Pumping Lemma for CFG

Lemma

If **L** is a context-free language, there is a pumping length **p** such that any string **w** ∈ **L** of length **≥ p** can be written as **w = uvxyz**, where **vy ≠ ε**, **|vxy| ≤ p**, and for all **i ≥ 0, uv^ixy^iz ∈ L**.

Applications of Pumping Lemma

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Problem

Find out whether the language **$L = \{x^ny^nz^n \mid n \geq 1\}$** is context free or not.

Solution

Let **L** is context free. Then, **L** must satisfy pumping lemma.

At first, choose a number **n** of the pumping lemma. Then, take z as $0^n1^n2^n$.

Break **z** into **uvwxy,** where

|vwx| ≤ n and vx ≠ ε.

Hence **vwx** cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases −

Case 1 − **vwx** has no 2s. Then **vx** has only 0s and 1s. Then **uwy**, which would have to be in **L**, has **n** 2s, but fewer than **n** 0s or 1s.

Case 2 − **vwx** has no 0s.

Here contradiction occurs.

Hence, **L** is not a context-free language.

Greibach Normal Form

A CFG is in Greibach Normal Form if the Productions are in the following forms −

$A \rightarrow b$

$A \rightarrow bD_1 \ldots D_n$

$S \rightarrow \varepsilon$

where A, D_1,....,D_n are non-terminals and b is a terminal.

Algorithm to Convert a CFG into Greibach 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'** → **S**.

Step 2 − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 − Remove all direct and indirect left-recursion.

Step 5 − Do proper substitutions of productions to convert it into the proper form of GNF.

Problem

Convert the following CFG into CNF

$S \rightarrow XY \mid Xn \mid p$

$X \rightarrow mX \mid m$

$Y \rightarrow Xn \mid o$

Solution

Here, **S** does not appear on the right side of any production and there are no unit or null productions in the production rule set. So, we can skip Step 1 to Step 3.

Step 4

Now after replacing

X in $S \rightarrow XY \mid Xo \mid p$

with

$mX \mid m$

we obtain

$S \rightarrow mXY \mid mY \mid mXo \mid mo \mid p.$

And after replacing

X in $Y \rightarrow X_n \mid o$

with the right side of

$X \rightarrow mX \mid m$

we obtain

$Y \rightarrow mXn \mid mn \mid o.$

Two new productions $O \rightarrow o$ and $P \rightarrow p$ are added to the production set and then we came to the final GNF as the following −

$S \rightarrow mXY \mid mY \mid mXC \mid mC \mid p$

$X \rightarrow mX \mid m$

$Y \rightarrow mXD \mid mD \mid o$

$O \rightarrow o$

$P \rightarrow p$

CFG Simplification

In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called **simplification of CFGs**. Simplification essentially comprises of the following steps −

- Reduction of CFG
- Removal of Unit Productions
- Removal of Null Productions

Reduction of CFG

CFGs 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, $\mathbf{W_1}$, that derive some terminal and initialize **i=1**.

Step 2 − Include all symbols, $\mathbf{W_{i+1}}$, that derive $\mathbf{W_i}$.

Step 3 − Increment **i** and repeat Step 2, until $\mathbf{W_{i+1} = W_i}$.

Step 4 − Include all production rules that have $\mathbf{W_i}$ in it.

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

Derivation Procedure −

Step 1 − Include the start symbol in $\mathbf{Y_1}$ and initialize **i = 1**.

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

Step 3 − Increment **i** and repeat Step 2, until $\mathbf{Y_{i+1} = Y_i}$.

Problem

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

Solution

Phase 1 −

T = { a, c, e }

W_1 = { A, C, E } from rules A → a, C → c and E → aA

W_2 = { A, C, E } U { S } from rule S → AC

W_3 = { A, C, E, S } U ∅

Since $W_2 = W_3$, we can derive G' as −

G' = { { A, C, E, S }, { a, c, e }, P, {S}}

where P: S → AC, A → a, C → c , E → aA | e

Phase 2 −

Y_1 = { S }

Y_2 = { S, A, C } from rule S → AC

Y_3 = { S, A, C, a, c } from rules A → a and C → c

Y_4 = { S, A, C, a, c }

Since $Y_3 = Y_4$, we can derive G" as −

G" = { { A, C, S }, { a, c }, P, {S}}

where P: S → AC, A → a, C → c

Removal of Unit Productions

Any production rule in the form A → B where A, B ∈ Non-terminal is called **unit production.**.

Removal Procedure −

Step 1 − To remove **A** → **B**, add production **A** → **x** to the grammar rule whenever **B** → **x** occurs in the grammar. [x ∈ Terminal, x can be Null]

Step 2 − Delete **A** → **B** from the grammar.

Step 3 − Repeat from step 1 until all unit productions are removed.

Problem

Remove unit production from the following −

S → XY, X → a, Y → Z | b, Z → M, M → N, N → a

Solution −

There are 3 unit productions in the grammar −

Y → Z, Z → M, and M → N

At first, we will remove M → N.

As N → a, we add M → a, and M → N is removed.

The production set becomes

S → XY, X → a, Y → Z | b, Z → M, M → a, N → a

Now we will remove Z → M.

As M → a, we add Z→ a, and Z → M is removed.

The production set becomes

S → XY, X → a, Y → Z | b, Z → a, M → a, N → a

Now we will remove Y → Z.

As Z → a, we add Y→ a, and Y → Z is removed.

The production set becomes

S → XY, X → a, Y → a | b, Z → a, M → a, N → a

Now Z, M, and N are unreachable, hence we can remove those.

The final CFG is unit production free −

S → XY, X → a, Y → a | b

Removal of Null Productions

In a CFG, a non-terminal symbol **'A'** is a nullable variable if there is a production **A → ε** or there is a derivation that starts at **A** and finally ends up with

ε: A → .....… → ε

Removal Procedure

Step 1 − Find out nullable non-terminal variables which derive ε.

Step 2 − For each production **A → a**, construct all productions **A → x** where **x** is obtained from **'a'** by removing one or multiple non-terminals from Step 1.

Step 3 − Combine the original productions with the result of step 2 and remove **ε - productions**.

Problem

Remove null production from the following −

S → ASA | aB | b, A → B, B → b | ∈

Solution −

There are two nullable variables − **A** and **B**

At first, we will remove B → ε.

After removing **B → ε**, the production set becomes −

S→ASA | aB | b | a, A ε B| b | &epsilon, B → b

Now we will remove A → ε.

After removing **A → ε**, the production set becomes −

S→ASA | aB | b | a | SA | AS | S, A → B| b, B → b

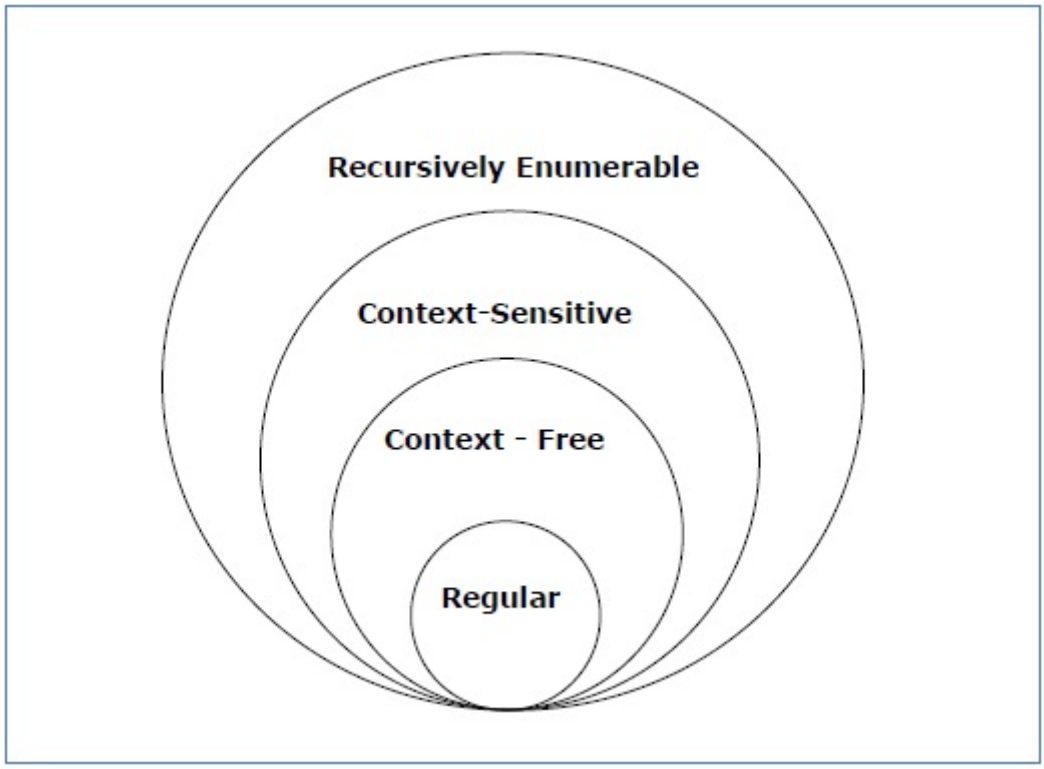
This is the final production set without null transition.

Chomsky Classification of Grammars

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other −

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type of grammar −



Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

```
X → ε
X → a | aY
Y → b
```

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

```
S → X a
X → a
X → aX
X → abc
X → ε
```

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

```
AB → AbBc
A → bcA
B → b
```

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α** → **β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Example

```
S → ACaB
Bc → acB
CB → DB
aD → Db
```

Language Generated by a Grammar

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar. A language generated by a grammar **G** is a subset formally defined by

$$L(G)=\{W|W \in \sum^*, S \Rightarrow G\ \mathbf{W}\}$$

If **L(G1) = L(G2)**, the Grammar **G1** is equivalent to the Grammar **G2**.

Example

If there is a grammar

G: N = {S, A, B} T = {a, b} P = {S → AB, A → a, B → b}

Here **S** produces **AB**, and we can replace **A** by **a**, and **B** by **b**. Here, the only accepted string is **ab**, i.e.,

L(G) = {ab}

Example

Suppose we have the following grammar −

G: N = {S, A, B} T = {a, b} P = {S → AB, A → aA|a, B → bB|b}

The language generated by this grammar −

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

$= \{a^m\ b^n \mid m \geq 1 \text{ and } n \geq 1\}$

Construction of a Grammar Generating a Language

We'll consider some languages and convert it into a grammar G which produces those languages.

Example

Problem − Suppose, $L(G) = \{a^m\ b^n \mid m \geq 0 \text{ and } n > 0\}$. We have to find out the grammar **G** which produces **L(G)**.

Solution

Since $L(G) = \{a^m\ b^n \mid m \geq 0 \text{ and } n > 0\}$

the set of strings accepted can be rewritten as −

L(G) = {b, ab,bb, aab, abb, …….}

Here, the start symbol has to take at least one 'b' preceded by any number of 'a' including null.

To accept the string set {b, ab, bb, aab, abb, …….}, we have taken the productions −

$S \rightarrow aS$, $S \rightarrow B$, $B \rightarrow b$ and $B \rightarrow bB$

$S \rightarrow B \rightarrow b$ (Accepted)

$S \rightarrow B \rightarrow bB \rightarrow bb$ (Accepted)

$S \rightarrow aS \rightarrow aB \rightarrow ab$ (Accepted)

$S \rightarrow aS \rightarrow aaS \rightarrow aaB \rightarrow aab$(Accepted)

$S \rightarrow aS \rightarrow aB \rightarrow abB \rightarrow abb$ (Accepted)

Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar −

G: ({S, A, B}, {a, b}, S, { $S \rightarrow aS \mid B$, $B \rightarrow b \mid bB$ })

Example

Problem − Suppose, L (G) = {$a^m b^n \mid m > 0$ and $n \geq 0$}. We have to find out the grammar G which produces L(G).

Solution −

Since L(G) = {$a^m b^n \mid m > 0$ and $n \geq 0$}, the set of strings accepted can be rewritten as −

L(G) = {a, aa, ab, aaa, aab ,abb, …….}

Here, the start symbol has to take at least one ‘a’ followed by any number of ‘b’ including null.

To accept the string set {a, aa, ab, aaa, aab, abb, …….}, we have taken the productions −

$S \rightarrow aA$, $A \rightarrow aA$, $A \rightarrow B$, $B \rightarrow bB$,$B \rightarrow \lambda$

$S \rightarrow aA \rightarrow aB \rightarrow a\lambda \rightarrow a$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaB \rightarrow aa\lambda \rightarrow aa$ (Accepted)

$S \rightarrow aA \rightarrow aB \rightarrow abB \rightarrow ab\lambda \rightarrow ab$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaaA \rightarrow aaaB \rightarrow aaa\lambda \rightarrow aaa$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaB \rightarrow aabB \rightarrow aab\lambda \rightarrow aab$ (Accepted)

$S \rightarrow aA \rightarrow aB \rightarrow abB \rightarrow abbB \rightarrow abb\lambda \rightarrow abb$ (Accepted)

Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar −

G: ({S, A, B}, {a, b}, S, {$S \rightarrow aA$, $A \rightarrow aA \mid B$, $B \rightarrow \lambda \mid bB$ })

Introduction to Grammars

n the literary sense of the term, grammars denote syntactical rules for conversation in natural languages. Linguistics have attempted to define grammars since the inception of natural languages like English, Sanskrit, Mandarin, etc.

The theory of formal languages finds its applicability extensively in the fields of Computer Science. **Noam Chomsky** gave a mathematical model of grammar in 1956 which is effective for writing computer languages.

Grammar

A grammar **G** can be formally written as a 4-tuple (N, T, S, P) where −

- **N** or $\mathbf{V_N}$ is a set of variables or non-terminal symbols.
- **T** or $\sum$ is a set of Terminal symbols.
- **S** is a special variable called the Start symbol, $S \in N$
- **P** is Production rules for Terminals and Non-terminals. A production rule has the form $\alpha \rightarrow \beta$, where α and β are strings on $V_N \cup \sum$ and least one symbol of α belongs to V_N.

Example

Grammar G1 −

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

Here,

- **S, A,** and **B** are Non-terminal symbols;
- **a** and **b** are Terminal symbols
- **S** is the Start symbol, $S \in N$
- Productions, **P : S → AB, A → a, B → b**

Example

Grammar G2 −

$$(\{S, A\}, \{a, b\}, S, \{S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow \varepsilon\})$$

Here,

- **S** and **A** are Non-terminal symbols.
- **a** and **b** are Terminal symbols.
- **ε** is an empty string.

- **S** is the Start symbol, S ∈ N
- Production **P : S → aAb, aA → aaAb, A → ε**

Derivations from a Grammar

Strings may be derived from other strings using the productions in a grammar. If a grammar **G** has a production **α → β**, we can say that **x α y** derives **x β y** in **G**. This derivation is written as −

$$x\ \alpha\ y \Rightarrow_G x\ \beta\ y$$

Example

Let us consider the grammar −

G2 = ({S, A}, {a, b}, S, {S → aAb, aA → aaAb, A → ε })

Some of the strings that can be derived are −

S ⇒ aAb using production S → aAb

⇒ aaAbb using production aA → aAb

⇒ aaaAbbb using production aA → aAb

⇒ aaabbb using production A → ε

Moore and Mealy Machines

Finite automata may have outputs corresponding to each transition. There are two types of finite state machines that generate output −

- Mealy Machine
- Moore machine

Mealy Machine

A Mealy Machine is an FSM whose output depends on the present state as well as the present input.

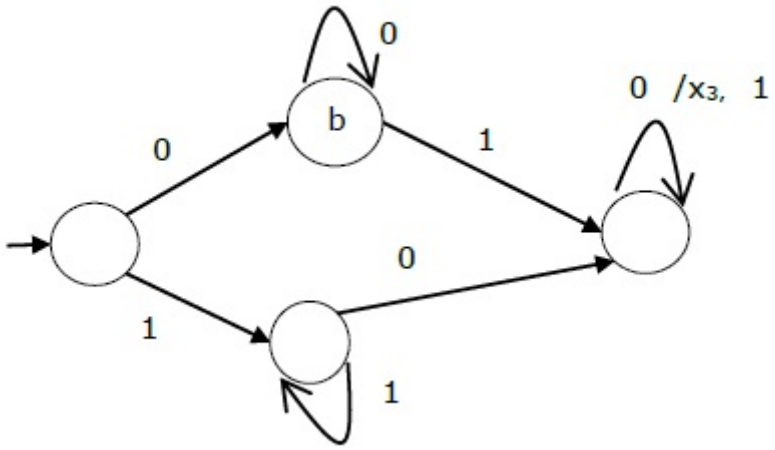
It can be described by a 6 tuple $(Q, \sum, O, \delta, X, q_0)$ where −

- **Q** is a finite set of states.
- $\sum$ is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where $\delta: Q \times \sum \rightarrow Q$
- **X** is the output transition function where $X: Q \times \sum \rightarrow O$
- $\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).

The state table of a Mealy Machine is shown below −

Present state	Next state			
	input = 0		input = 1	
	State	Output	State	Output
→ a	b	x_1	c	x_1
b	b	x_2	d	x_3
c	d	x_3	c	x_1
d	d	x_3	d	x_2

The state diagram of the above Mealy Machine is −



Moore Machine

Moore machine is an FSM whose outputs depend on only the present state.

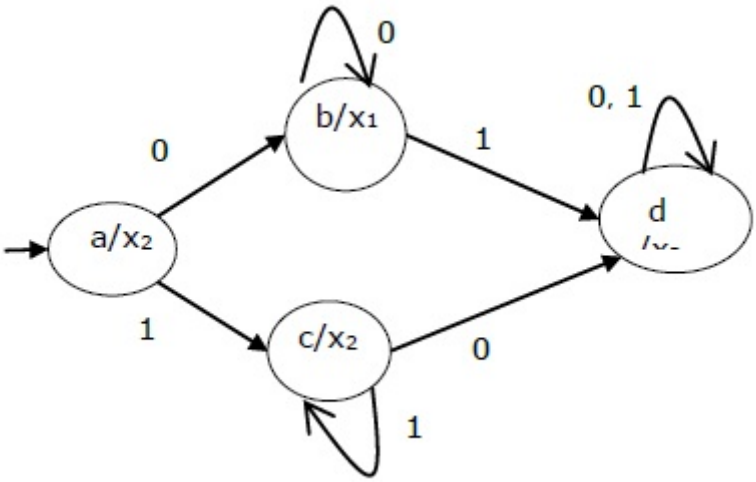
A Moore machine can be described by a 6 tuple (Q, ∑, O, δ, X, q_0) where −

- **Q** is a finite set of states.
- ∑ is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where δ: Q × ∑ → Q
- **X** is the output transition function where X: Q → O
- **q_0** is the initial state from where any input is processed (q_0 ∈ Q).

The state table of a Moore Machine is shown below −

Present state	Next State		Output
	Input = 0	Input = 1	
→ a	b	c	x_2
b	b	d	x_1
c	c	d	x_2
d	d	d	x_3

The state diagram of the above Moore Machine is −



Mealy Machine vs. Moore Machine

The following table highlights the points that differentiate a Mealy Machine from a Moore Machine.

Mealy Machine	Moore Machine
Output depends both upon the present state and the present input	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
The value of the output function is a function of the transitions and the changes, when the input logic on the present state is done.	The value of the output function is a function of the current state and the changes at the clock edges, whenever state changes occur.
Mealy machines react faster to inputs. They generally react in the same clock cycle.	In Moore machines, more logic is required to decode the outputs resulting in more circuit delays. They generally react one clock cycle later.

Moore Machine to Mealy Machine

Algorithm 4

Input − Moore Machine

Output − Mealy Machine

Step 1 − Take a blank Mealy Machine transition table format.

Step 2 − Copy all the Moore Machine transition states into this table format.

Step 3 − Check the present states and their corresponding outputs in the Moore Machine state table; if for a state Q_i output is m, copy it into the output columns of the Mealy Machine state table wherever Q_i appears in the next state.

Example

Let us consider the following Moore machine −

Present State	Next State		Output
	a = 0	a = 1	
→ a	d	b	1
b	a	d	0
c	c	c	0
d	b	a	1

Now we apply Algorithm 4 to convert it to Mealy Machine.

Step 1 & 2 −

Present State	Next State			
	a = 0		a = 1	
	State	Output	State	Output
→ a	d		b	
b	a		d	
c	c		c	
d	b		a	

Step 3 −

Present State	Next State			
	a = 0		a = 1	
	State	Output	State	Output
=> a	d	1	b	0
b	a	1	d	1
c	c	0	c	0
d	b	0	a	1

Mealy Machine to Moore Machine

Algorithm 5

Input − Mealy Machine

Output − Moore Machine

Step 1 − Calculate the number of different outputs for each state (Q_i) that are available in the state table of the Mealy machine.

Step 2 − If all the outputs of Qi are same, copy state Q_i. If it has n distinct outputs, break Q_i into n states as Q_{in} where **n** = 0, 1, 2.....

Step 3 − If the output of the initial state is 1, insert a new initial state at the beginning which gives 0 output.

Example

Let us consider the following Mealy Machine −

Present State	Next State			
	a = 0		a = 1	
	Next State	Output	Next State	Output
→ a	d	0	b	1
b	a	1	d	0
c	c	1	c	0
d	b	0	a	1

Here, states ‘a’ and ‘d’ give only 1 and 0 outputs respectively, so we retain states ‘a’ and ‘d’. But states ‘b’ and ‘c’ produce different outputs (1 and 0). So, we divide **b** into $\mathbf{b_0}$, $\mathbf{b_1}$ and **c** into $\mathbf{c_0}$, $\mathbf{c_1}$.

Present State	Next State		Output
	a = 0	a = 1	
→ a	d	b_1	1
b_0	a	d	0
b_1	a	d	1
c_0	c_1	C_0	0
c_1	c_1	C_0	1
d	b_0	a	0

DFA Minimization

DFA Minimization using Myphill-Nerode Theorem

Algorithm

Input − DFA

Output − Minimized DFA

Step 1 − Draw a table for all pairs of states (Q_i, Q_j) not necessarily connected directly [All are unmarked initially]

Step 2 − Consider every state pair (Q_i, Q_j) in the DFA where $Q_i \in F$ and $Q_j \notin F$ or vice versa and mark them. [Here F is the set of final states]

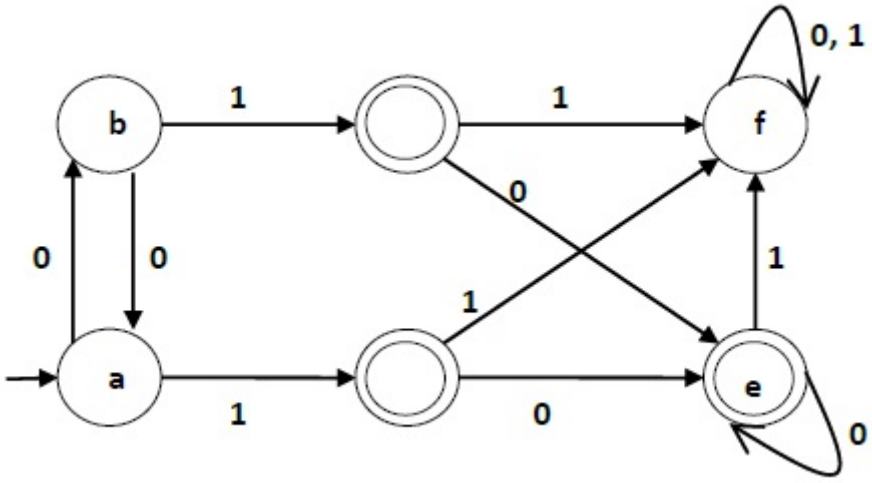
Step 3 − Repeat this step until we cannot mark anymore states −

If there is an unmarked pair (Q_i, Q_j), mark it if the pair {δ (Q_i, A), δ (Q_i, A)} is marked for some input alphabet.

Step 4 − Combine all the unmarked pair (Q_i, Q_j) and make them a single state in the reduced DFA.

Example

Let us use Algorithm 2 to minimize the DFA shown below.



Step 1 − We draw a table for all pair of states.

	a	b	c	d	e	f
a						
b						
c						
d						
e						
f						

Step 2 − We mark the state pairs.

	a	b	c	d	e	f
a						
b						
c	✓	✓				
d	✓	✓				
e	✓	✓				
f			✓	✓	✓	

Step 3 − We will try to mark the state pairs, with green colored check mark, transitively. If we input 1 to state ‘a’ and ‘f’, it will go to state ‘c’ and ‘f’ respectively. (c, f) is already marked, hence we will mark pair (a, f). Now, we input 1 to state ‘b’ and ‘f’; it will go to state ‘d’ and ‘f’ respectively. (d, f) is already marked, hence we will mark pair (b, f).

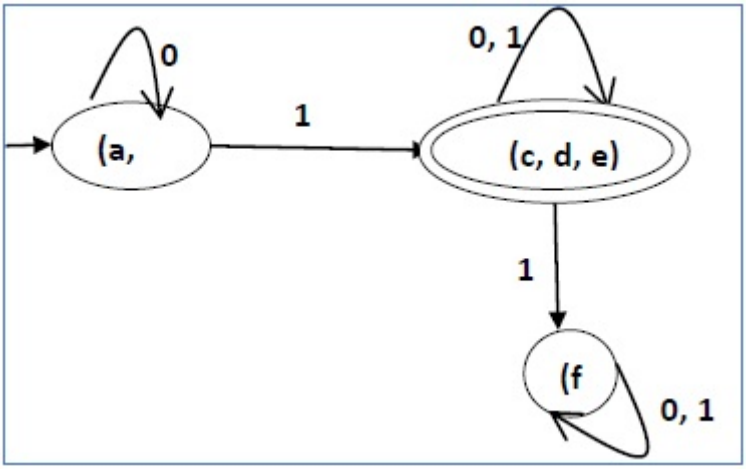
	a	b	c	d	e	f
a						
b						
c	✓	✓				
d	✓	✓				
e	✓	✓				
f	✓	✓	✓	✓	✓	

After step 3, we have got state combinations {a, b} {c, d} {c, e} {d, e} that are unmarked.

We can recombine {c, d} {c, e} {d, e} into {c, d, e}

Hence we got two combined states as − {a, b} and {c, d, e}

So the final minimized DFA will contain three states {f}, {a, b} and {c, d, e}



DFA Minimization using Equivalence Theorem

If X and Y are two states in a DFA, we can combine these two states into {X, Y} if they are not distinguishable. Two states are distinguishable, if there is at least one string S, such that one of δ (X, S) and δ (Y, S) is accepting and another is not accepting. Hence, a DFA is minimal if and only if all the states are distinguishable.

Algorithm 3

Step 1 − All the states **Q** are divided in two partitions − **final states** and **non-final states** and are denoted by $\mathbf{P_0}$. All the states in a partition are 0^{th} equivalent. Take a counter **k** and initialize it with 0.

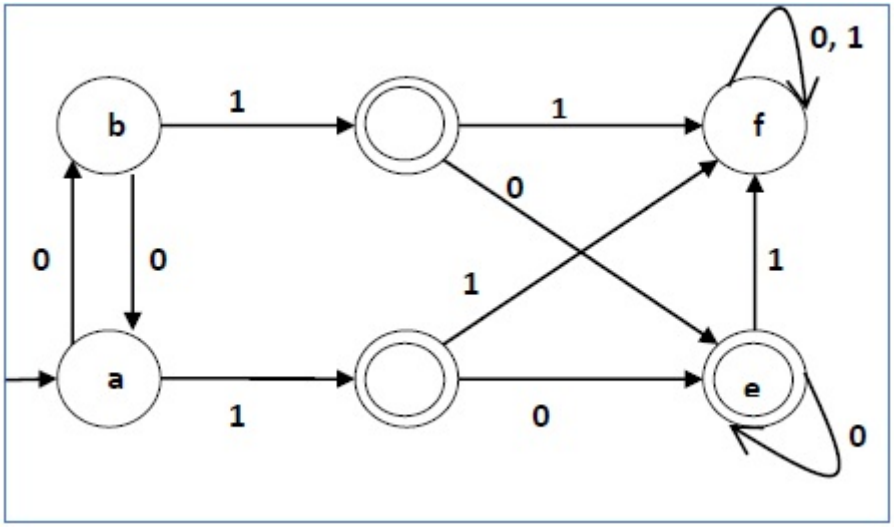
Step 2 − Increment k by 1. For each partition in P_k, divide the states in P_k into two partitions if they are k-distinguishable. Two states within this partition X and Y are k-distinguishable if there is an input **S** such that **δ(X, S)** and **δ(Y, S)** are (k-1)-distinguishable.

Step 3 − If $P_k \neq P_{k-1}$, repeat Step 2, otherwise go to Step 4.

Step 4 − Combine k^{th} equivalent sets and make them the new states of the reduced DFA.

Example

Let us consider the following DFA −



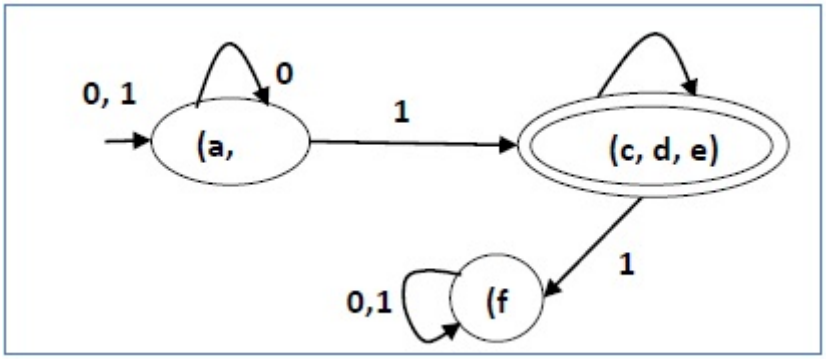
q	δ(q,0)	δ(q,1)
a	b	c
b	a	d
c	e	f
d	e	f
e	e	f
f	f	f

Let us apply the above algorithm to the above DFA −

- P_0 = {(c,d,e), (a,b,f)}
- P_1 = {(c,d,e), (a,b),(f)}
- P_2 = {(c,d,e), (a,b),(f)}

Hence, $P_1 = P_2$.

There are three states in the reduced DFA. The reduced DFA is as follows −



Q	δ(q,0)	δ(q,1)
(a, b)	(a, b)	(c,d,e)
(c,d,e)	(c,d,e)	(f)
(f)	(f)	(f)

NDFA to DFA Conversion

Problem Statement

Let **X = (Q_x, ∑, δ_x, q_0, F_x)** be an NDFA which accepts the language L(X). We have to design an equivalent DFA **Y = (Q_y, ∑, δ_y, q_0, F_y)** such that **L(Y) = L(X)**. The following procedure converts the NDFA to its equivalent DFA −

Algorithm

Input − An NDFA

Output − An equivalent DFA

Step 1 − Create state table from the given NDFA.

Step 2 − Create a blank state table under possible input alphabets for the equivalent DFA.

Step 3 − Mark the start state of the DFA by q0 (Same as the NDFA).

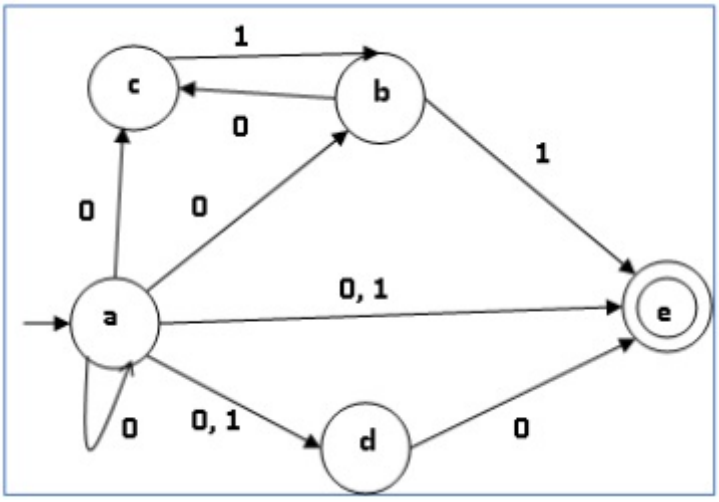
Step 4 − Find out the combination of States {Q_0, Q_1,... , Q_n} for each possible input alphabet.

Step 5 − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

Example

Let us consider the NDFA shown in the figure below.

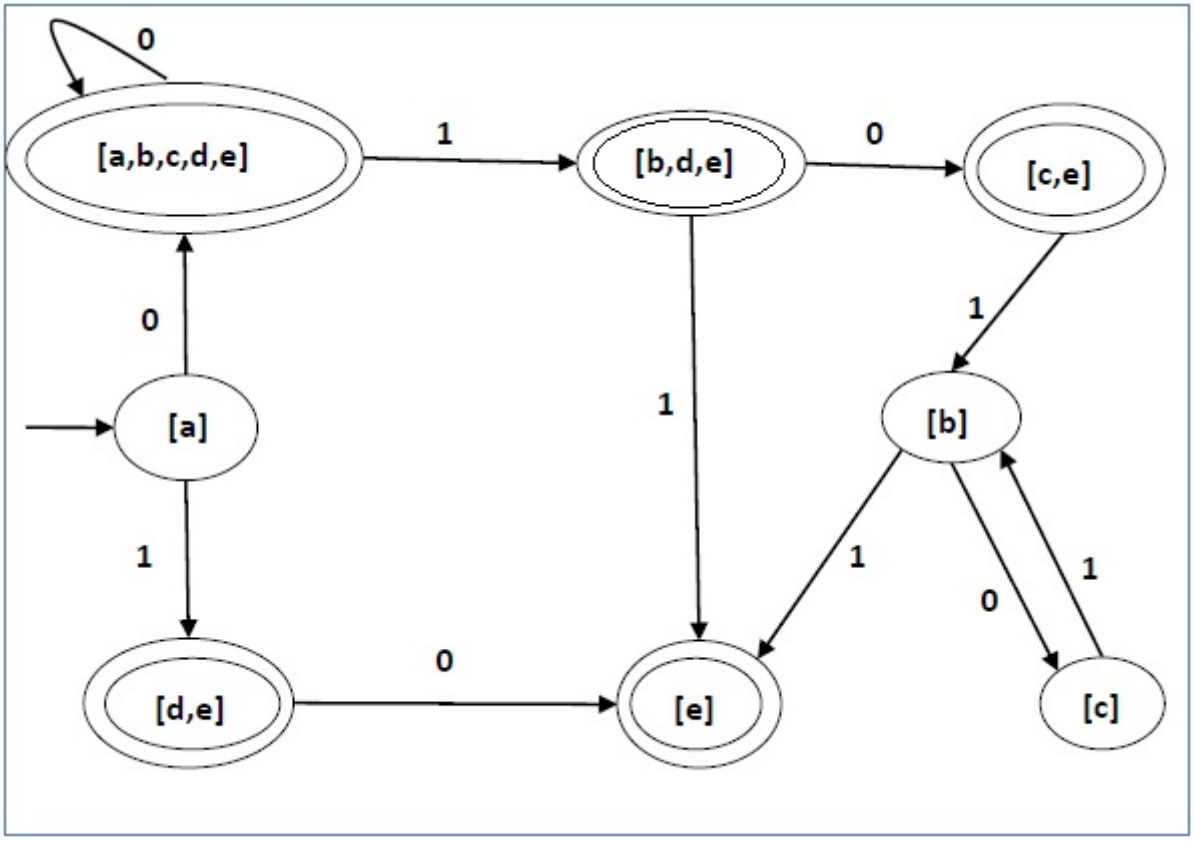


q	δ(q,0)	δ(q,1)
a	{a,b,c,d,e}	{d,e}
b	{c}	{e}
c	∅	{b}
d	{e}	∅
e	∅	∅

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

q	δ(q,0)	δ(q,1)
[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	∅
[b,d,e]	[c,e]	[e]
[e]	∅	∅
[c, e]	∅	[b]
[b]	[c]	[e]
[c]	∅	[b]

The state diagram of the DFA is as follows −


0
[a,b,c,d,e]
1
[b,d,e]
0
[c,e]
0
1
[a]
1
[b]
1
1
1
0
[d,e]
0
[e]
[c]

Construction of DFA

Learn Vid Fun...

Type-02:

For strings starting with a particular substring

Step-01: Decide the minimum number of states required in the DFA and draw them.

> **Rule:** All strings starting with 'n' length substring will require minimum (n+2) states in its DFA

Step-02- Decide the strings for which you will construct the DFA.

Step-03- Construct the DFA for the above decided strings.

Remember: Always go with the existing path. Create a new path only when you can't find a path to go with.

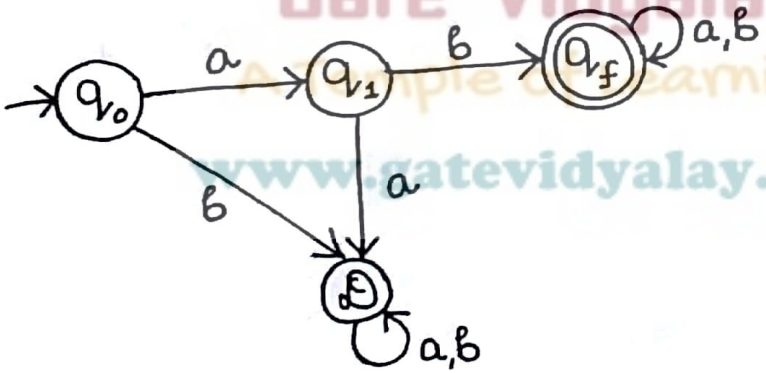
Step-04- After drawing the DFA for the above decided strings, send the left possible combinations to the dead state not over the starting state.

Question- Draw the DFA for the language accepting strings starting with 'ab' over input alphabet $\Sigma = \{a, b\}$

Solution- Regular expression for the given language is-

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

Minimum number of states in the DFA = 4



Strings we will check

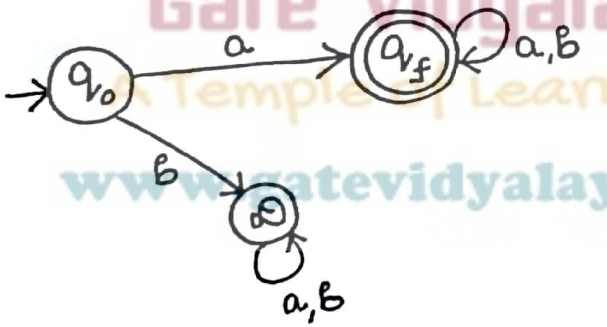
- ab
- aba
- abab

Question- Draw the DFA for the language accepting strings starting with 'a' over input alphabets $\Sigma = \{a, b\}$

Solution- Regular expression for the given language is-

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

Minimum number of states in the DFA = 3



Strings we will check

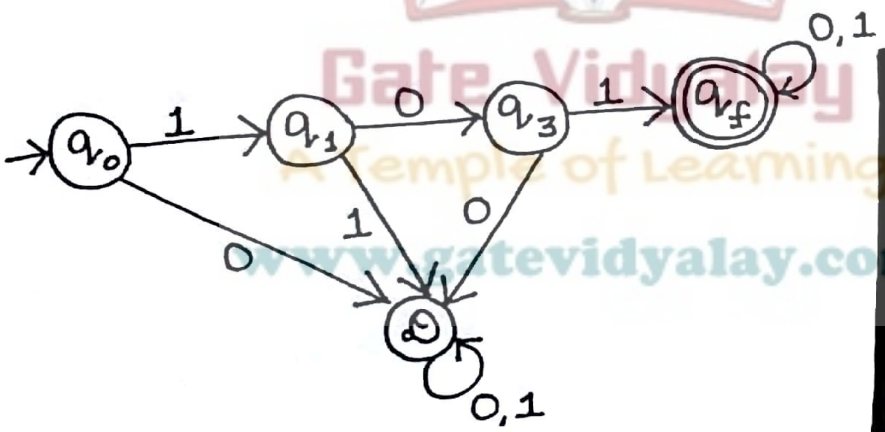
- a
- aa

Question- Draw the DFA for the language accepting strings starting with '101' over input alphabets $\Sigma = \{0, 1\}$

Solution- Regular expression for the given language is-

$$101(0+1)^*$$

Minimum number of states in the DFA = 5



Strings we will check

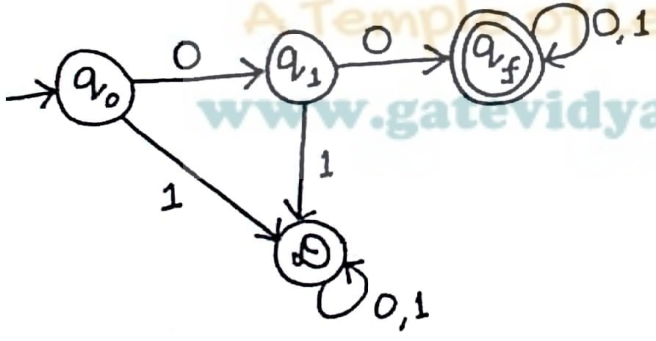
- 101
- 1011
- 10110
- 101101

Question- Construct a DFA that accepts a language L over $\Sigma = \{0,1\}$ such that L is the set of all strings starting with '00'.

Solution- Regular expression for the given language is -

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

Minimum number of states in the DFA = 4



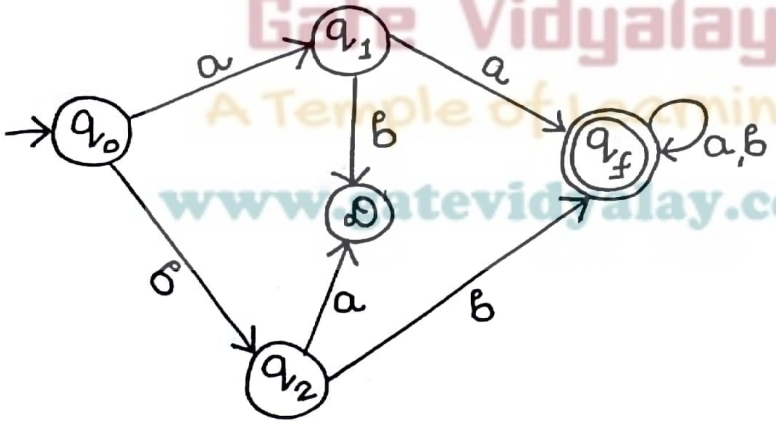
Strings we will check

- 00
- 000
- 00000

Question- Construct a DFA that accepts a language L over $\Sigma = \{a, b\}$ such that L is the set of all strings starting with 'aa' or 'bb'.

Solution- Regular expression for the given language is-

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



Strings we will check

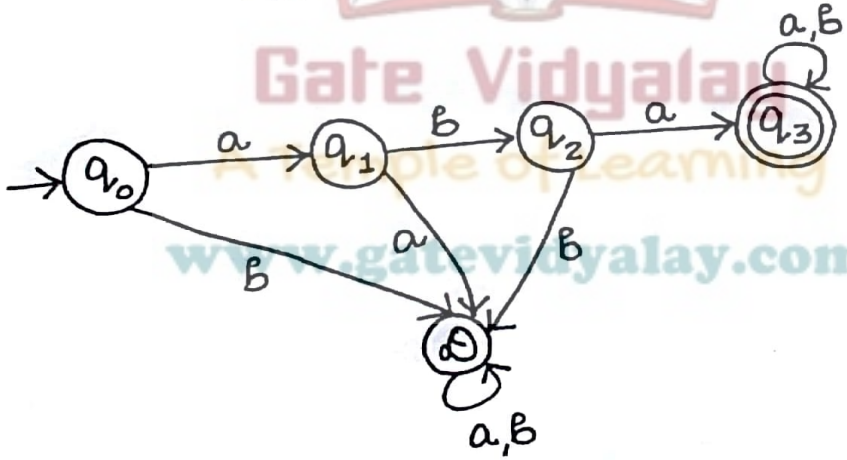
- aa
- aaa
- aaaa
- bb
- bbb
- bbbb

Question- Construct a DFA that accepts a language L over $\Sigma = \{a, b\}$ such that L is the set of all strings starting with 'aba'

Solution- Regular expression for the given language is-

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

Minimum number of states in the DFA = 5



Strings we will check

- aba
- abaa
- abaab
- abaaba