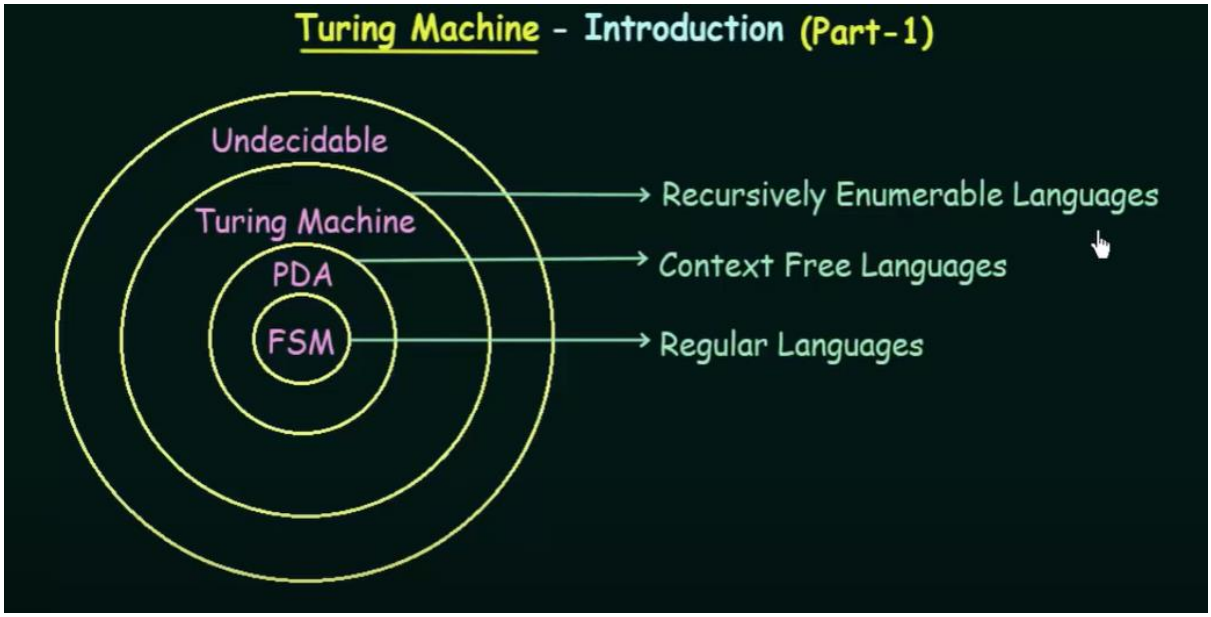
Q => Construct PDA For $L = \{ a^i b^j c^k \mid i=j \text{ or } j=k \}$

Sol => $L = \{ a^i b^j c^k \mid i=j \text{ or } j=k \}$

$i=j=n$	$j=k=m$
$a^n b^n c^k$	$a^i b^m c^m$
Read a - Push a	Read a - NOP
Read b - Pop	Read b - Push b
Read c - NOP	Read c - Pop



Turing Machines



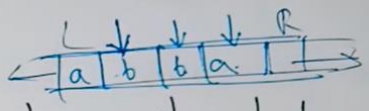
Turing Machine

⇒ Turing M/c has infinite size tape and it is used to accept Recursive Enumerable languages.

- ⇒ TM can move in both directions. Also it doesn't accept ϵ.
- ⇒ If the string inserted in not in lang, m/c will halt in non-final state.

⇒ TM is a mathematical model which consists of an infinite length tape divided into cells on which i/p is given.

⇒ Turing M/c has infinite size tape and it is used to accept Recursive Enumerable languages.



- ⇒ TM can move in both directions. Also it doesn't accept ϵ.
- ⇒ If the string inserted in not in lang, m/c will halt in non-final state.

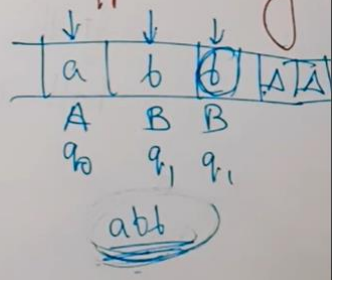
⇒ TM is a mathematical model which consists of an infinite length tape divided into cells on which i/p is given.

⇒ It consists of a head which reads the i/p tape.

⇒ A state reg stores the state of T.M.

⇒ After reading an i/p symbol, it is replaced with another symbol, its internal state is changed & it moves from one cell to the right or left.

⇒ If the TM reaches the final state, the i/p string is accepted, otherwise rejected.



Formal Definition:

A TM can be formally described as 7-tuples $(Q, X, \Sigma, \delta, q_0, B, F)$ where:-

- Q is a finite set of states
- X is the tape alphabet
- Σ is the i/p alphabet
- δ is a transition function;

$$\delta : Q \times X \rightarrow Q \times X \times \{\text{left shift, Right}\}$$

- q_0 is the initial state
- B is the Blank symbol

(B, F) Where :-

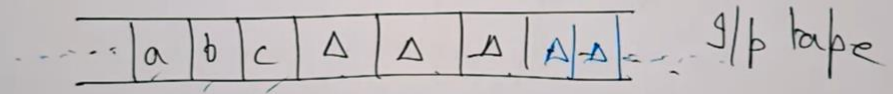
- Q is a finite set of states $\{q_0, q_1, q_2, \ldots, q_n\}$
- X is the tape alphabet
- Σ is the i/p alphabet
- δ is a transition function;

$$\delta : Q \times X \rightarrow Q \times X \times \{\text{left shift, Right}\}$$

- q_0 is the initial state
- B is the Blank symbol Δ ␣
- F is the set of final states

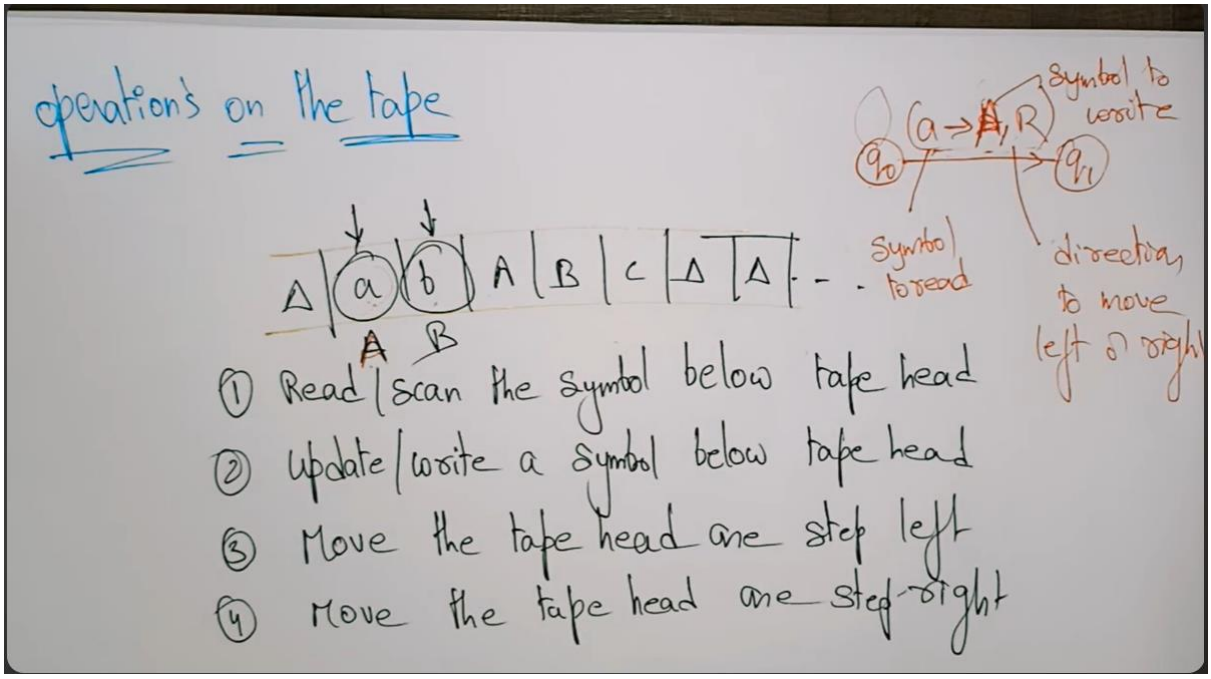
Basic Model of Turing Machine

① The i/p tape is having an infinite no: of cells, Each cell containg one i/p symbol. The Empty tape is filled by blank characters

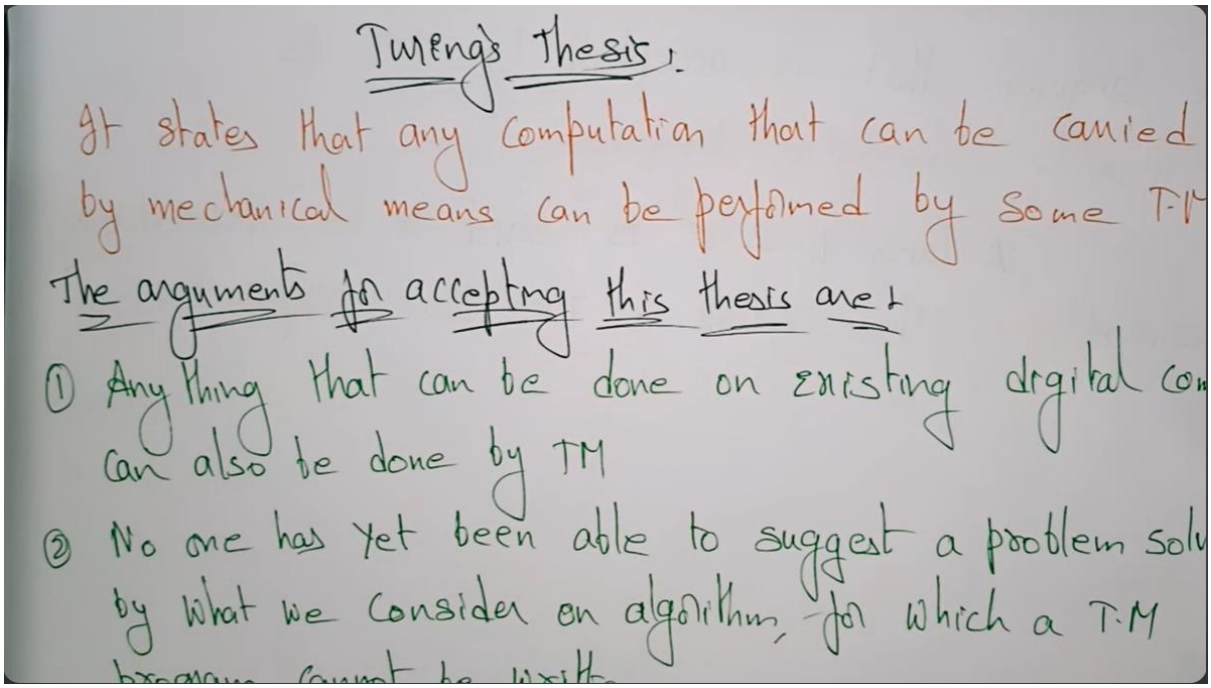


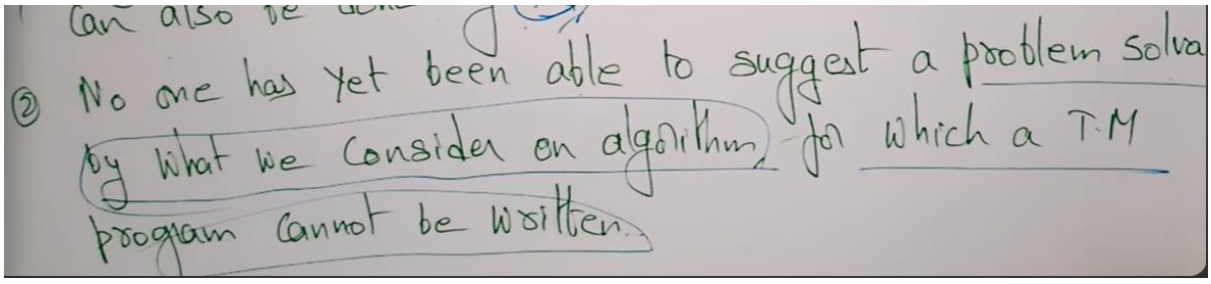
② The Finite control & the tape head which is responsible for reading the current i/p symbol. The tape head can move to left to right.

Operations on tape



Turing Thesis





⇒ The language that is accepted by TM is Recursively Enumerable language.

A lang $L \subseteq \Sigma$ is said to be Recursive Enumerable if there exists a TM. that accepts it.

replacing same set of rules for any no. of time

list of ele

Language accepted by Turing Machine

⇒ The T.M accepts all the lang even though they are recursive Enumerable.

⇒ Recursive means repeating same set of rules for any no: of times.

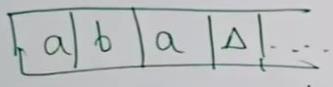
⇒ Enumerable means a list of elements

⇒ TM also accepts the computable functions, such as addition multiplication, subtraction, division, and many more.

Eg: Construct a TM which accepts the lang of aba over

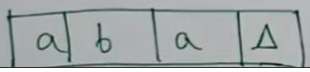
$\Sigma = \{a, b\}$

Sol: We will assume that on i/p tape the string 'aba' is placed

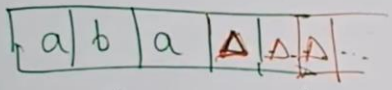


If the tape head is 'read out' 'aba' string then TM will halt after reading Δ.

⇒ Initially, state is q_0 & head points to 'a' as:

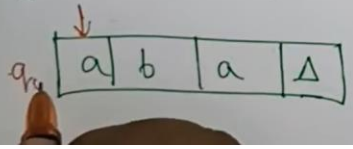


Sol: We will assume that on i/p tape the string 'aba' is placed

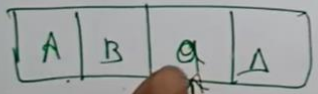


If the tape head is 'read out' 'aba' string then TM will halt after reading Δ.

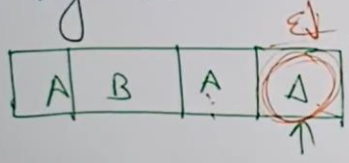
⇒ Initially, state is q_0 & head points to 'a' as:



⇒ The move will be $\delta(q_1, b) = \delta(q_2, B, R)$ which means it will go to state q_2, replaced 'b' by 'B' and head will move to right as:



The move will be $\delta(q_2, a) = \delta(q_3, A, R)$ which means it will go to state q_3, replaced 'a' by 'A' and head will move to right as:



The move $\delta(q_3, \Delta) = (q_4, \Delta, S)$ which means it will go to state q_4 which is HALT state which is accept state for any I.

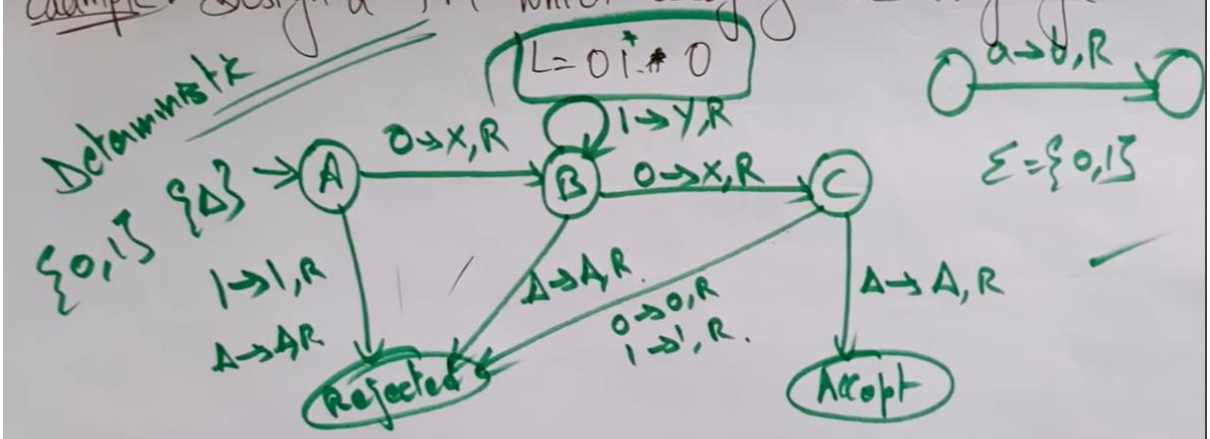
Can be represented by Transition table

States	a	b	Δ
q_0	(q_1, A, R)	-	-
q_1	-	(q_2, B, R)	-
q_2	(q_3, A, R)	-	-
q_3	-	-	(q_4, Δ, S)
q_4	-	-	-

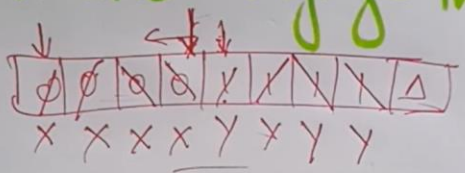
Example: Design a TM which recognize the language

Deterministic

$\{0,1\}$ $\{\Delta\}$



Example:- Design a TM which recognize the lang $L=0^N1^N$

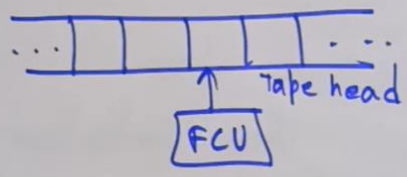


Algorithm:

- change "0" to "x"
- Move Right to first "1"
  - If none: Reject
- change "1" to "y"
- Move left to left most "0"
- Repeat the above steps until no more "0"s

Variants of Turing Machine

⇒ Till now we have seen normal T.M with tape infinite size, Tape head & one finite control unit



⇒ Different Variations of Turing Machines

→ Multitape Turing Machines

→ Non Deterministic Turing Machines

⇒ Different Variations of Turing Machines

→ Multitape Turing Machines

→ Non Deterministic Turing Machines

→ Multihead Turing Machines

→ Off line Turing Machines

→ Multidimensional Turing Machines

① Multitape TM :

→ A T.M with several tapes

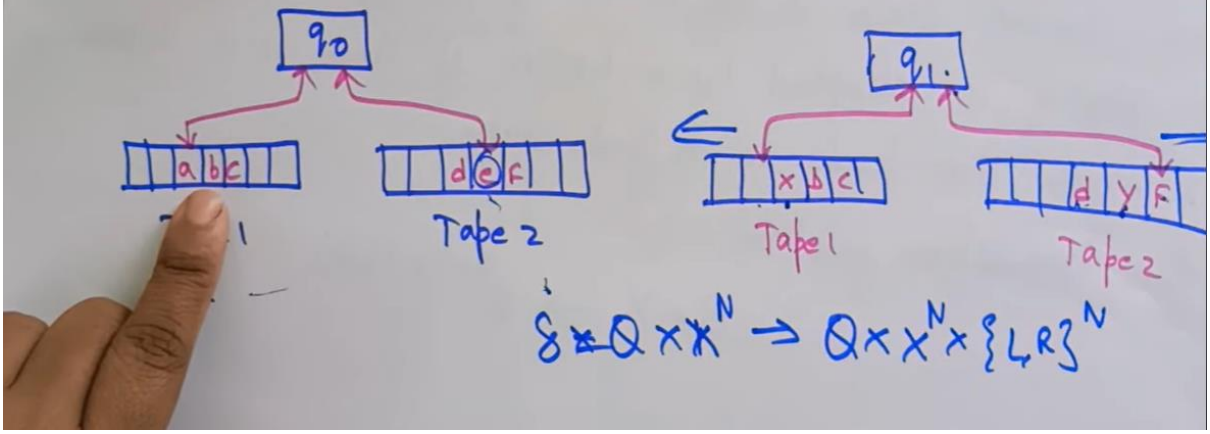
→ Every Tapes have their controlled own R/W Head

→ For N-tape T.M $M = \{Q, \Sigma, X, \delta, q_0, B, F\}$

We define $\delta = Q \times X^N \to Q \times X^N \times \{L, R\}^N$

For eg: if n=2, with current configuration

$\delta(q_0, a, e) = (q_1, x, y, L, R)$ $N=2$



$\delta : Q \times X^N \rightarrow Q \times X^N \times \{L,R\}^N$

② Non Deterministic Turing Machine :-

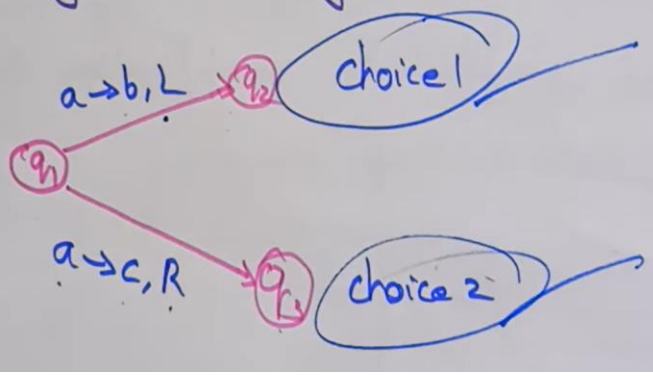
→ It is similar to DTM except that for any i/p sy and current state it has a no: of choices

→ A string is accepted by a NDTM if there is a seq of moves that leads to a final state.

→ The transition function

$\delta : Q \times X \rightarrow 2^{Q \times X \times (L,R)}$

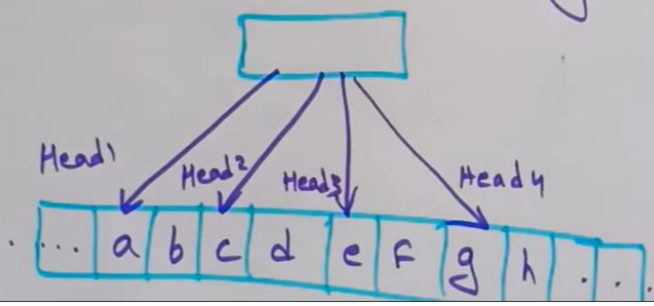
→ A NDTM is allowed to have more than one transition for a given tape symbol



③ Multihead TM :

→ It has no. of heads instead of one.

→ Each head independently read/write symbols and move left/right or keep stationary



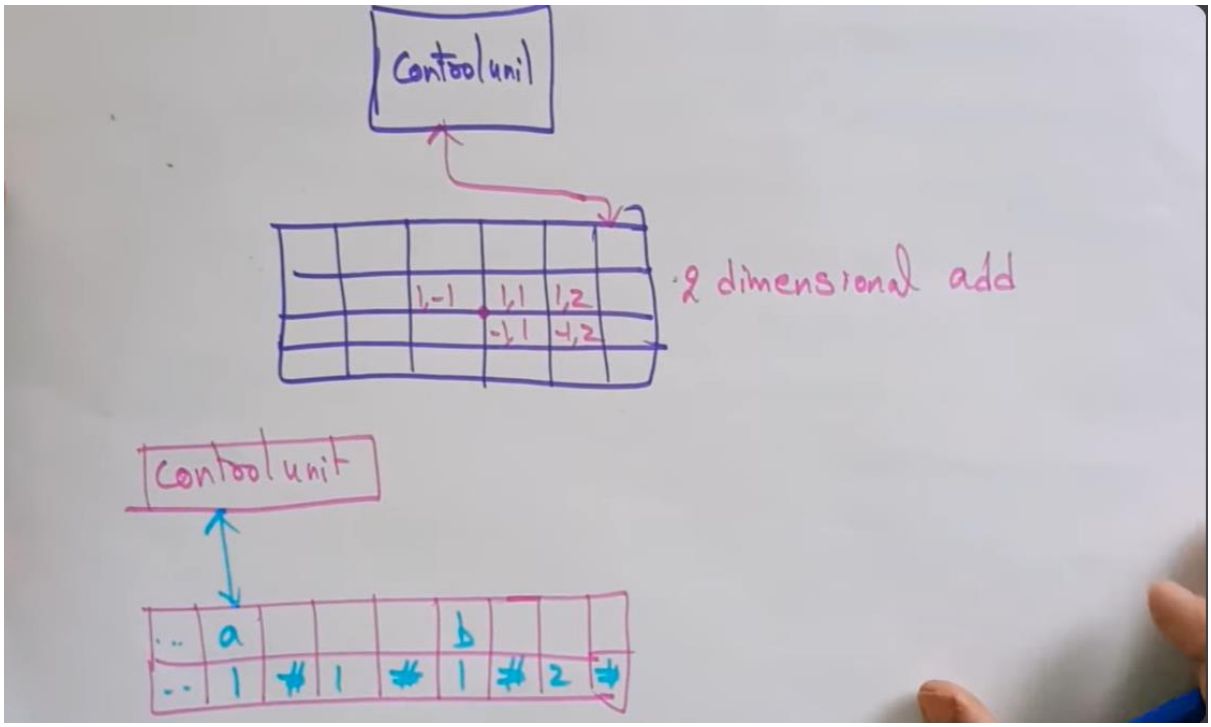
⑤ Multidimensional Turing Machine

→ Multidimensional TM has a multidimensional tape.

for eg:- a two-dimensional T.M would read and write on an infinite tape divided into squares, like a checkboard.

⇒ In a two dimensional TM transaction function defined as:

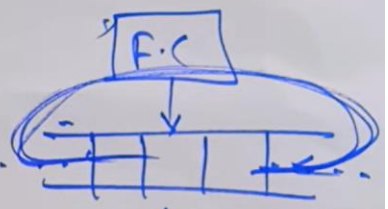
$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\}$$



Recursive & Recursively Enumerable languages

→ The T.M may

- ✱ Halt & accept the input
- ✱ Halt & reject the input, or
- ✱ Never halt / loop



"abc" ⇒ accept & Halt

↓

Halt & reject

Recursively Enumerable lang :-

There is a T.M for a language which accept every string otherwise not.

Recursive language :-

There is a T.M for a language which halt on every string

Recursive lang

Halt on every string

recursive enumerable lang

Accept every string or not

Halting problem

⇒ Halting problem is undecidability. It is not a problem, it ju
asks question :- "is it possible to tell whether a given m/c
will halt for some given i/p"

Eg: input: A T.M & i/p string w

Problem: Does the TM finish computing of the string w in
finite no: of steps ? The answer must be Yes or no

Proof:- Assum T.M exists to solve this problem & then

Eg: input: A T.M & i/p string w

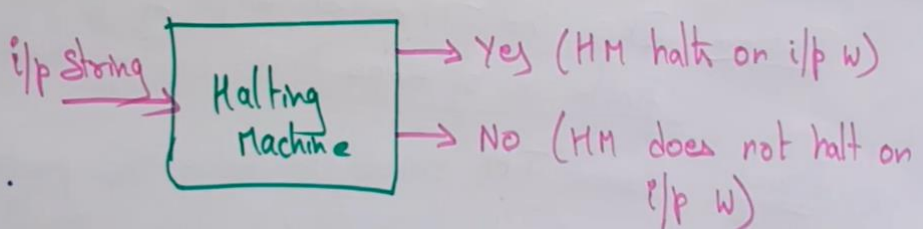
Problem: Does the TM finish computing of the string w in a
finite no: of steps ? The answer must be Yes or no.

Proof:- Assum T.M exists to solve this problem & then
we will show it is contradicting itself.

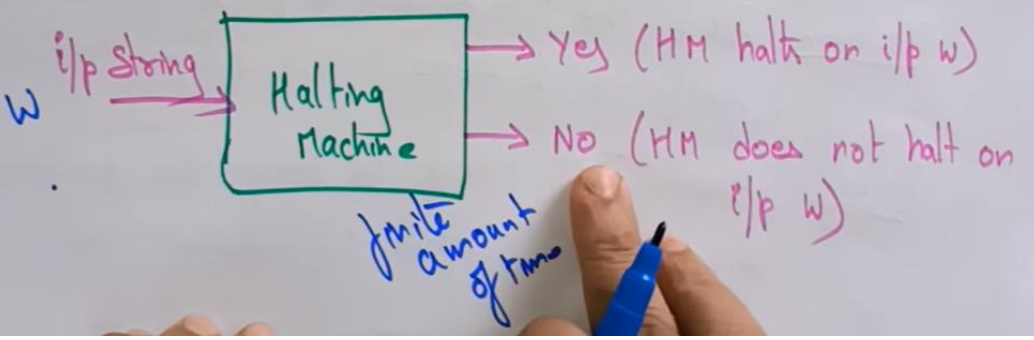
We will call this T.M as a Halting m/c that produce a
'Yes' or 'no' in a finite amount of time.

⇒ If the halting m/c finishes in a finite amount of time,
o/p comes as 'Yes' otherwise as 'no'.

The block dig of Halting m/c :-



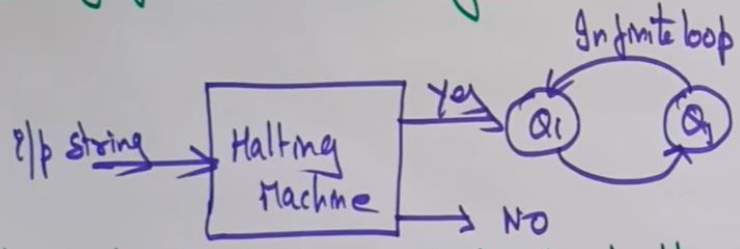
the block dig of Halting m/c :-



Now we will design an inverted halting machine as :-

- If H returns Yes, then loop forever
- If H returns No, then Halt.

The dig for inverted Halting m/c :-



After that a m/c $(HM)_2$ which i/p itself constructed as :-

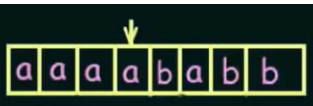
- If $(HM)_2$ halts on i/p, loop forever

After that a m/c $(HM)_2$ which i/p itself constructed as :-

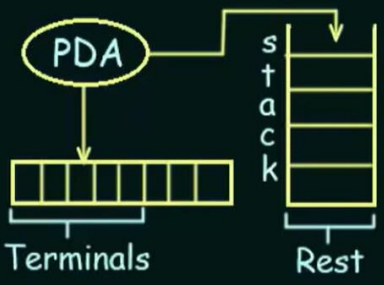
- If $(HM)_2$ halts on i/p, loop forever
- Else, halt.

After this we got a contradiction then the halting prob is undecidable

FSM: The Input String

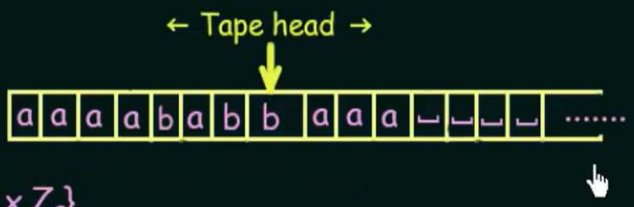


PDA: -> The Input String
-> A Stack



TURING MACHINE:

-> A Tape

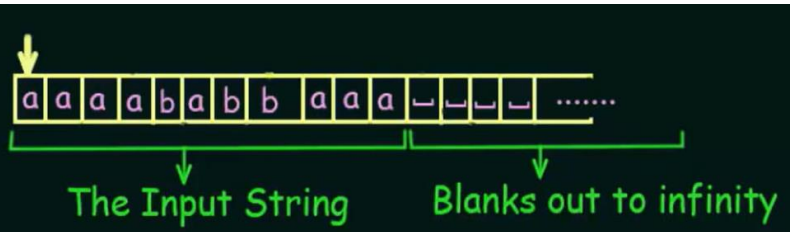


Tape Alphabets: $\Sigma = \{ 0,1,a,b,x,Z_0 \}$

The Blank ⌴ is a special symbol. $\quad ⌴ \notin \Sigma$

The blank is a special symbol used the fill the infinite tape

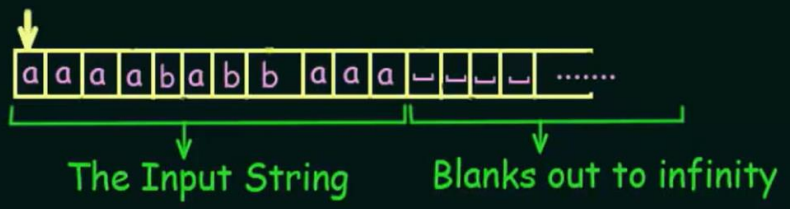
Initial Configuration:



Operations on the Tape:

-> Read /Scan symbol below the Tape Head
-> Update / Write a symbol below the Tape Head
-> Move the Tape Head one step LEFT
-> Move the Tape Head one step RIGHT

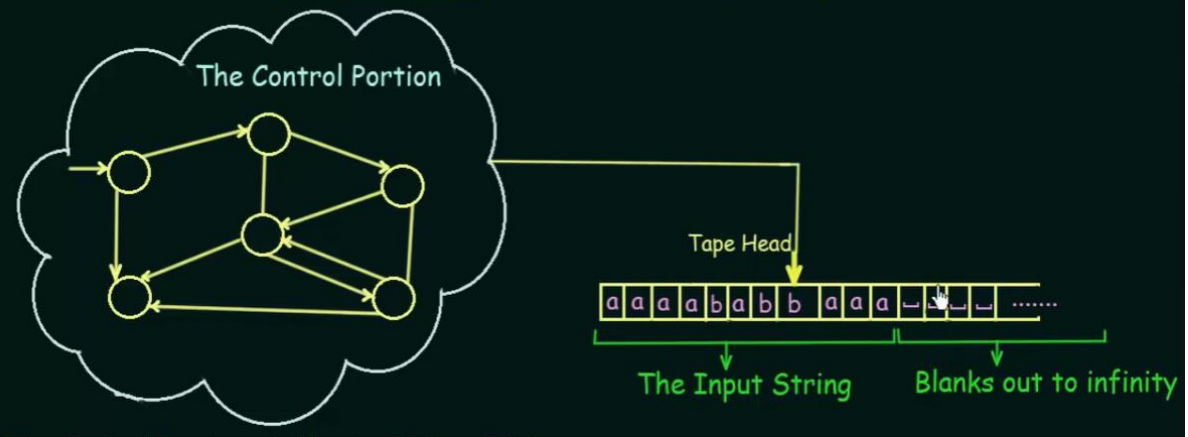
Initial Configuration:



Operations on the Tape:

-> Read /Scan symbol below the Tape Head
-> Update / Write a symbol below the Tape Head
-> Move the Tape Head one step LEFT
-> Move the Tape Head one step RIGHT

Turing Machine - Introduction (Part-2)



The Control Portion similar to FSM or PDA
The PROGRAM
It is deterministic

Rules of Operation - 1

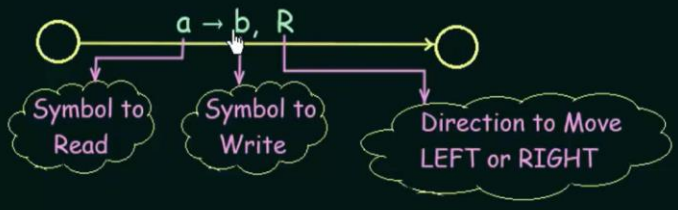
At each step of the computation:

-> Read the currect symbol

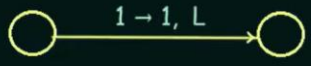
-> Update (i.e. write) the same cell

-> Move exactly one cell either LEFT or RIGHT

If we are at the left end of the tape, and trying to move LEFT, then do not move. Stay at the left end



If you dont 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)

Turing Machine - Example (Part-1)

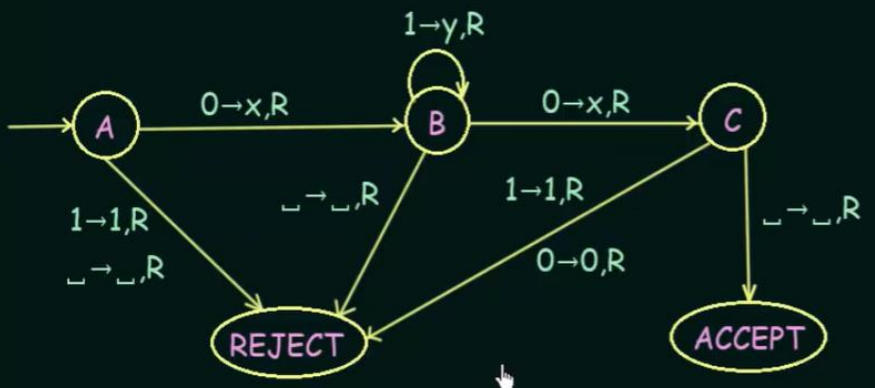
Design a Turing Machine which recognizes the language

L = 01*0

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

b = ␣

0110 ✓

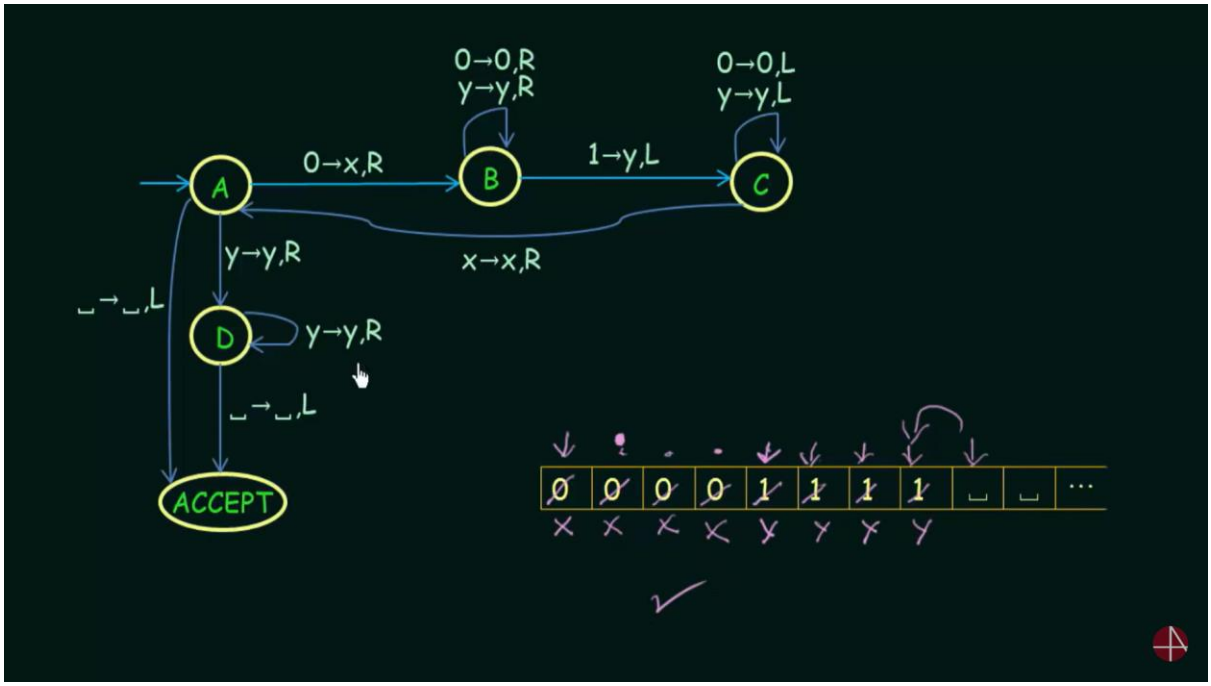


Turing Machine - Example (Part-2)

Design a Turing Machine which recognizes the language $L = 0^N 1^N$

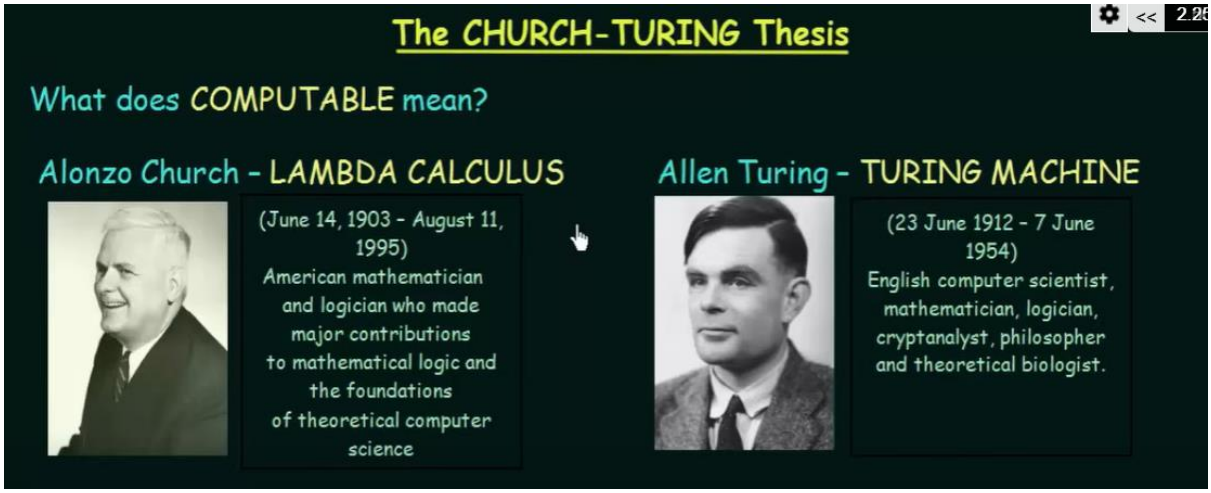
Algorithm:

- Change "0" to "x"
- Move RIGHT to First "1"
  If None: REJECT
- Change "1" to "y"
- Move LEFT to Leftmost "0"
- Repeat the above steps until no more "0"s
- Make sure no more "1"s remain



Church Turing Thesis

https://www.geeksforgeeks.org/churchs-thesis-for-turing-machine/



Several Variations of Turing Machine:

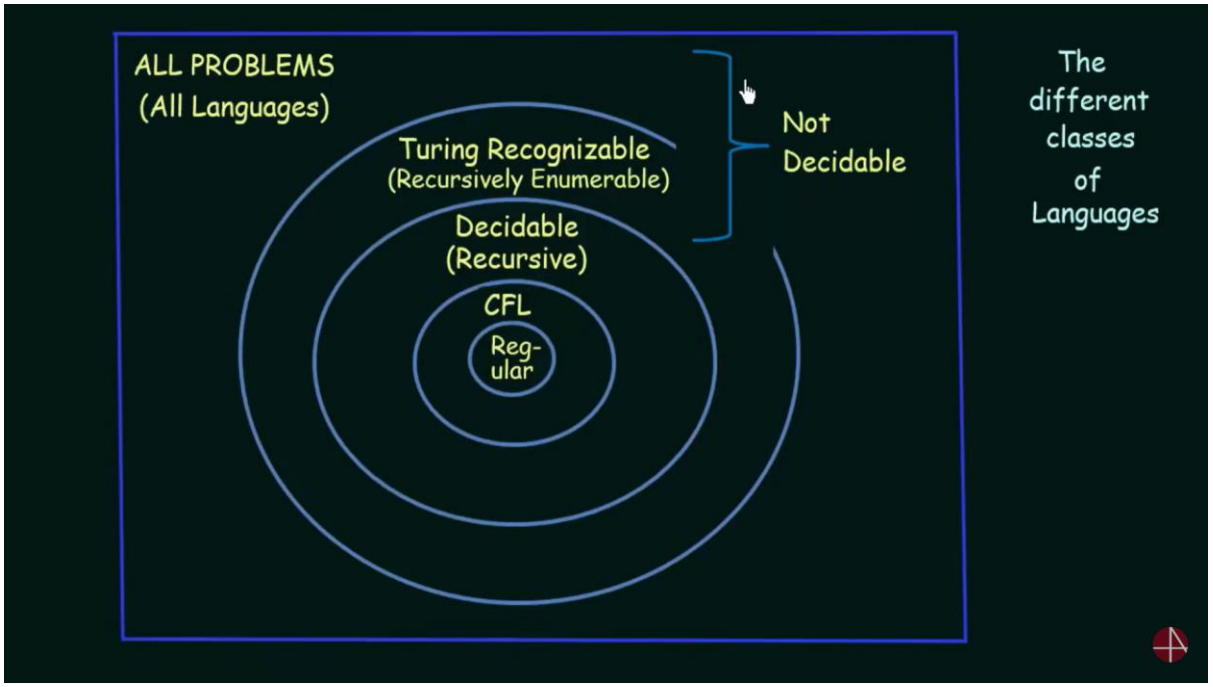
- One Tape or many
- Infinite on both ends
- Alphabets only {0, 1} or more?
- Can the Head also stay in the same place?
- Allow Non-Determinism

All variations are equivalent in computing capability

Turing Machine and Lambda Calculus are also equivalent in power

Algorithmically computable
Means
Computable by a TURING MACHINE

Turing Machine
≠
Turing Test

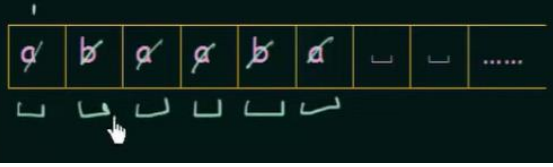


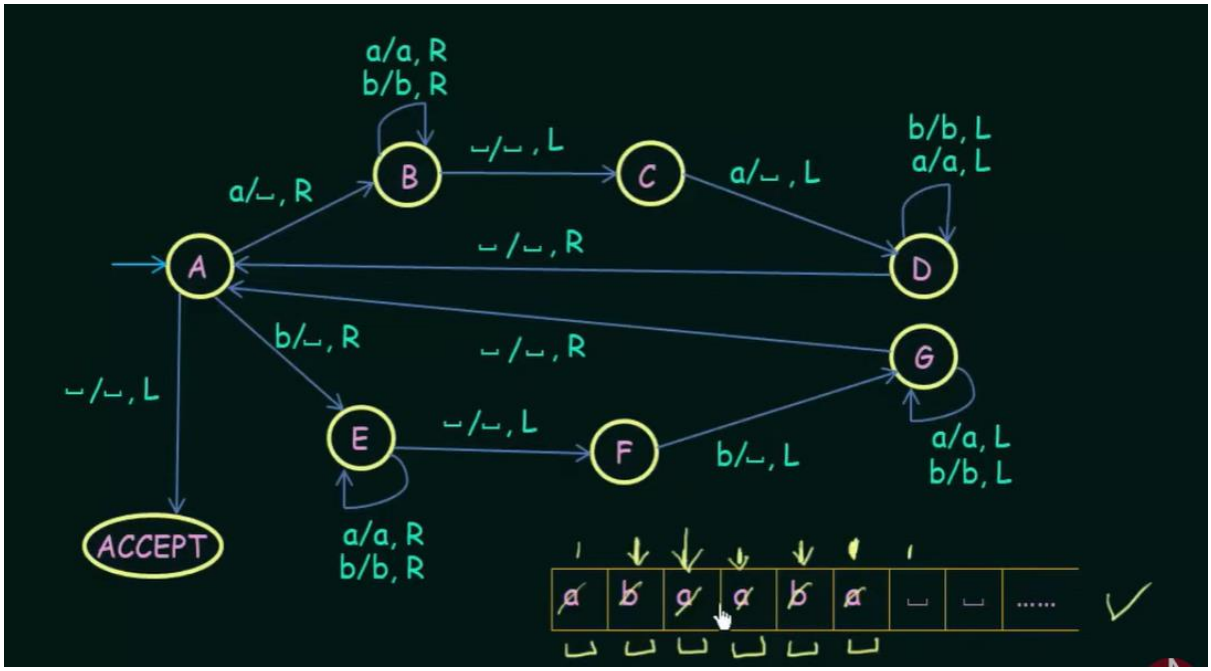
The different classes of Languages

Turing Machine for Even Palindromes

Design a Turing Machine that accepts Even Palindromes over the alphabet
$\Sigma = \{a, b\}$

Eg.

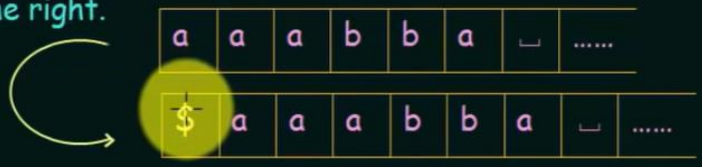


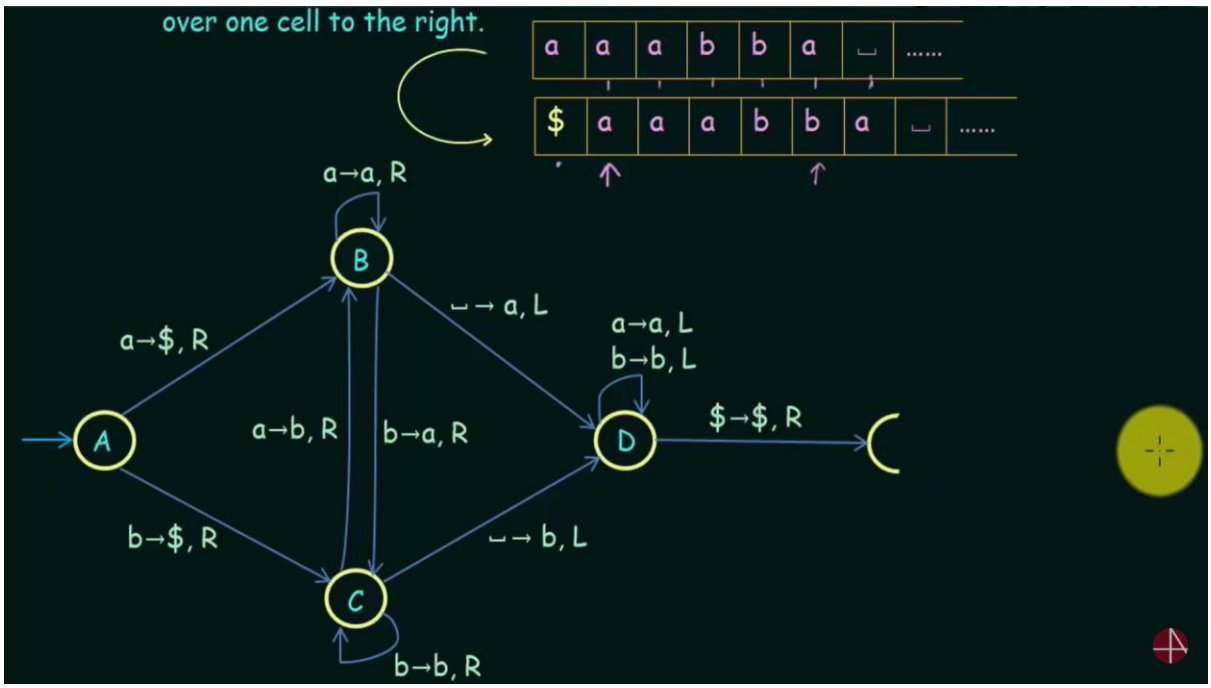


Turing Machine Programming Techniques (Part-1)

Problem: How can we recognize the left end of the Tape of a Turing Machine ?

Solution: Put a Special Symbol $ on the left end of the Tape and shift the input over one cell to the right.





Turing Machine Programming Techniques (Part-2)

Example: Build a Turing Machine to recognize the language $0^N1^N0^N$

IDEA

We already have a Turing Machine to turn 0^N1^N to x^Ny^N and to decide that language.

USE THIS TURING MACHINE AS A SUBROUTINE

Turing Machine Programming Techniques (Part-2)

Example: Build a Turing Machine to recognize the language $0^N1^N0^N$

IDEA

We already have a Turing Machine to turn 0^N1^N to x^Ny^N and to decide that language.

USE THIS TURING MACHINE AS A SUBROUTINE

Step 1: 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0

x x x x x y y y y y 0 0 0 0 0

Step 2: Build a similar Turing Machine to recognize y^N0^N

Step 3: Build the final Turing Machine by combining these two smaller Turing Machines together into one larger Turing Machine

Step 1: 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0

x x x x x y y y y y 0 0 0 0 0

Step 2: Build a similar Turing Machine to recognize y^N0^N

Step 3: Build the final Turing Machine by combining these two smaller Turing Machines together into one larger Turing Machine

Turing Machine Programming Techniques (Part-3)

COMPARING TWO STRINGS

A Turing Machine to decide $\{ w \# w \mid w \in \{a,b,c\}^* \}$

Solution:

- Use a new symbol such as 'x'
- Replace each symbol into an x after it has been examined

Solution:

- Use a new symbol such as 'x'
- Replace each symbol into an x after it has been examined

a b b a c # a b b a c

⇩

x b b a c # x b b a c

⇩

x x b a c # x x b a c

⇩

x x x a c # x x x a c

⇩

x x x x c # x x x x c

⇩

x x x x x # x x x x x

Problem:
Can we do it non-destructively? i.e. without loosing the original strings?

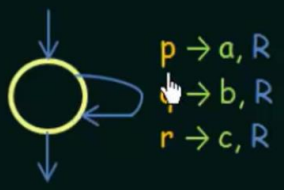
Solution:
Replace each unique symbol with another unique symbol instead of replacing all with the same symbol Eg. a → p
b → q
c → r

a b b a c # a b b a c

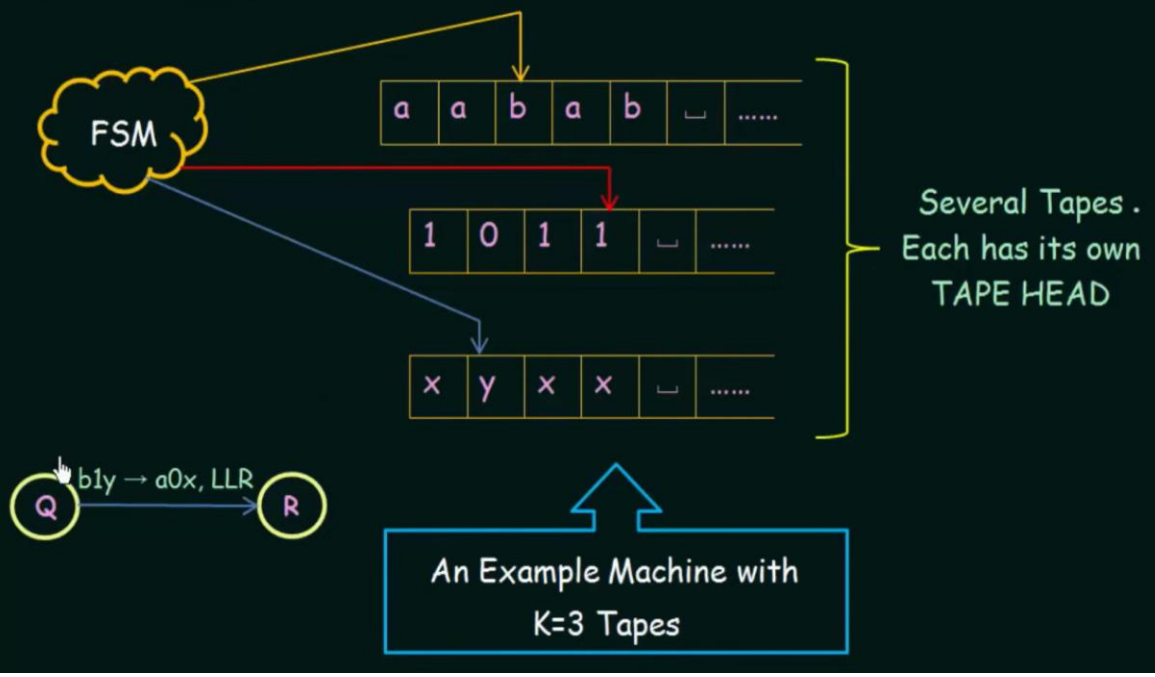
⇩

p q q p r # p q q p r

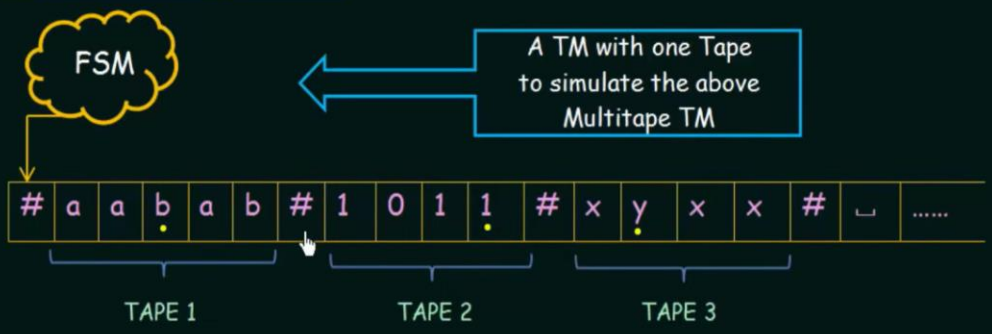
Restore the original strings if required



Multitape Turing Machine



Single Tape Turing Machine



- Add "dots" to show where Head "K" is
- To simulate a transition from state Q, we must scan our Tape to see which symbols are under the K Tape Heads
- Once we determine this and are ready to MAKE the transition, we must scan across the tape again to update the cells and move the dots
- Whenever one head moves off the right end, we must shift our tape so we can insert a ␣

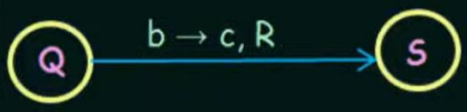
Nondeterminism in Turing Machine (Part-1)

Nondeterministic Turing Machines:

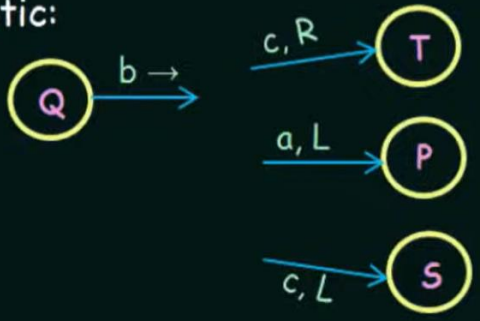
Transition Function:

$\delta : \quad Q \times \Sigma \rightarrow P\{\Gamma \times (R/L) \times Q\}$

Deterministic:

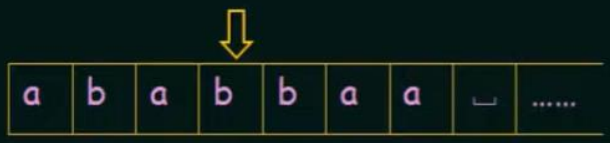


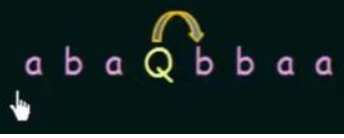
Nondeterministic:



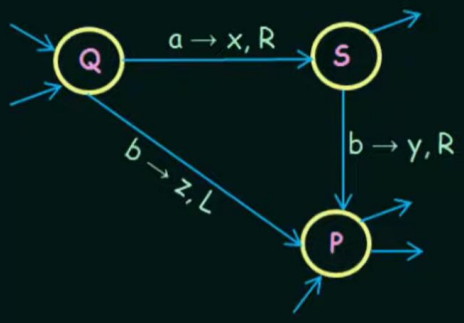
CONFIGURATION

- A way to represent the entire state of a TM at a moment during computation
- A string which captures:
  - The current state
  - The current position of the Head
  - The entire Tape contents

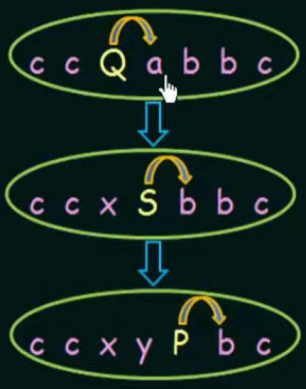




Deterministic TM:

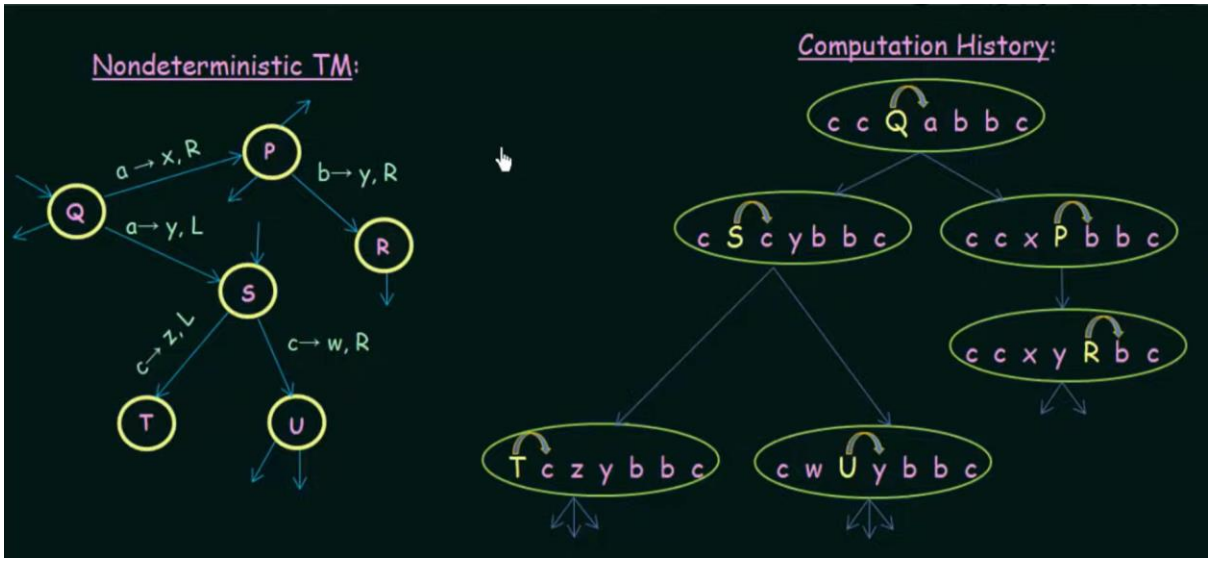


Computation History:



With Nondeterminism:

At each moment in the computation there can be more than one successor configuration



Outcomes of a Nondeterministic Computation:

ACCEPT If any branch of the computation accepts, then the nondeterministic TM will Accept.

REJECT If all branches of the computation HALT and REJECT (i.e. no branches accept, but all computations HALT) then the Nondeterministic TM Rejects.

LOOP Computation continues but ACCEPT is never encountered. Some branches in the computation history are infinite.

Nondeterminism in Turing Machine (Part-2)

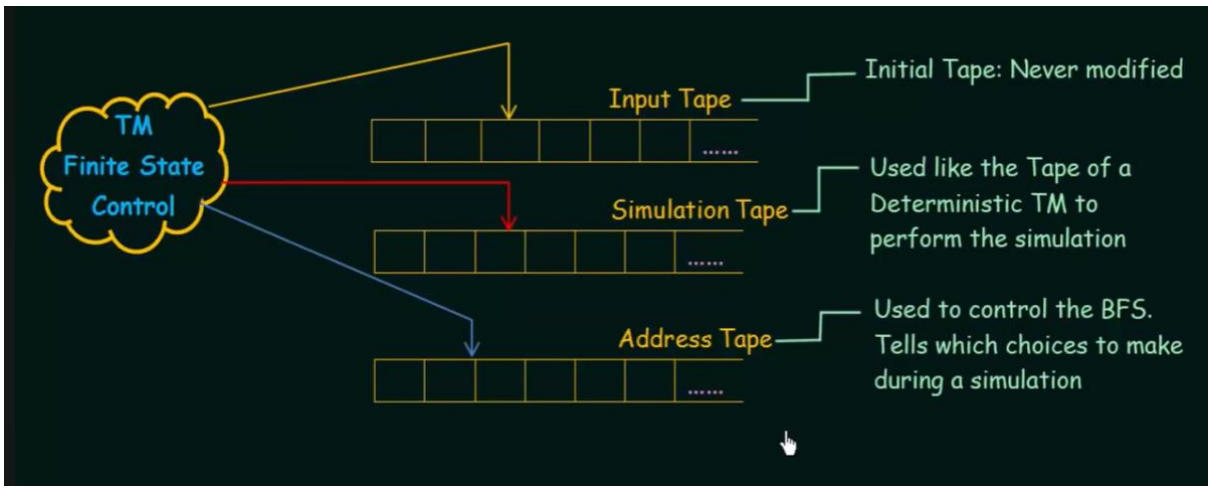
Theorem: Every NondeterministicTM has an equivalent Deterministic TM

Proof:

- Given a Nondeterministic TM (N) show how to construct an equivalent Deterministic TM (D)
- If N accepts on any branch, the D will Accept
- If N halts on every branch without any ACCEPT, then D will Halt and Reject.

Approach:

- Simulate N
- Simulate all branches of computation
- Search for any way N can Accept



Algorithm:

Initially: TAPE 1 contains the Input
TAPE 2 and TAPE 3 are empty

- Copy TAPE 1 to TAPE 2
- Run the Simulation
- Use TAPE 2 as "The Tape"
- When choices occur (i.e. when Nondeterministic branch points are encountered) consult TAPE 3
- TAPE 3 contains a Path. Each number tells which choice to make
- Run the Simulation all the way down the branch as far as the address/ path goes (or the computation dies)
- Try the next branch
- Increment the address on TAPE 3
- REPEAT

If ACCEPT is ever encountered, Halt and Accept
If all branches Reject or die out, then Halt and Reject

Turing machine as problem solvers

Turing Machine as Problem Solvers

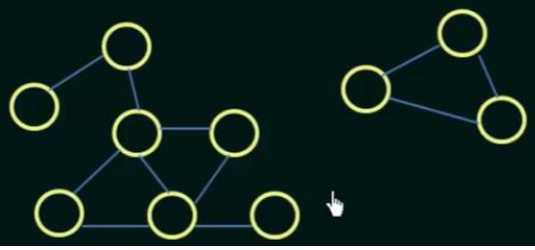
Any arbitrary Problem can be expressed as a language

-Any instance of the problem is encoded into a string

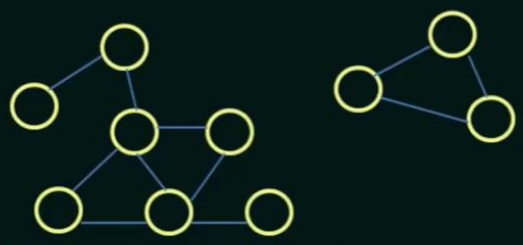
The string is in the language ⇨ The answer is YES

The string is not in the language ⇨ The answer is NO

Example: Is this undirected graph connected?



Example: Is this undirected graph connected?



We must encode the problem into a language.

A = { <G> | G is a connected graph }

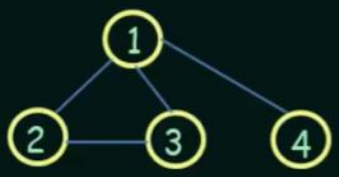
We would like to find a TM to decide this language:

ACCEPT = "YES", This is a connected graph

REJECT = "NO", This is not a connected graph / or this is not a valid Representation of a graph.

LOOP = This problem is decidable. Our TM will always halt

Representation of Graph:



<G> = (1,2,3,4) ((1,2), (2,3), (1,3), (1,4))

List of nodes Edges

Σ = { (,), ,, 1, 2, 3, 4, ……., 0 }

(	1	,	2	,	3	,	4	,	……..	)	)	␣	……

High Level Algorithm:

```
Select a Node and Mark it
REPEAT
    For each node N
        If N is unmarked and there is an edge from N to an already marked
        node
    Then
        Mark Node N
    End

Until no more nodes can be marked
    For each Node N
        If N is unmarked
        Then REJECT
    End
ACCEPT
```

Implementation Level Algorithm:

- Check that input describes a valid graph
- Check Node List
    - Scan "(" followed by digits ...
    - Check that all nodes are different i.e. no repeats
    - Check edge lists ...
        etc.

- Mark First Node
    - Place a dot under the first node in the node list
    - Scan the node list to find a node that is not marked
        etc.

Decidability and Undecidability

Recursive Language:

- A language 'L' is said to be recursive if there exists a Turing machine which will accept all the strings in 'L' and reject all the strings not in 'L'.
- The Turing machine will halt every time and give an answer (accepted or rejected) for each and every string input.

Recursively Enumerable Language:

- A language 'L' is said to be a recursively enumerable language if there exists a Turing machine which will accept (and therefore halt) for all the input strings which are in 'L'.
- But may or may not halt for all input strings which are not in 'L'.

Decidable Language:

A language 'L' is decidable if it is a recursive language. All decidable languages are recursive languages and vice-versa.

Partially Decidable Language:

A language 'L' is partially decidable if 'L' is a recursively enumerable language.

Undecidable Language:

- A language is undecidable if it is not decidable.
- An undecidable language may sometimes be partially decidable but not decidable.
- If a language is not even partially decidable, then there exists no Turing machine for that language

Recursive Language	TM will always Halt
Recursively Enumerable Language	TM will halt sometimes & may not halt sometimes
Decidable Language	Recursive Language
Partially Decidable Language	Recursively Enumerable Language
UNDECIDABLE	No TM for that language

The Universal Turing Machine

The Language

A_{TM} = { <M,w> | M is a Turing Machine and M accepts w}

is Turing Recognizable

Given the description of a TM and some input, can we determine whether the machine accepts it?

- Just Simulate/ Run the TM on the input

M Accepts w: Our Algorithm will Halt & Accept
M Rejects w: Our Algorithm will Halt & Reject.
M Loops on w: Our Algorithm will not Halt.

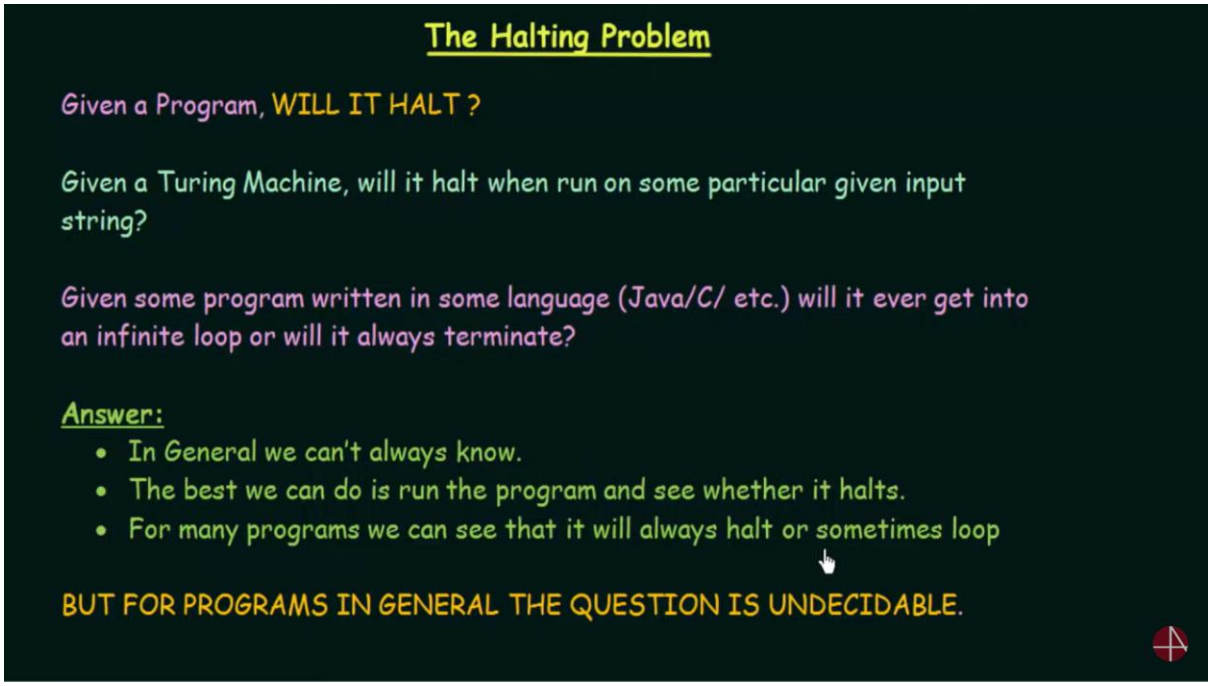
The Universal Turing Machine

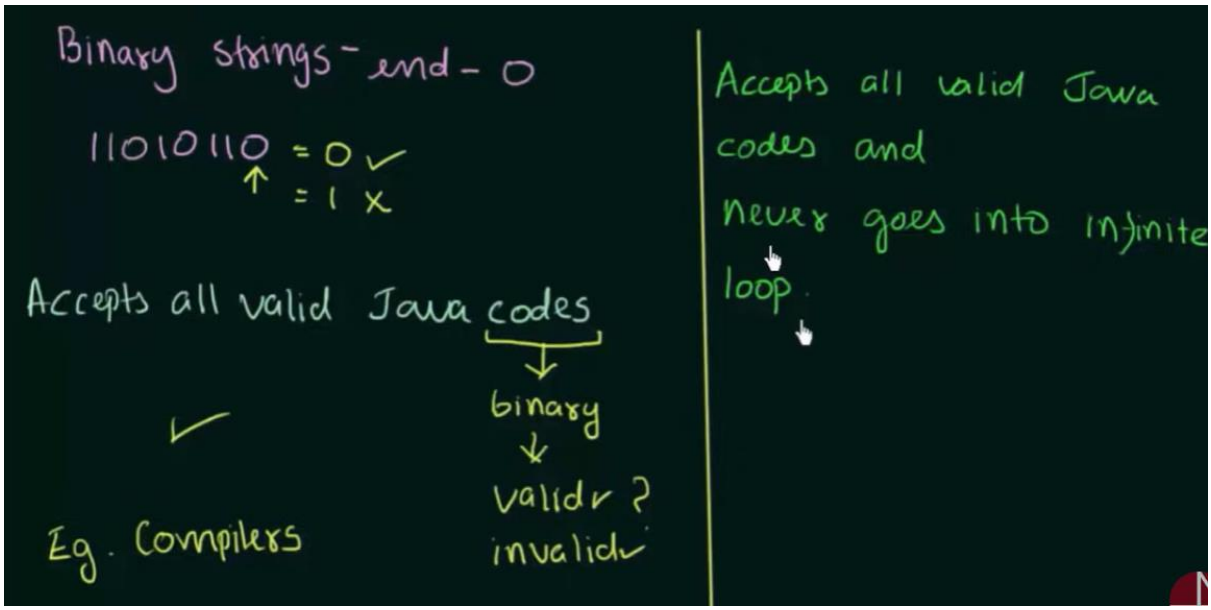
Input: M = the description of some TM
w = an input string for M
Action: - Simulate M
- Behave just like M would (may accept, reject or loop)

The UTM is a recognizer (but not a decider) for

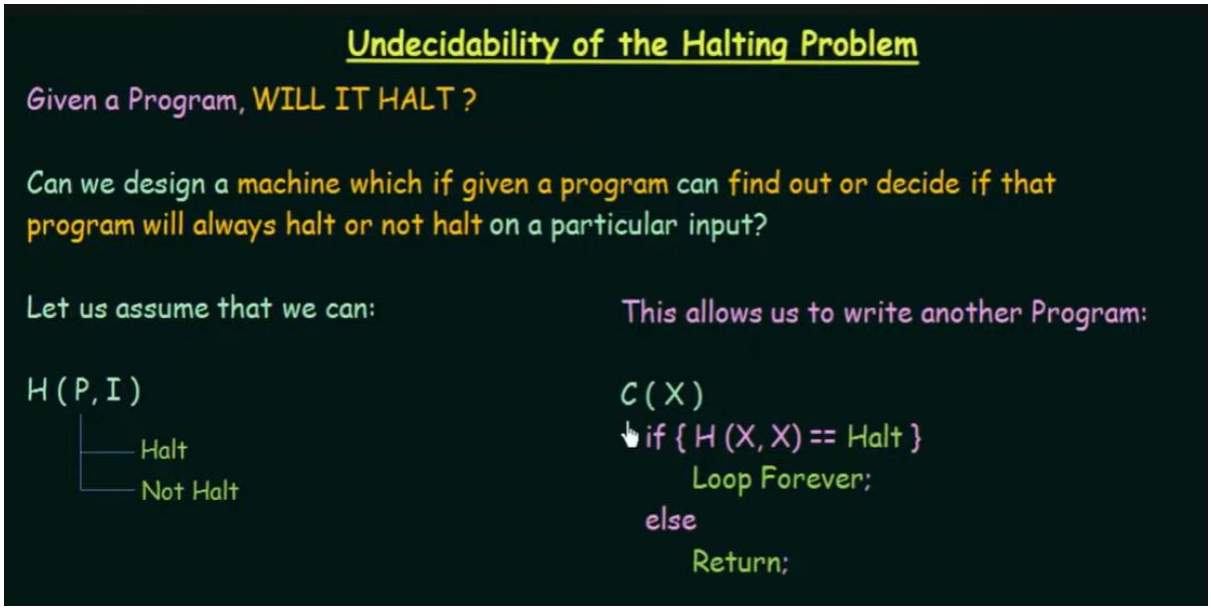
A_{TM} = { <M,w> | M is a TM and M accepts w}

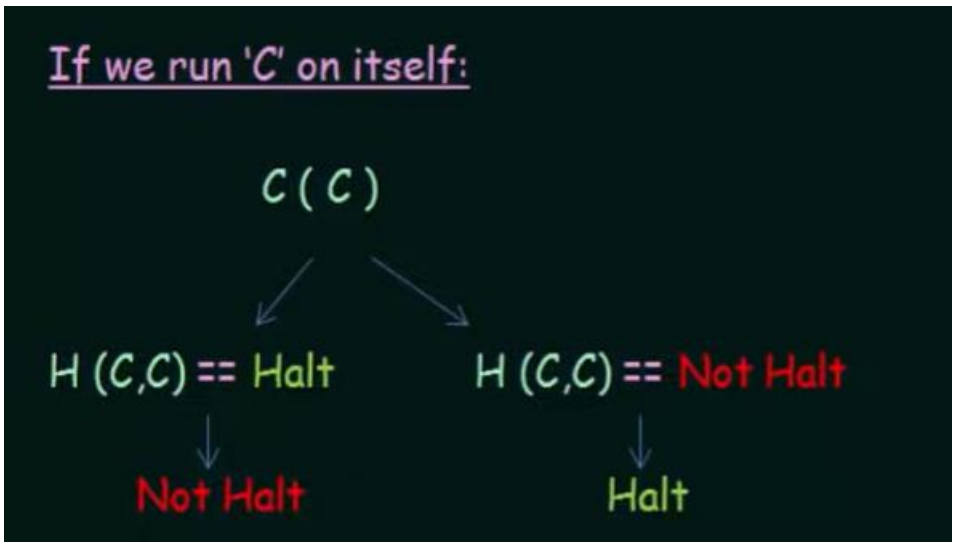
The Halting Problem





Undecidability of the Halting Problem





The Post Correspondence Problem

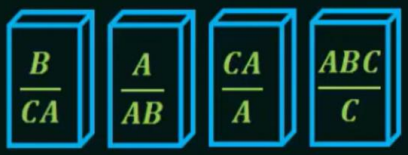
The Post Correspondence Problem

The Post correspondence problem is an undecidable decision problem that was introduced by Emil Post in 1946

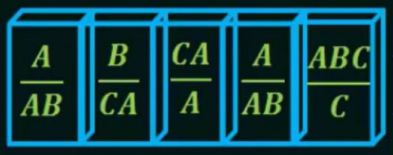
Emil Leon Post

The Post Correspondence Problem

Dominos:

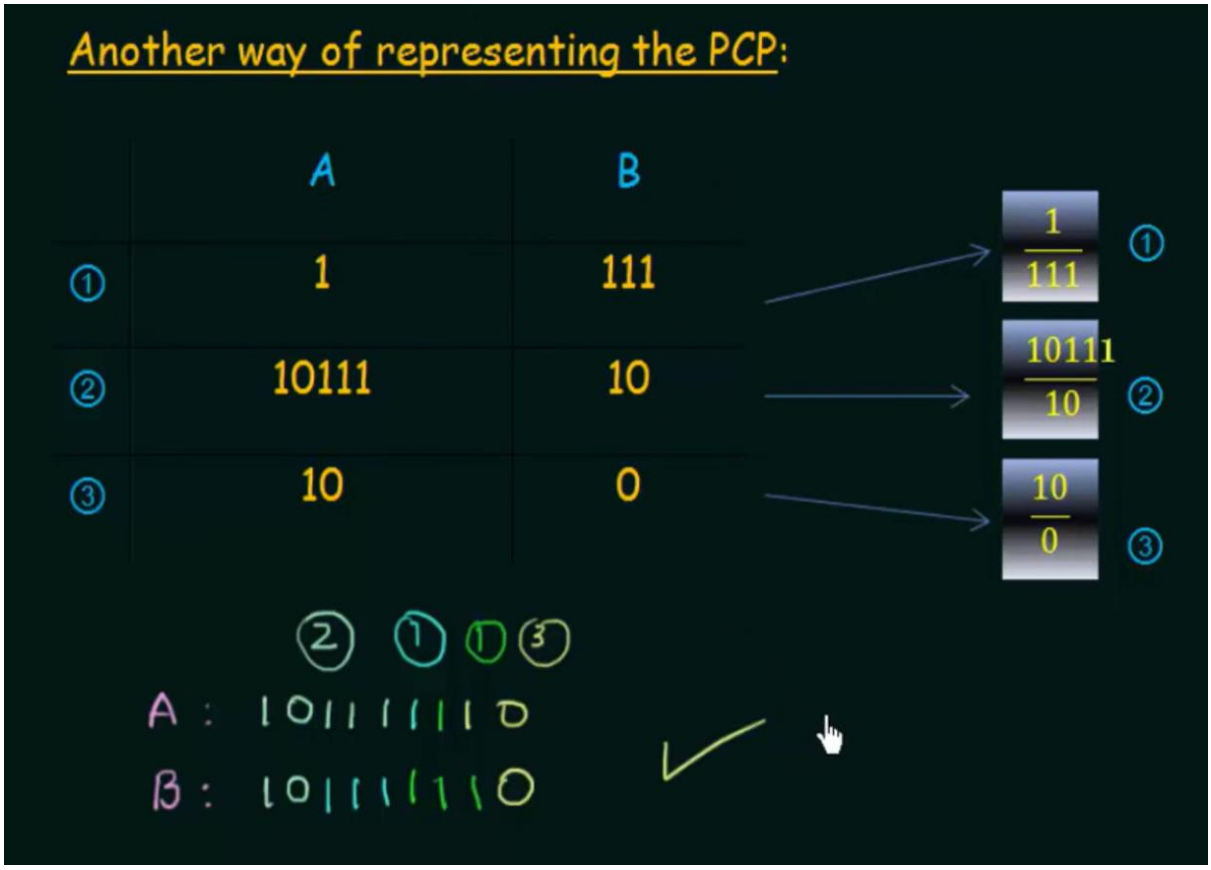


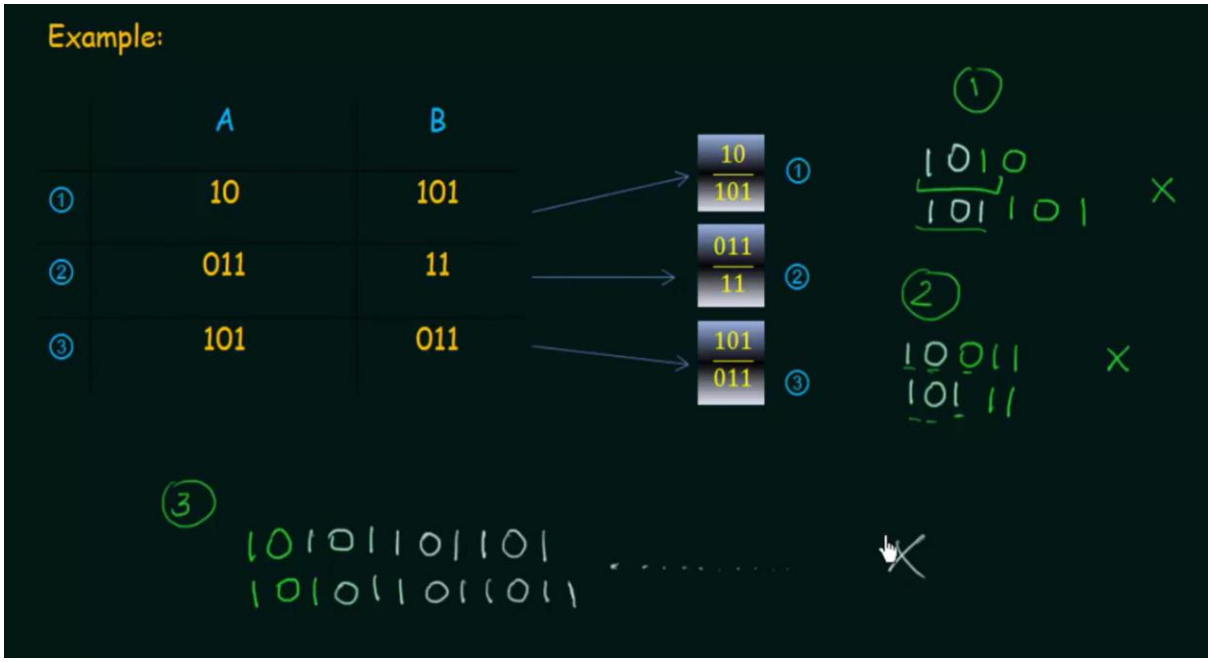
We need to find a sequence of dominos such that the top and bottom strings are the same.



A B C A A A B C

A B C A A A B C





Proof of Undecidability of the Post correspondence problem

Undecidability of the Post Correspondence Problem

PROOF:

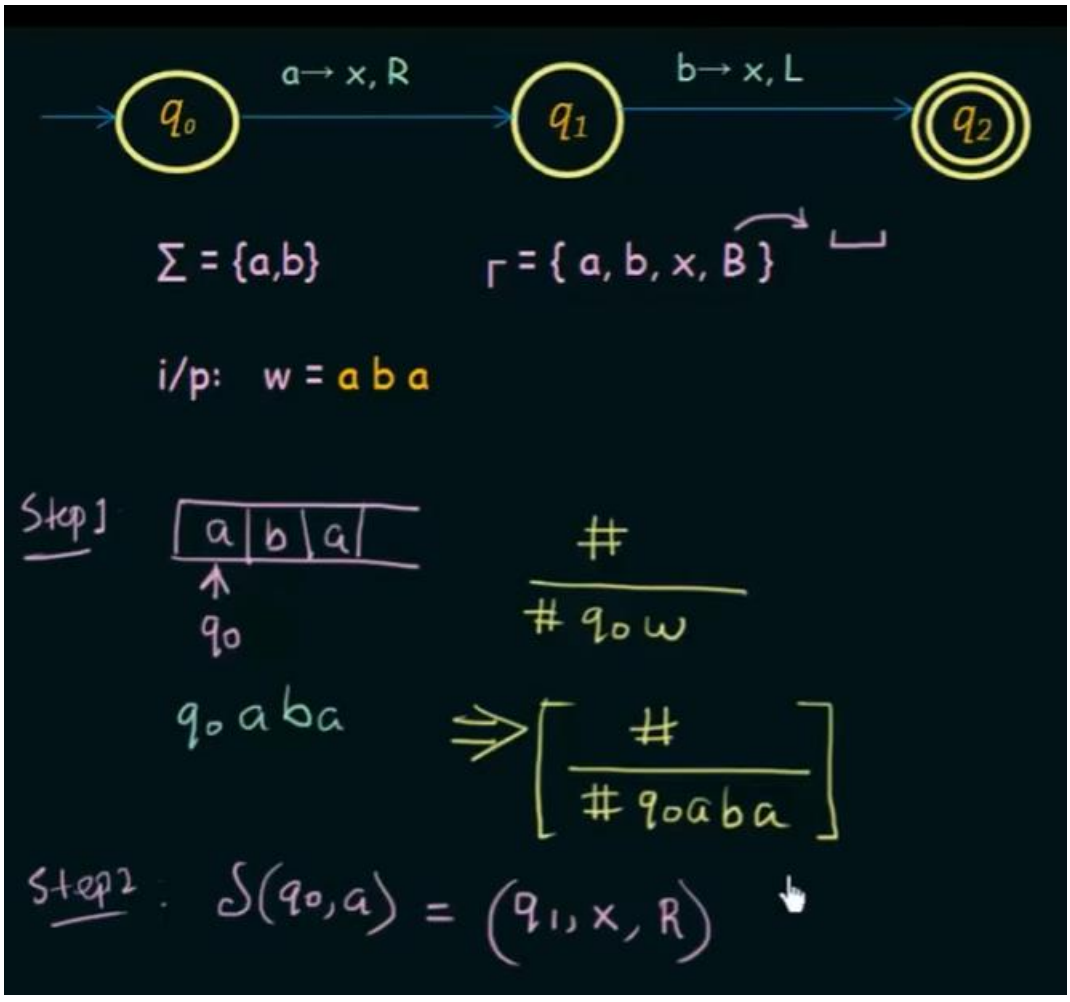
Approach: Take a problem that is already proven to be undecidable and try to convert it to PCP.

If we can successfully convert it to an equivalent PCP then we prove that PCP is also undecidable.

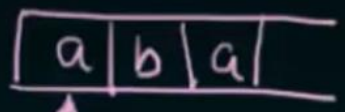
ACCEPTANCE PROBLEM OF A TURING MACHINE *(This is an undecidable probem)*

⇩

Convert this to PCP (called Modified PCP – MPCP)



Step 1:



$$\frac{\#}{\#\, q_0 w \#}$$

$q_0 aba \Rightarrow \left[\frac{\#}{\#\, q_0 aba \#}\right]$

Step 2: $\delta(q_0, a) = (q_1, x, R)$

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

Step 3: $\delta(q_1, b) = (q_2, x, L)$

$Y \in \Gamma$

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

$Y q_1 b = q_2 Y x$

$\left[\frac{a q_1 b}{q_2 a x}\right], \left[\frac{b q_1 b}{q_2 b x}\right], \left[\frac{x q_1 b}{q_2 x x}\right], \left[\frac{B q_1 b}{q_2 B x}\right]$

Step4: For all possible Tape symbols

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

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

Step5: For all possible tape symbols after reaching the accepting state.

Accept (q_2)

$\left[\frac{aq_2}{q_2}\right] \left[\frac{q_2a}{q_2}\right], \left[\frac{bq_2}{q_2}\right], \left[\frac{q_2b}{q_2}\right] \left[\frac{Xq_2}{q_2}\right] \left[\frac{q_2X}{q_2}\right] \left[\frac{Bq_2}{q_2}\right] \left[\frac{q_2B}{q_2}\right]$

Step6: $\left[\frac{\#}{\#}\right] \left[\frac{\#}{B\#}\right]$

Step7: $\left[\frac{q_2\#\#}{\#}\right]$

Solution:

$$\left[\frac{\#}{\#q_0aba\#}\right]\left[\frac{q_0a}{xq_1}\right]\left[\frac{b}{b}\right]\left[\frac{a}{a}\right]\left[\frac{\#}{\#}\right]$$

$$\left[\frac{\#}{\#xq_1ba\#}\right]\left[\frac{xq_1b}{q_2xx}\right]\left[\frac{a}{a}\right]\left[\frac{\#}{\#}\right]$$

$$\left[\frac{\#}{\#q_2xxa\#}\right]\left[\frac{q_2x}{q_2}\right]\left[\frac{x}{x}\right]\left[\frac{a}{a}\right]\left[\frac{\#}{\#}\right]$$

$$\left[\frac{\#}{\#q_2xa\#}\right]\left[\frac{q_2x}{q_2}\right]\left[\frac{a}{a}\right]\left[\frac{\#}{\#}\right]$$

$$\left[\frac{\#}{\#xq_1ba\#}\right]\left[\frac{xq_1b}{q_2xx}\right]\left[\frac{a}{a}\right]\left[\frac{\#}{\#}\right]$$

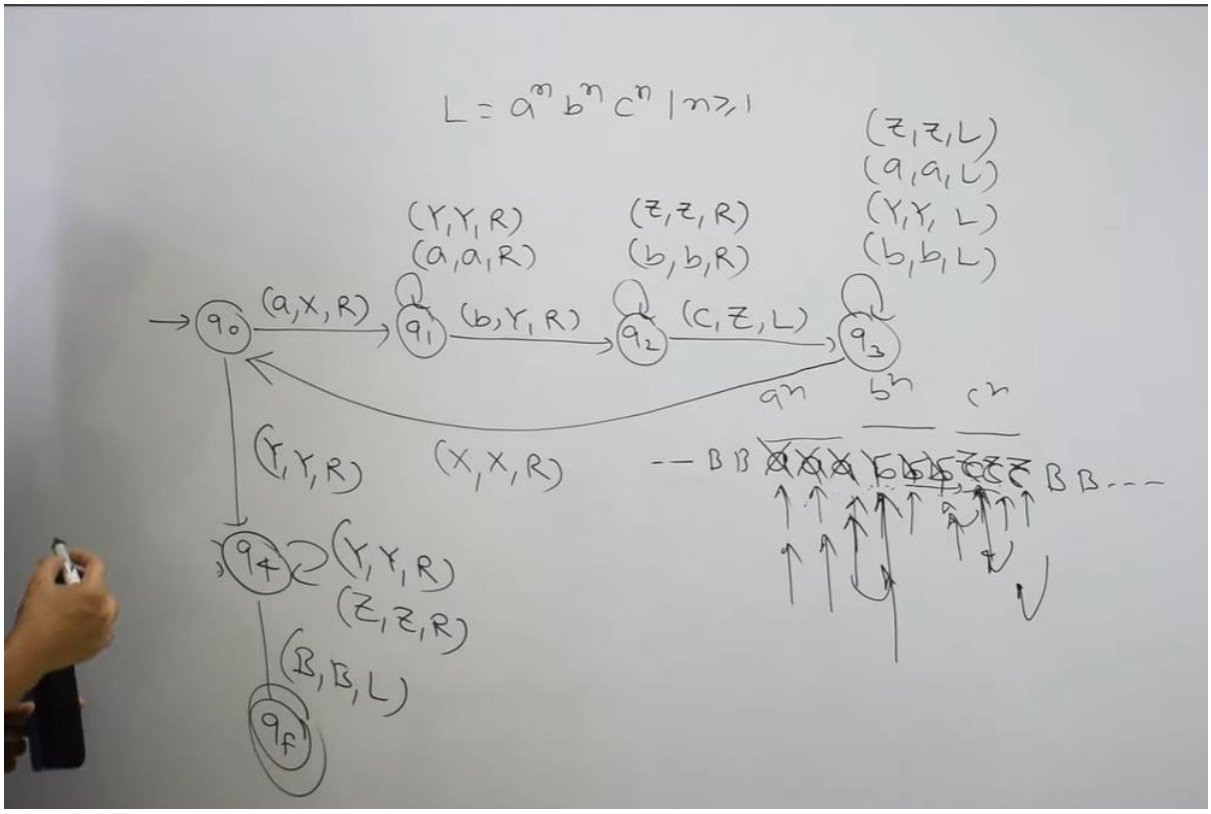
$$\left[\frac{\#}{\#q_2xxa\#}\right]\left[\frac{q_2x}{q_2}\right]\left[\frac{x}{x}\right]\left[\frac{a}{a}\right]\left[\frac{\#}{\#}\right]$$

$$\left[\frac{\#}{\#q_2xa\#}\right]\left[\frac{q_2x}{q_2}\right]\left[\frac{a}{a}\right]\left[\frac{\#}{\#}\right]$$

$$\left[\frac{\#}{\#q_2a\#}\right]\left[\frac{q_2a}{q_2}\right]\left[\frac{\#}{\#}\right]$$

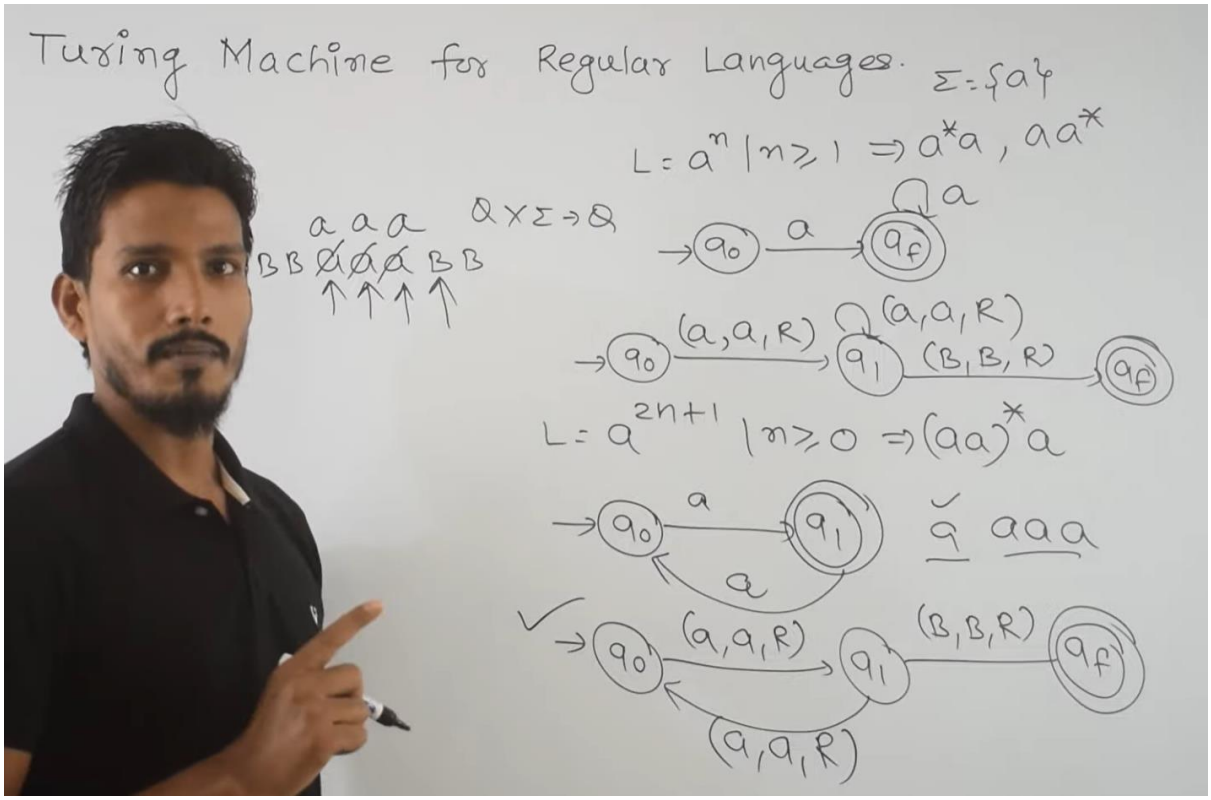
$$\left[\frac{\#}{\#q_2\#}\right]\left[\frac{q_2\#\#}{\#}\right]$$

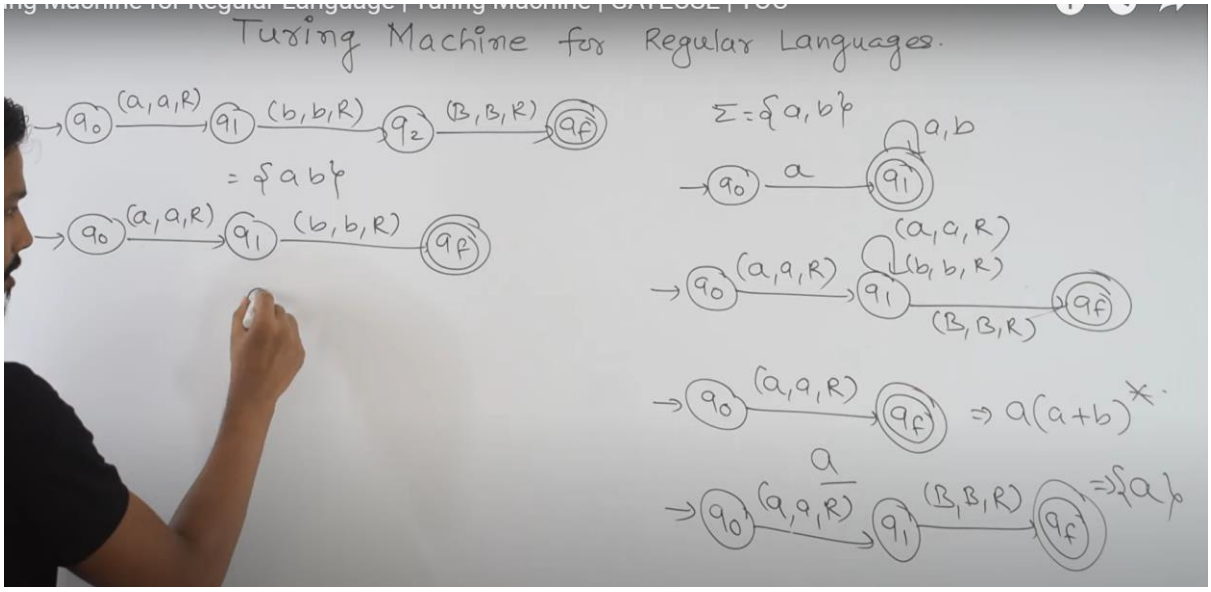
Solved examples

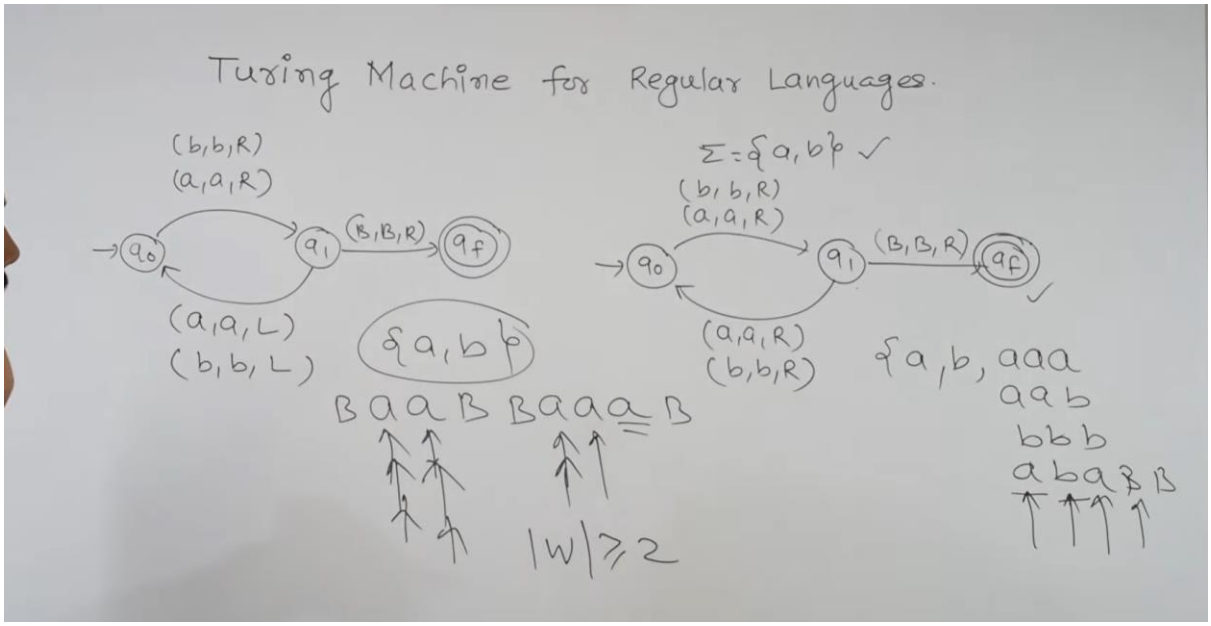


Turing Machine for Regular Language

$L(M) = L$ where M is a TM that moves only to the right side so L is regular

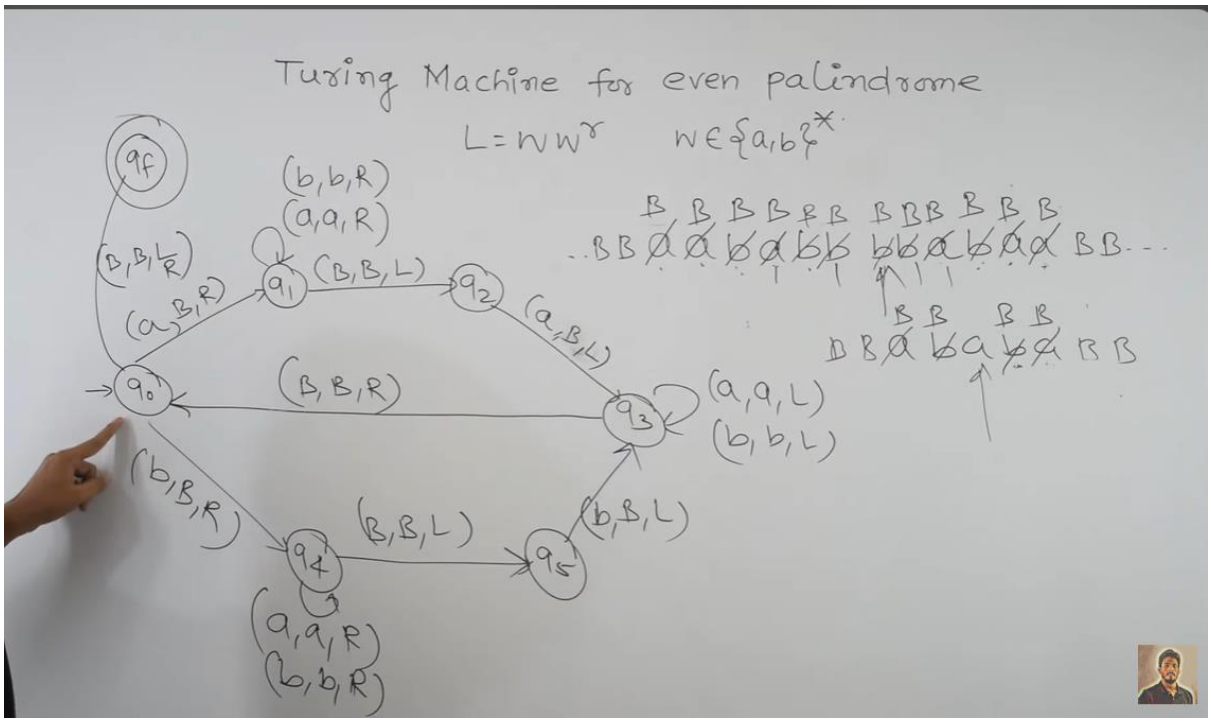

Turing Machine for Regular Languages.
Σ = {a}
L = a^n | n ≥ 1 ⇒ a*a , aa*
a a a
B B a a a B B
Q × Σ → Q
q0 a qf
a
q0 (a,a,R) q1 (a,a,R) (B,B,R) qf
L = a^(2n+1) | n ≥ 0 ⇒ (aa)*a
q0 a q1
a
q aaa
q0 (a,a,R) q1 (B,B,R) qf
(a,a,R)


Turing Machine for Regular Languages.
q0 (a,a,R) q1 (b,b,R) q2 (B,B,R) qf
= {ab}
q0 (a,a,R) q1 (b,b,R) qf
Σ = {a,b}
a,b
q0 a q1
q0 (a,a,R) q1 (a,a,R) (b,b,R) (B,B,R) qf
q0 (a,a,R) qf ⇒ a(a+b)*
a
q0 (a,a,R) q1 (B,B,R) qf ⇒ {a}

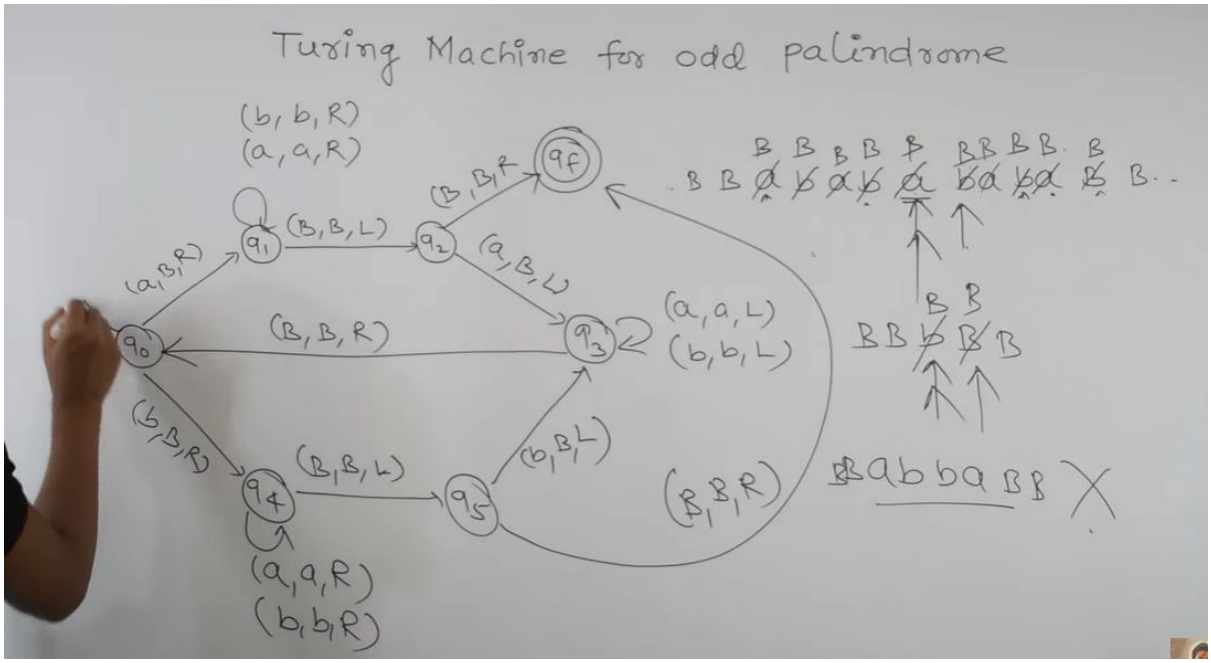


Both the languages are regular in these examples. If we are given a turing machine and

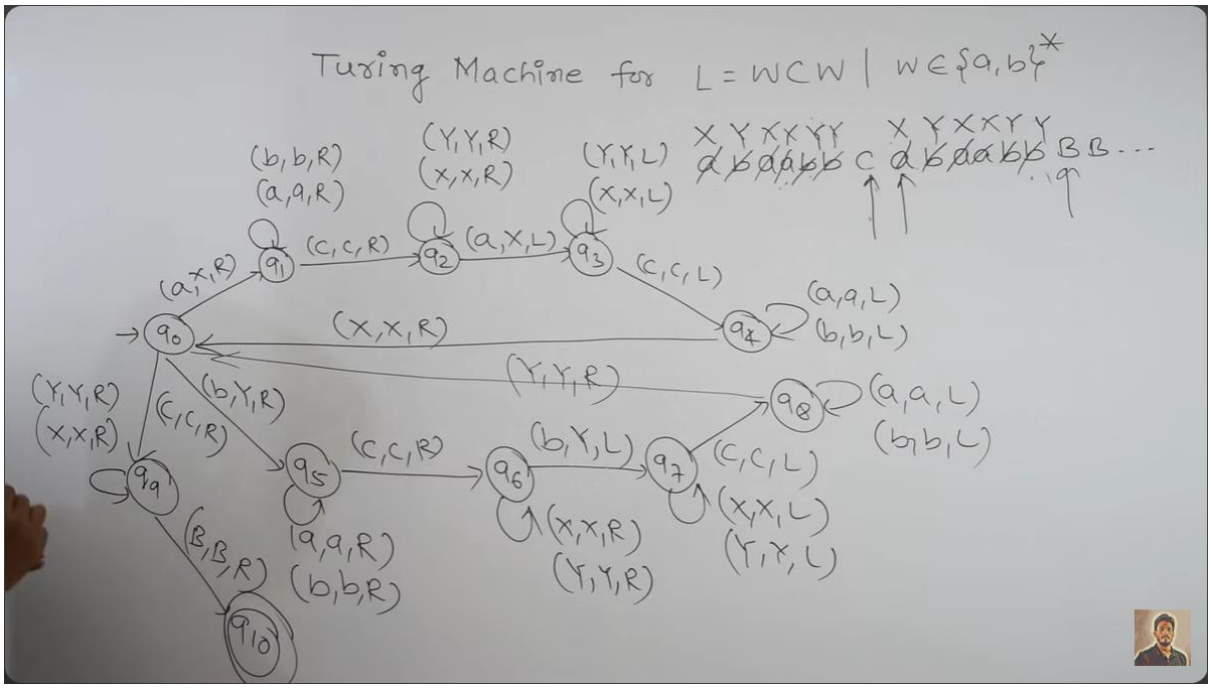
Turing machine for even palindrome



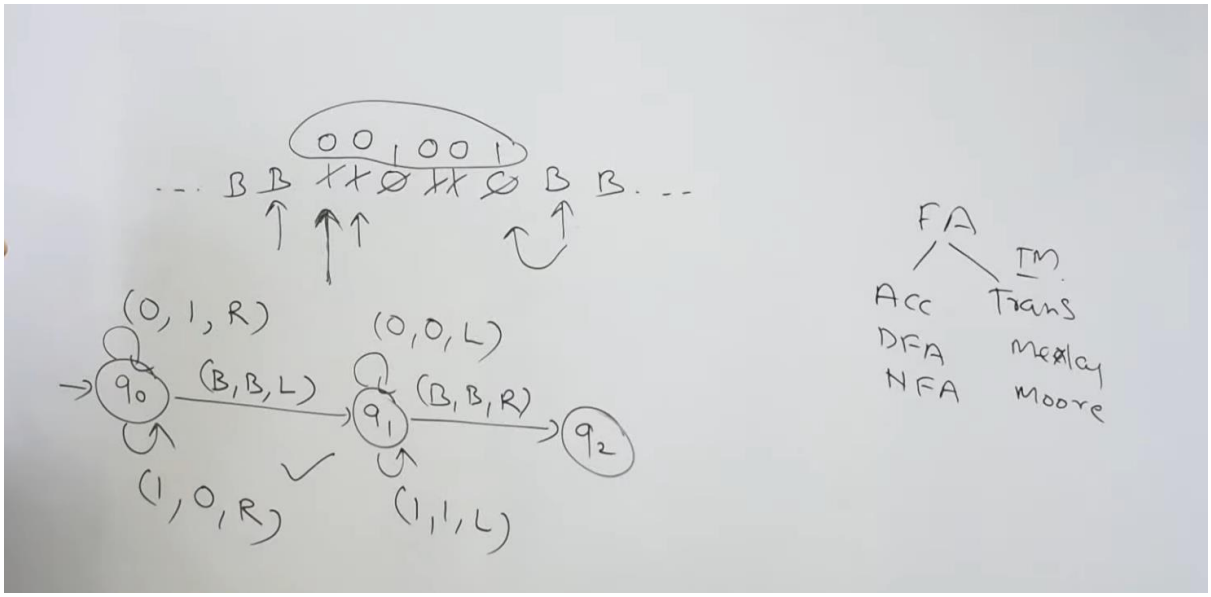
Turing machine for Odd Palindrome



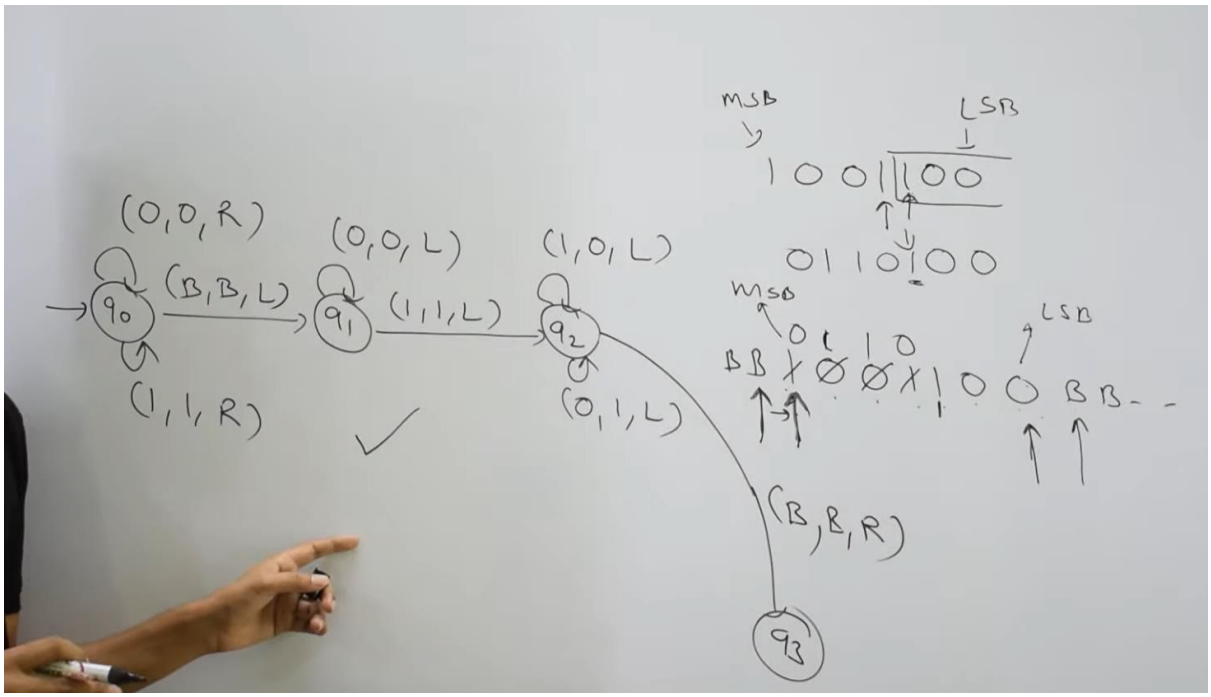
Turing machine for wcw



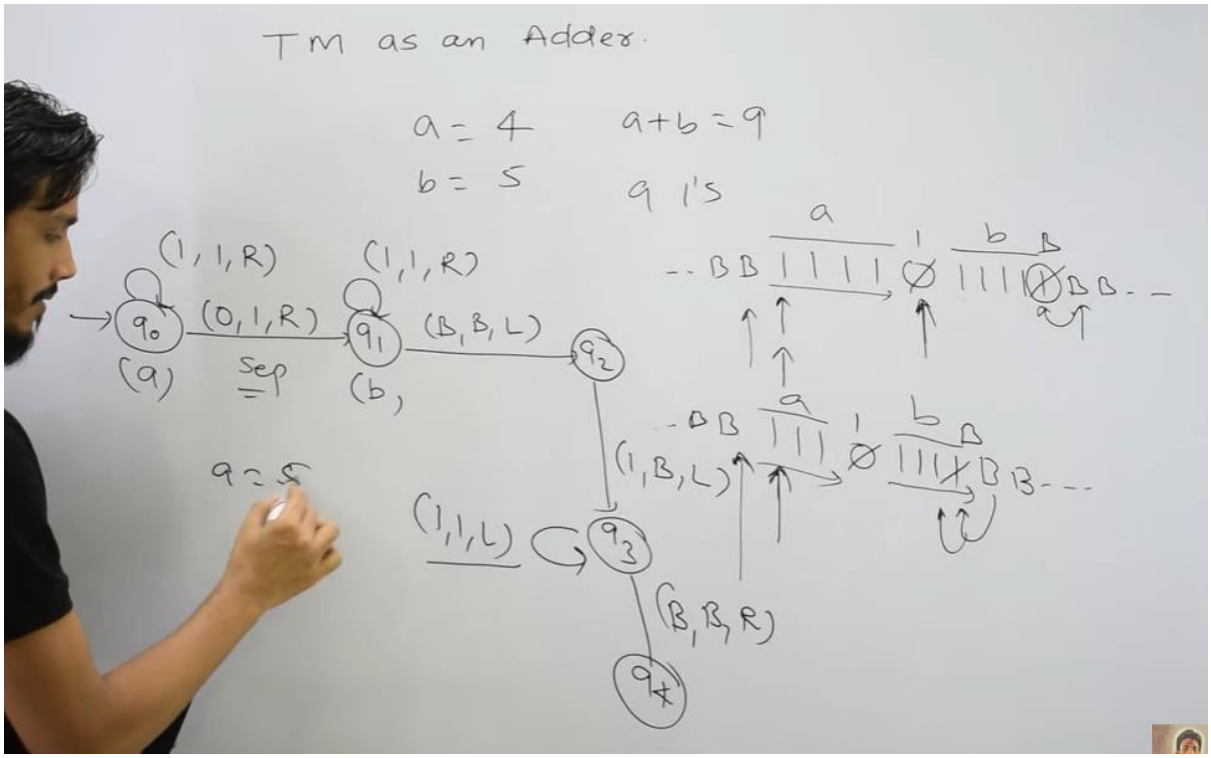
Turing Machine for 1's complement



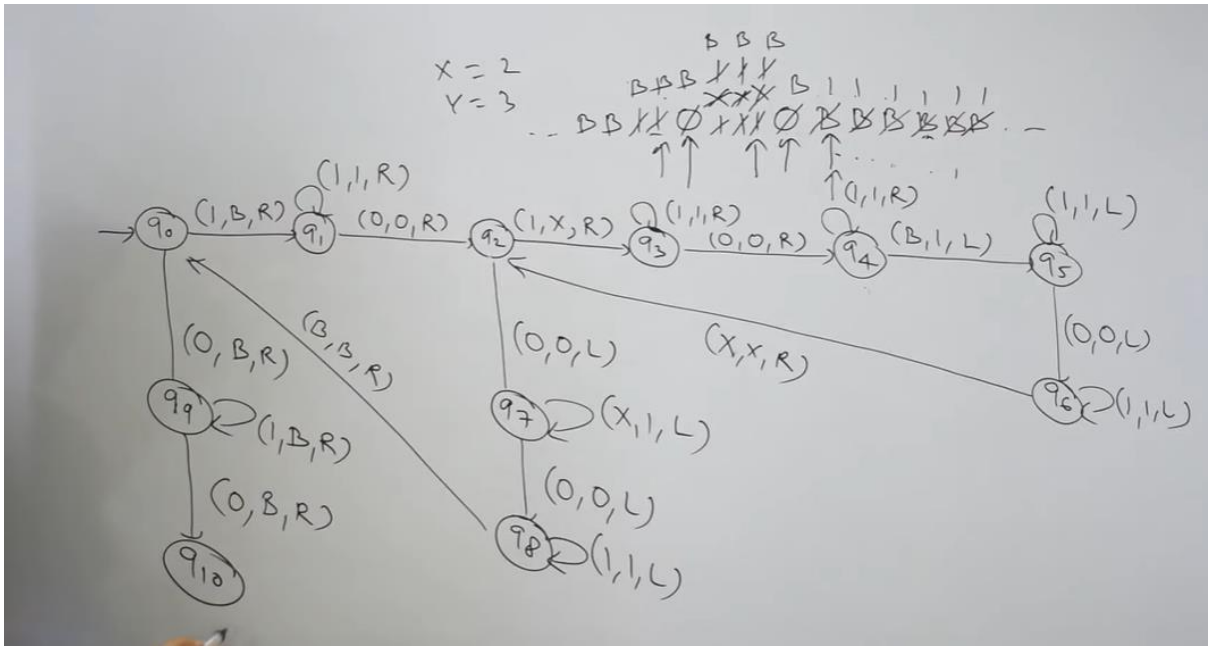
Turing Machine for 2's complement



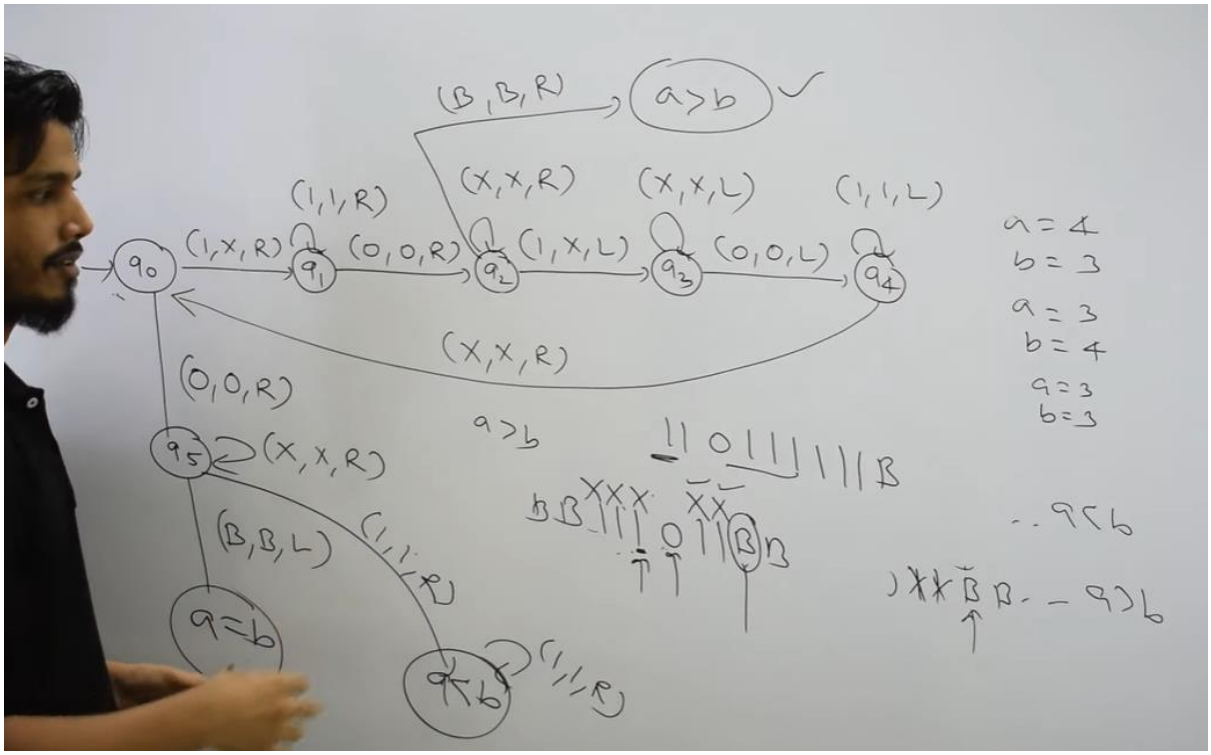
Turing Machine for Addition



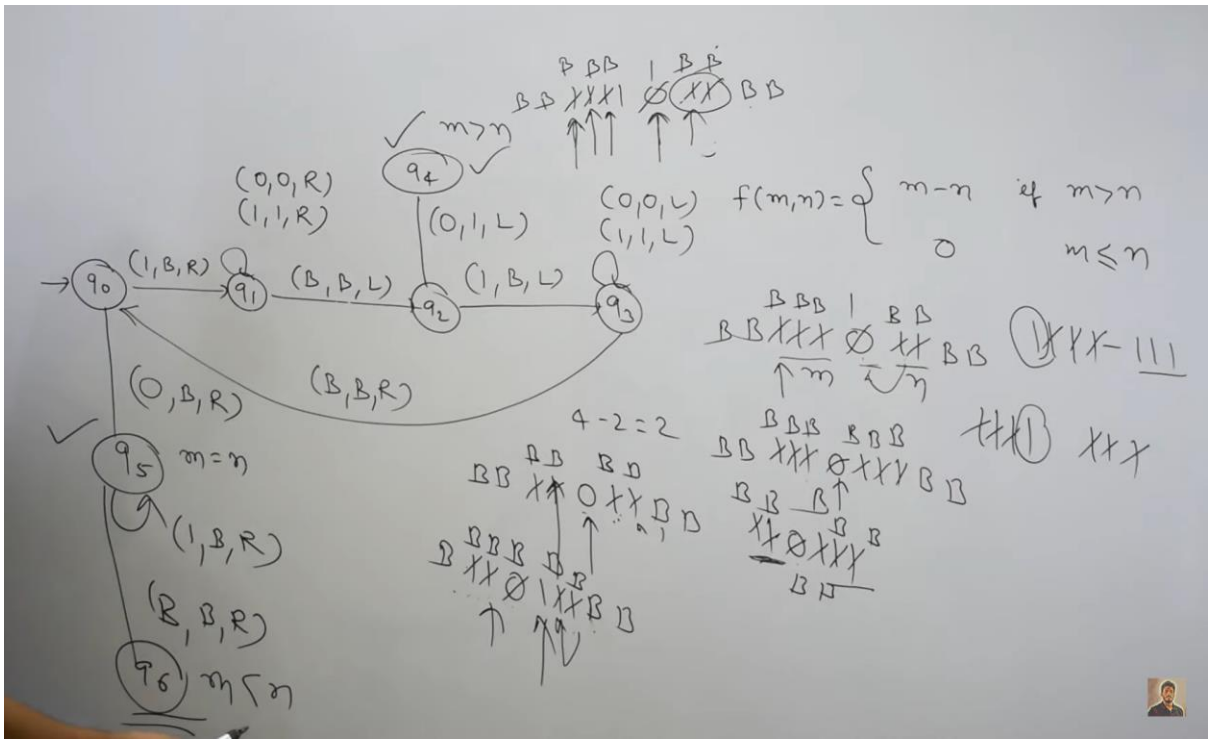
Turing Machine for Multiplication



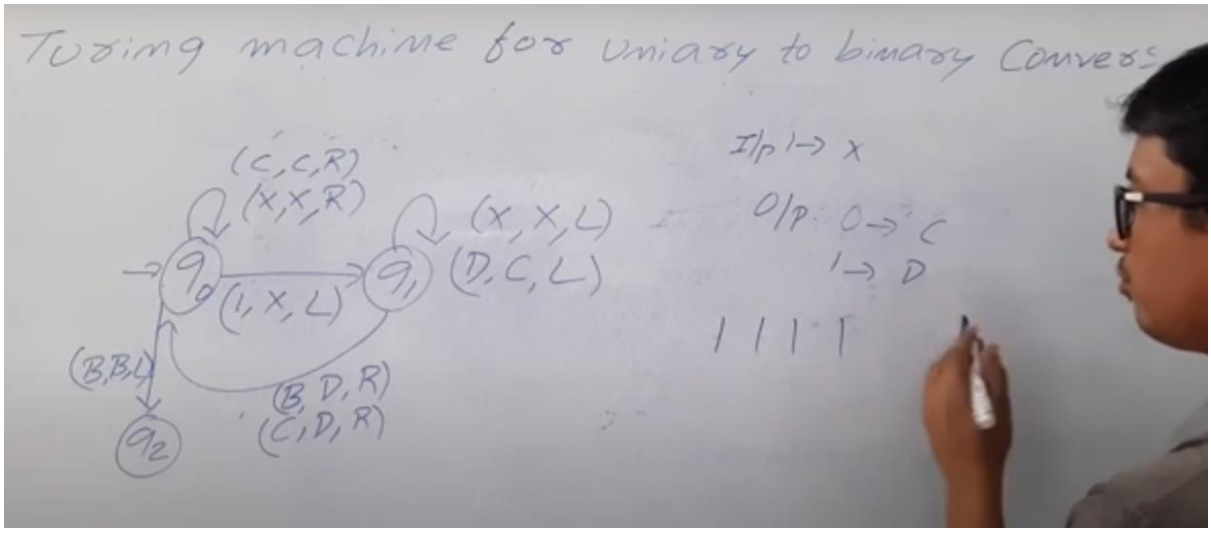
Turing Machine for Comparison



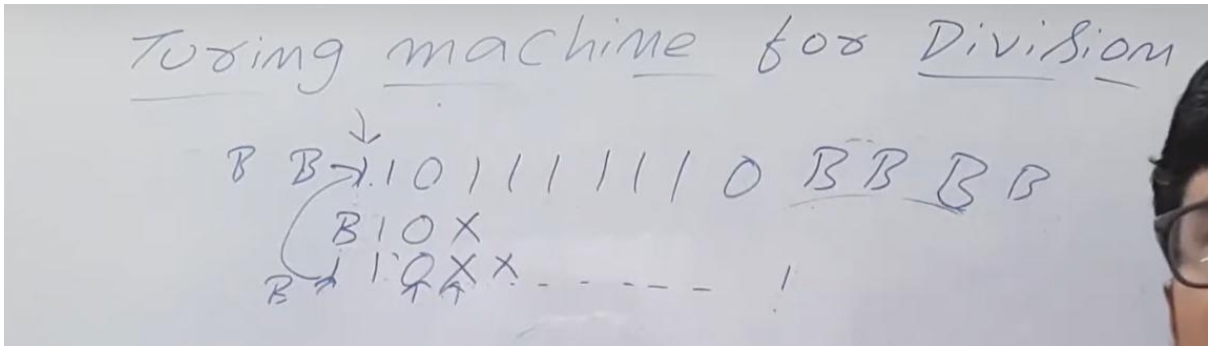
Turing Machine for Subtraction

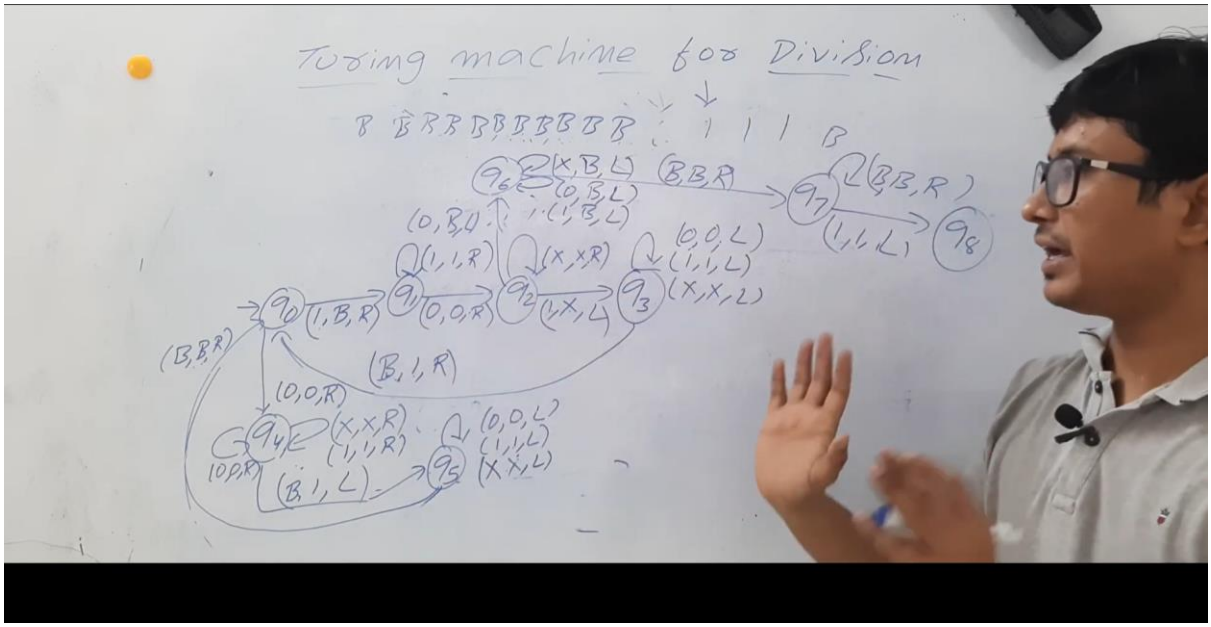


Turing machine for Unary to Binary



Turing machine for Division





Recursive and Recursively Enumerable Languages

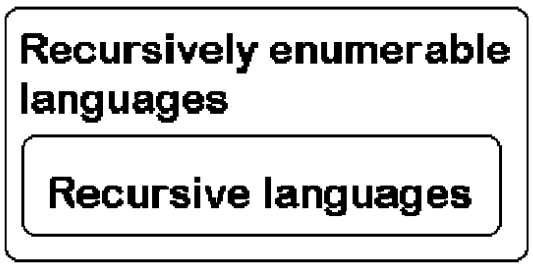
Remember that there are *three* possible outcomes of executing a Turing machine over a given input. The Turing machine may

- Halt and accept the input;
- Halt and reject the input; or
- Never halt.

A language is *recursive* if there exists a Turing machine that accepts every string of the language and rejects every string (over the same alphabet) that is not in the language.

Note that, if a language L is recursive, then its complement -L must also be recursive. (Why?)

A language is *recursively enumerable* if there exists a Turing machine that accepts every string of the language, and does not accept strings that are not in the language. (Strings that are not in the language may be rejected or may cause the Turing machine to go into an infinite loop.)



Clearly, every recursive language is also recursively enumerable. It is not obvious whether every recursively enumerable language is also recursive.

Closure Properties of Recursive Languages

- **Union**: If L1 and If L2 are two recursive languages, their union L1∪L2 will also be recursive because if TM halts for L1 and halts for L2, it will also halt for L1∪L2.
- **Concatenation:** If L1 and If L2 are two recursive languages, their concatenation L1.L2 will also be recursive. For Example:
- L1= $\{a^n b^n c^n | n>=0\}$
- L2= $\{d^m e^m f^m | m>=0\}$
- L3= L1.L2
- = $\{a^n b^n c^n d^m e^m f^m | m>=0$ and $n>=0\}$ is also recursive.

  L1 says n no. of a's followed by n no. of b's followed by n no. of c's. L2 says m no. of d's followed by m no. of e's followed by m no. of f's. Their concatenation first matches no. of a's, b's and c's and then matches no. of d's, e's and f's. So it can be decided by TM.

- **Kleene Closure:** If L1is recursive, its kleene closure L1* will also be recursive. For Example:

L1= $\{a^n b^n c^n | n>=0\}$

L1*= $\{ a^n b^n c^n || n>=0\}^*$ is also recursive.

- **Intersection and complement**: If L1 and If L2 are two recursive languages, their intersection L1 ∩ L2 will also be recursive. For Example:
- L1= {$a^nb^nc^ndm|n>=0$ and $m>=0$}
- L2= {$a^nb^nc^nd^n|n>=0$ and $m>=0$}
- L3=L1 ∩ L2
- = { $a^nb^nc^nd^n$ $|n>=0$} will be recursive.

L1 says n no. of a's followed by n no. of b's followed by n no. of c's and then any no. of d's. L2 says any no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. Their intersection says n no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. So it can be decided by turing machine, hence recursive. Similarly, complementof recursive language L1 which is ∑*-L1, will also be recursive.

RE - Recursive Enumerable REC- Recursive Language

Note: As opposed to REC languages, RE languages are not closed under complementon which means complement of RE language need not be RE.

The Church - Turing Thesis

Intuitive notion of an algorithm: a sequence of steps to solve a problem.

Questions: What is the meaning of "solve" and "problem"?

Answers:

Problem: This is a mapping. Can be represented as a function, or as a set membership "yes/no" question.

To solve a problem: To find a Turing machine that computes the function or answers the question.

Church-Turing Thesis: Any Turing machine that halts on all inputs corresponds to an algorithm,
and any algorithm can be represented by a Turing machine.

This is the formal definition of an algorithm. This is not a theorem - only a hypothesis.

In computability theory, the **Church–Turing thesis** (also known as the **Church-Turing conjecture**, **Church's thesis**, **Church's conjecture**, and **Turing's thesis**) is a combined hypothesis ("thesis") about the nature of functions whose values are effectively calculable; i.e. computable. In simple terms, it states that "everything computable is computable by a Turing machine."

Counter machine

A **counter machine** is an abstract machine used in formal logic and theoretical computer science to model computation. It is the most primitive of the four types of register machines. A counter machine comprises a set of one or more unbounded *registers*, each of which can hold a single non-negative integer, and a list of (usually sequential) arithmetic and control instructions for the machine to follow.

The primitive model register machine is, in effect, a multitape 2-symbol Post-Turing machine with its behavior restricted so its tapes act like simple "counters".

By the time of Melzak, Lambek, and Minsky the notion of a "computer program" produced a different type of simple machine with many left-ended tapes cut from a Post-Turing tape. In all cases the models permit only two tape symbols { mark, blank }.[3]

Some versions represent the positive integers as only a strings/stack of marks allowed in a "register" (i.e. left-ended tape), and a blank tape represented by the count "0". Minsky eliminated the PRINT instruction at the expense of providing his model with a mandatory single mark at the left-end of each tape.[3]

In this model the single-ended tapes-as-registers are thought of as "counters", their instructions restricted to only two (or three if the TEST/DECREMENT instruction is atomized). Two common instruction sets are the following:

(1): { INC (r), DEC (r), JZ (r,z) }, i.e.

{ INCrement contents of register #r; DECrement contents of register #r; IF contents of #r=Zero THEN Jump-to Instruction #z}

(2): { CLR (r); INC (r); JE (r_i, r_j, z) }, i.e.

{ CLeaR contents of register r; INCrement contents of r; compare contents of r_i to r_j and if Equal then Jump to instruction z}

Although his model is more complicated than this simple description, the Melzak "pebble" model extended this notion of "counter" to permit multi- pebble adds and subtracts.

Basic features

For a given counter machine model the instruction set is tiny—from just one to six or seven instructions. Most models contain a few arithmetic operations and at least one conditional operation (if *condition* is true, then jump). Three *base models*, each using three instructions, are drawn from the following collection. (The abbreviations are arbitrary.)

- CLR (r): CLeaR register *r*. (Set *r* to zero.)
- INC (r): INCrement the contents of register *r*.
- DEC (r): DECrement the contents of register *r*.
- CPY (r_j, r_k): CoPY the contents of register r_j to register r_k leaving the contents of r_j intact.
- JZ (r, z): IF register *r* contains Zero THEN Jump to instruction *z* ELSE continue in sequence.
- JE (r_j, r_k, z): IF the contents of register r_j Equals the contents of register r_k THEN Jump to instruction *z* ELSE continue in sequence.

In addition, a machine usually has a HALT instruction, which stops the machine (normally after the result has been computed).

Using the instructions mentioned above, various authors have discussed certain counter machines:

- set 1: { INC (r), DEC (r), JZ (r, z) }, (Minsky (1961, 1967), Lambek (1961))
- set 2: { CLR (r), INC (r), JE (r_j, r_k, z) }, (Ershov (1958), Peter (1958) as interpreted by Shepherdson-Sturgis (1964); Minsky (1967); Schönhage (1980))
- set 3: { INC (r), CPY (r_j, r_k), JE (r_j, r_k, z) }, (Elgot-Robinson (1964), Minsky (1967))

The three counter machine base models have the same computational power since the instructions of one model can be derived from those of another. All are equivalent to the computational power of Turing machines (but only if Gödel numbers are used to encode data in the register or registers; otherwise their power is equivalent to the primitive recursive functions). Due to their unary processing style, counter machines are typically exponentially slower than comparable Turing machines.

Universal Turing Machines

Turing machines are abstract computing devices. Each Turing machine represents a particular algorithm. Hence we can think of Turing machines as being "hard-wired".

Is there a programmable Turing machine that can solve any problem solved by a "hard-wired" Turing machine?

The answer is "yes", the programmable Turing machine is called "universal Turing machine".

Basic Idea:

The Universal TM will take as input a description of a standard TM and an input ***w*** in the alphabet of the standard TM, and will halt if and only if the standard TM halts on ***w***.