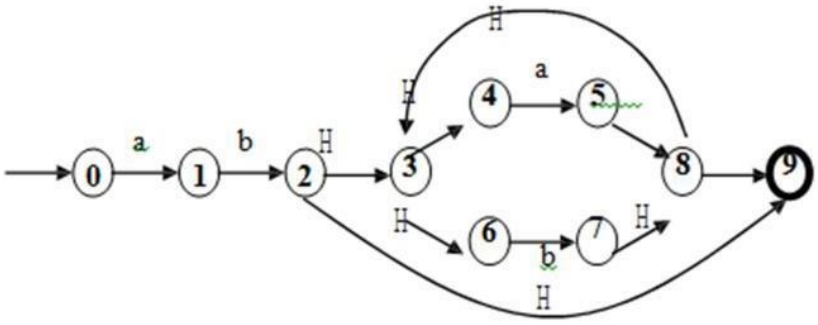
Theory of Automata & Formal Language

Final Term Examination

Q1: (a) Convert the following NFA to its equivalent DFA: (5 Marks)



Conversion from NFA to DFA

In this section, we will discuss the method of converting NFA to its equivalent DFA. In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M). There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').

Steps for converting NFA to DFA:

Step 1: Initially Q' = ϕ

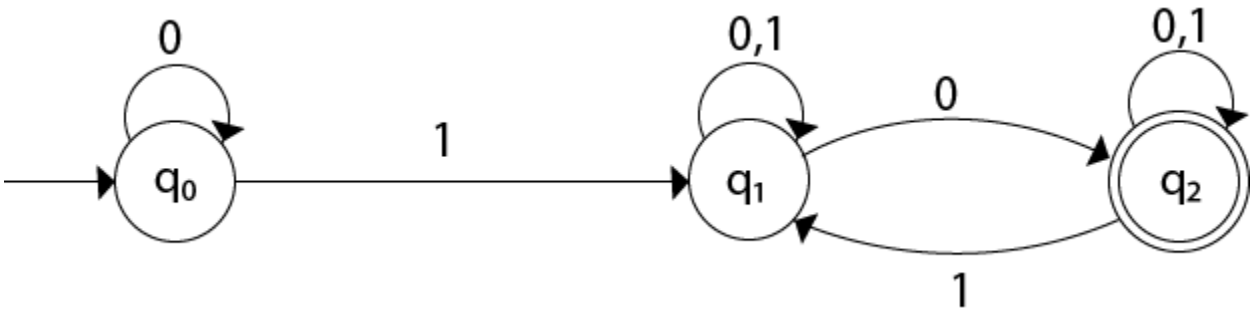
Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

Step 4: In DFA, the final state will be all the states which contain F(final states of NFA)

Example 1:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

Now we will obtain δ' transition for state q0.

1. δ'([q0], 0) = [q0]
2. δ'([q0], 1) = [q1]

The δ' transition for state q1 is obtained as:

1. δ'([q1], 0) = [q1, q2] (**new** state generated)
2. δ'([q1], 1) = [q1]

The δ' transition for state q2 is obtained as:

1. δ'([q2], 0) = [q2]
2. δ'([q2], 1) = [q1, q2]

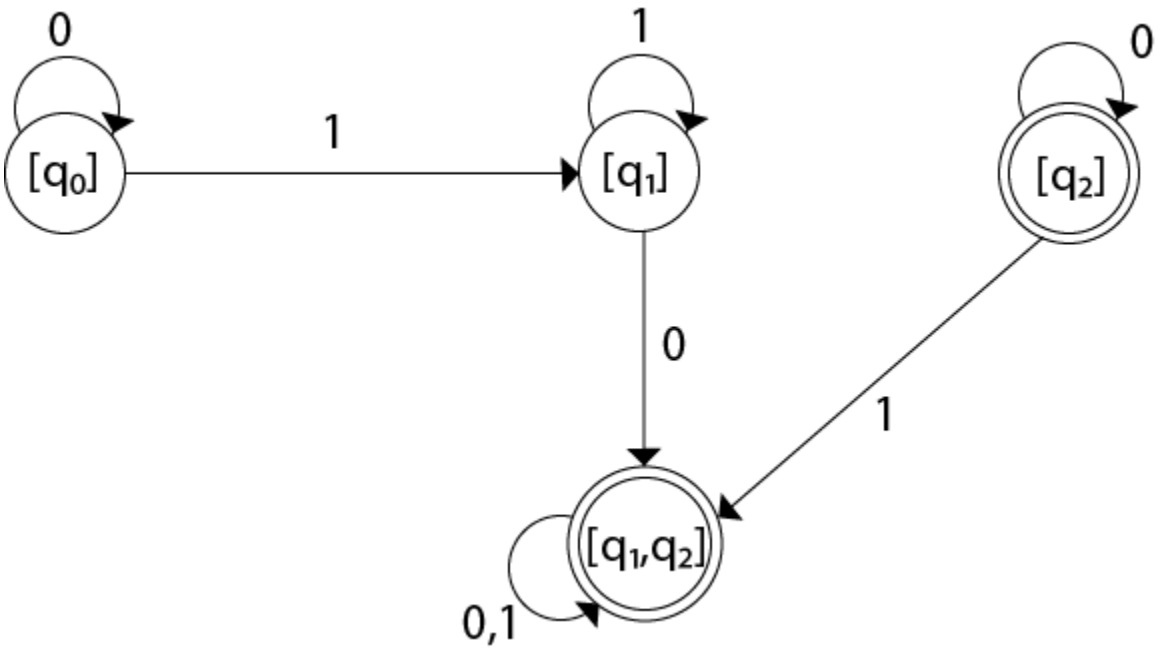
Now we will obtain δ' transition on [q1, q2].

1. δ'([q1, q2], 0) = δ(q1, 0) ∪ δ(q2, 0)
2. = {q1, q2} ∪ {q2}
3. = [q1, q2]
4. δ'([q1, q2], 1) = δ(q1, 1) ∪ δ(q2, 1)
5. = {q1} ∪ {q1, q2}
6. = {q1, q2}
7. = [q1, q2]

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

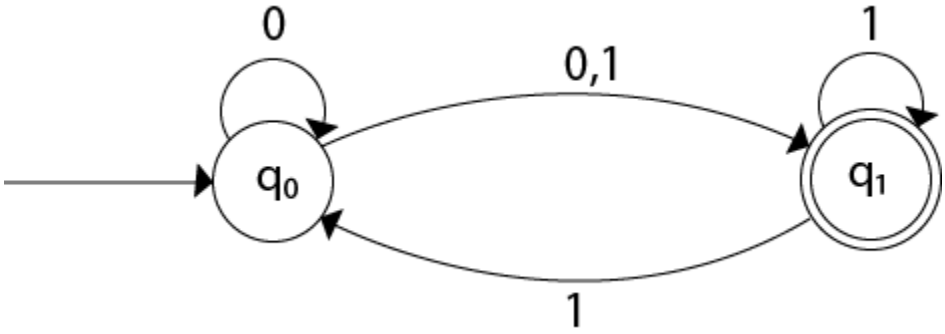
The Transition diagram will be:



The state q2 can be eliminated because q2 is an unreachable state.

Example 2:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	{q0, q1}	{q1}
*q1	ϕ	{q0, q1}

Now we will obtain δ' transition for state q0.

1. δ'([q0], 0) = {q0, q1}
2. = [q0, q1] (**new** state generated)
3. δ'([q0], 1) = {q1} = [q1]

The δ' transition for state q1 is obtained as:

1. δ'([q1], 0) = ϕ
2. δ'([q1], 1) = [q0, q1]

Now we will obtain δ' transition on [q0, q1].

1. δ'([q0, q1], 0) = δ(q0, 0) ∪ δ(q1, 0)
2. = {q0, q1} ∪ ϕ
3. = {q0, q1}
4. = [q0, q1]

Similarly,

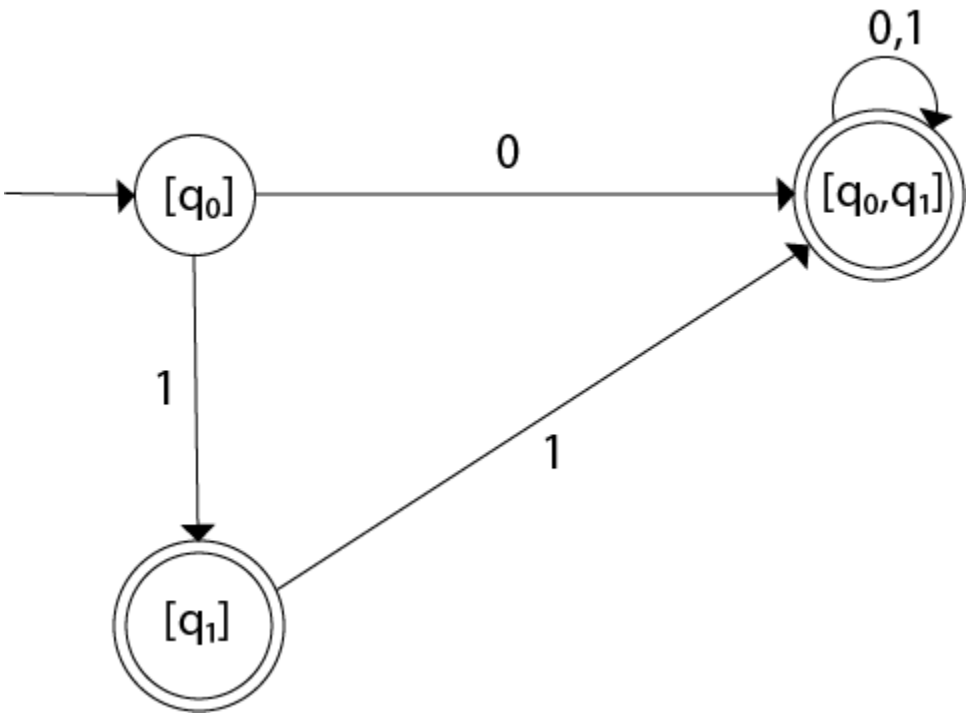
1. δ'([q0, q1], 1) = δ(q0, 1) ∪ δ(q1, 1)
2. = {q1} ∪ {q0, q1}
3. = {q0, q1}
4. = [q0, q1]

As in the given NFA, q1 is a final state, then in DFA wherever, q1 exists that state becomes a final state. Hence in the DFA, final states are [q1] and [q0, q1]. Therefore set of final states F = {[q1], [q0, q1]}.

The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0, q1]	[q1]
*[q1]	ϕ	[q0, q1]
*[q0, q1]	[q0, q1]	[q0, q1]

The Transition diagram will be:

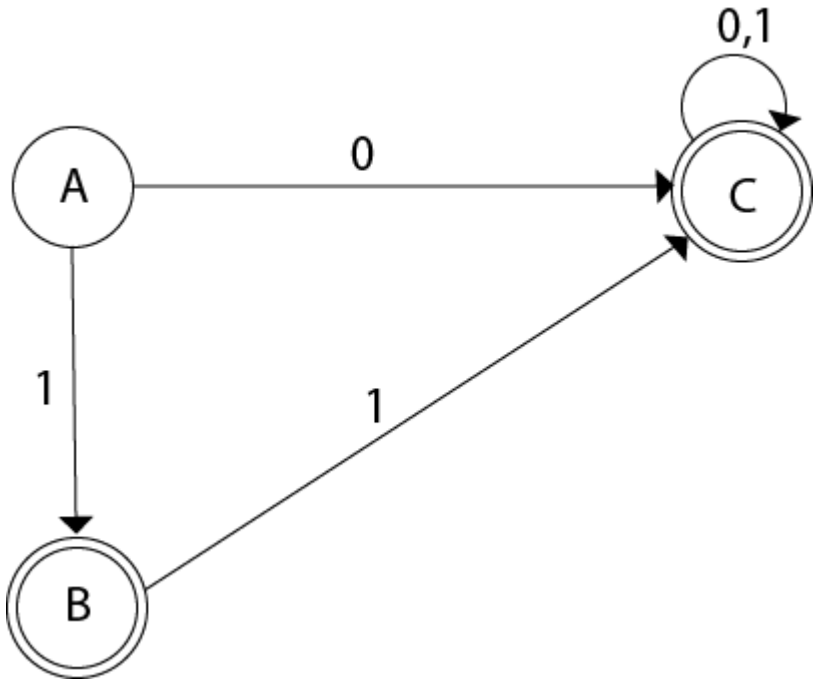


Even we can change the name of the states of DFA.

Suppose

1. A = [q0]
2. B = [q1]
3. C = [q0, q1]

With these new names the DFA will be as follows:



(b) Using pumping lemma for CFL prove that below languages are not context free

{p | p is a prime}. (5 Marks)

The pumping lemma for context-free languages states that, for any context-free language L and any sufficiently long string w in L, there exists a number n, called the "pumping length," such that w can be broken down into three substrings, w = xyz, where the following conditions hold:

1. **|y| > 0 and |xy| <= n**
2. **For any integer i >= 0, xy^i z is in L.**

In order to prove that a language is not context-free, you can use the pumping lemma by assuming that it is context-free and then finding a string that contradicts the conditions of the lemma.

For example, if you have a language L, and you want to prove it is not context-free. You can pick a string w in L that is longer than the pumping length n. Now you can break it into x,y,z such that |y|>0 and |xy|<=n. Now if you repeat y i times and

concatenate it with x,z. The resulting string should be in L if L is context-free. However, if you find a string that doesn't satisfy this property, it implies that the language is not context-free.

In other words, if you can prove that for some string w in L, there is no way to divide it into x, y, and z such that the conditions of the pumping lemma hold, then you have proven that L is not a context-free language.

Q2: (a) Explain Programming techniques for Turning Machines (5 Marks)

Programming Techniques for Turing Machines

- The following programmming techniques can be used to make the behavior of a TM clearer but none of these techniques adds any additional computational power to a basic TM.
    - Storage in the state
        - We can make a state a tuple with a fixed number of fixed-size components.
        - Components of the tuple can hold a fixed amount of data to simplify the behavior of a TM program.
    - Multiple tracks
        - We can make each symbol of the input alphabet a tuple with a fixed number of fixed-size components.
        - Components of the tuple can hold marks and other other information to indicate that an input tape square has been previously visited or has been given some fixed value.
    - Sets of states as subroutines
        - We can group states into "subroutines" where a subroutine has its own start state and another state which can serve as a "return" state.

(b) Obtain a grammar to generate the following language L ={WWR Where W {a, b}*}.(5 Marks)

Problem: Design a non deterministic PDA for accepting the language L = {wwR w $\in$ $(a, b)^+$}, i.e.,
L = {aa, bb, abba, aabbaa, abaaba, ...}

Prerequisite – Pushdown Automata, Pushdown Automata Acceptance by Final State
Explanation: In this type of input string, one input has more than one transition states, hence it is called non-deterministic PDA, and the input string contain any order of 'a' and 'b'. Each input alphabet has more than one possibility to move next state. And finally when the stack is empty then the string is accepted by the NPDA.
In this NPDA we used some symbols which are given below:
Γ = { a, b, z }

Where, Γ = set of all the stack alphabet
z = stack start symbol
a = input alphabet
b = input alphabet

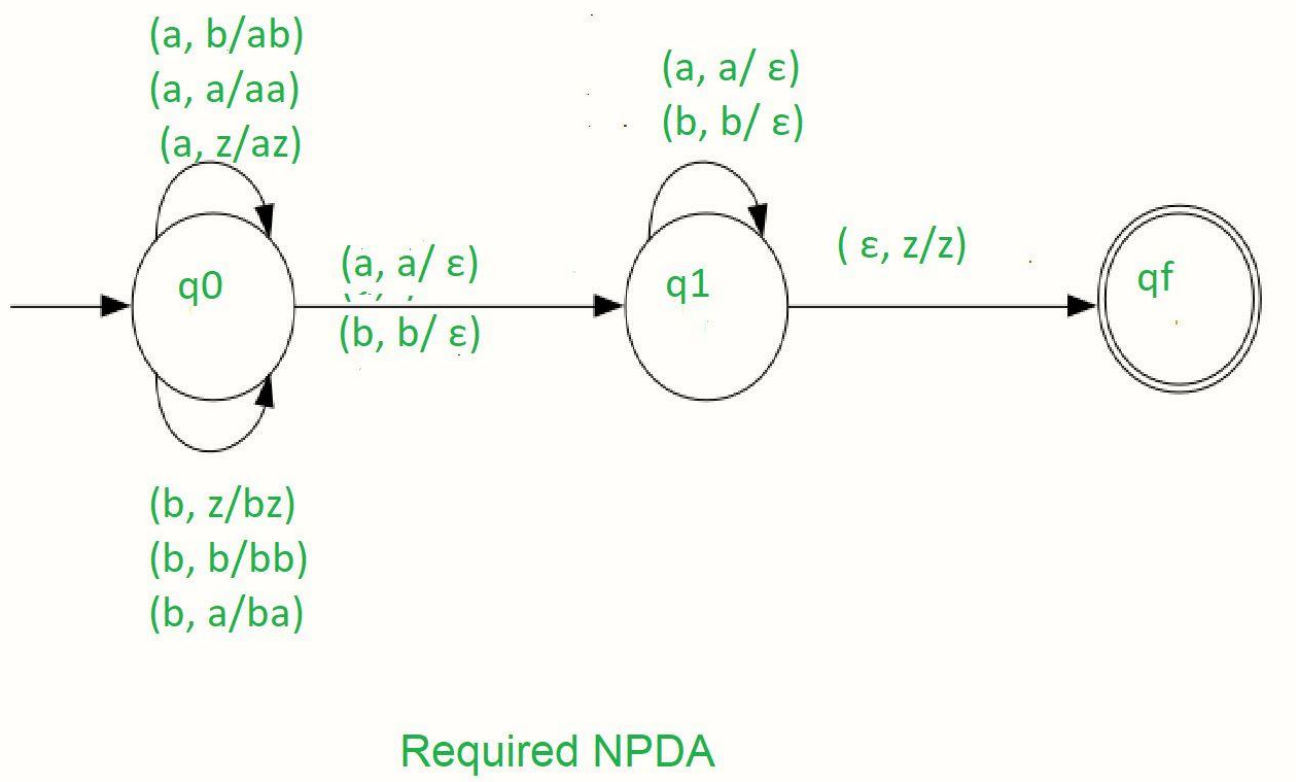
The approach used in the construction of PDA –

As we want to design an NPDA, thus every times 'a' or 'b' comes then either push into the stack or move into the next state. It is dependent on a string. When we see the input alphabet which is equal to the top of the stack then that time pop operation applies on the stack and move to the next step.
So, in the end, if the stack becomes empty then we can say that the string is accepted by the PDA.

STACK Transition Function

(q0, a, z) (q0, az) (q0, a, a) (q0, aa) (q0, b, z) (q0, bz) (q0, b, b) (q0, bb) (q0, a, b) (q0, ab) (q0, b, a) (q0, ba) (q0, a, a) (q1, ∈) (q0, b, b) (q1, ∈) (q1, a, a) (q1, ∈) (q1, b, b) (q1, ∈) (q1, ∈, z) (qf, z)

Where, q0 = Initial state
qf = Final state
∈ = indicates pop operation



So, this is our required non-deterministic PDA for accepting the language L = {wwR w ∈ (a, b)*}

Example:
We will take one input string: "abbbba".

- Scan string from left to right
- The first input is 'a' and follows the rule:
- on input 'a' and STACK alphabet Z, push the two 'a's into STACK as: (a, Z/aZ) and state will be q0
- on input 'b' and STACK alphabet 'a', push the 'b' into STACK as: (b, a/ba) and state will be q0
- on input 'b' and STACK alphabet 'b', push the 'b' into STACK as: (b, b/bb) and state will be q0
- on input 'b' and STACK alphabet 'b' (state is q1), pop one 'b' from STACK as: (b, b/∈) and state will be q1
- on input 'b' and STACK alphabet 'b' (state is q1), pop one 'b' from STACK as: (b, b/∈) and state will be q1
- on input 'a' and STACK alphabet 'a' and state q1, pop one 'a' from STACK as: (a, a/∈) and state will remain q1
- on input ∈ and STACK alphabet Z, go to the final state(qf) as : (∈, Z/Z)

So, at the end the stack becomes empty then we can say that the string is accepted by the PDA.

NOTE: ***This DPDA will not accept the empty language*****.**

Problem:

Design a deterministic PDA for accepting the language L = { wcwR w ∈ (a, b)*}, i.e.,

{aca, bcb, abcba, abacaba, aacaa, bbcbb, ...}

In each string, the substring which is present on the left side of c is the reverse of the substring which is the present right side of c.

Explanation:

Here we need to maintain string in such a way that, the substring which is present on the left side of c is exactly the reverse substring which is the right side of c. For doing this we used a stack. In string ‘a’ and ‘b’ are present any order and ‘c’ come only one time. When ‘c’ comes then the pop operation is started into the stack. And when a stack is empty then language is accepted.

Γ = {a, b, z}

Where, Γ = set of all the stack alphabet
z = stack start symbol
a = input alphabet
b = input alphabet

The approach used in the construction of PDA:

As we want to design PDA In every time when ‘a’ or ‘b’ comes we push into the stack and stay on the same state q0. And when ‘c’ comes then we move to the next state q1 without pushing ‘c’ into the stack. And after when comes an input which is the same as the top of the stack then pop from the stack and stay on the same state. POP operation is performed until the input string is ended. Finally when the input is ∈, then move to the final state qf.
When if the stack will become empty then the language is accepted.

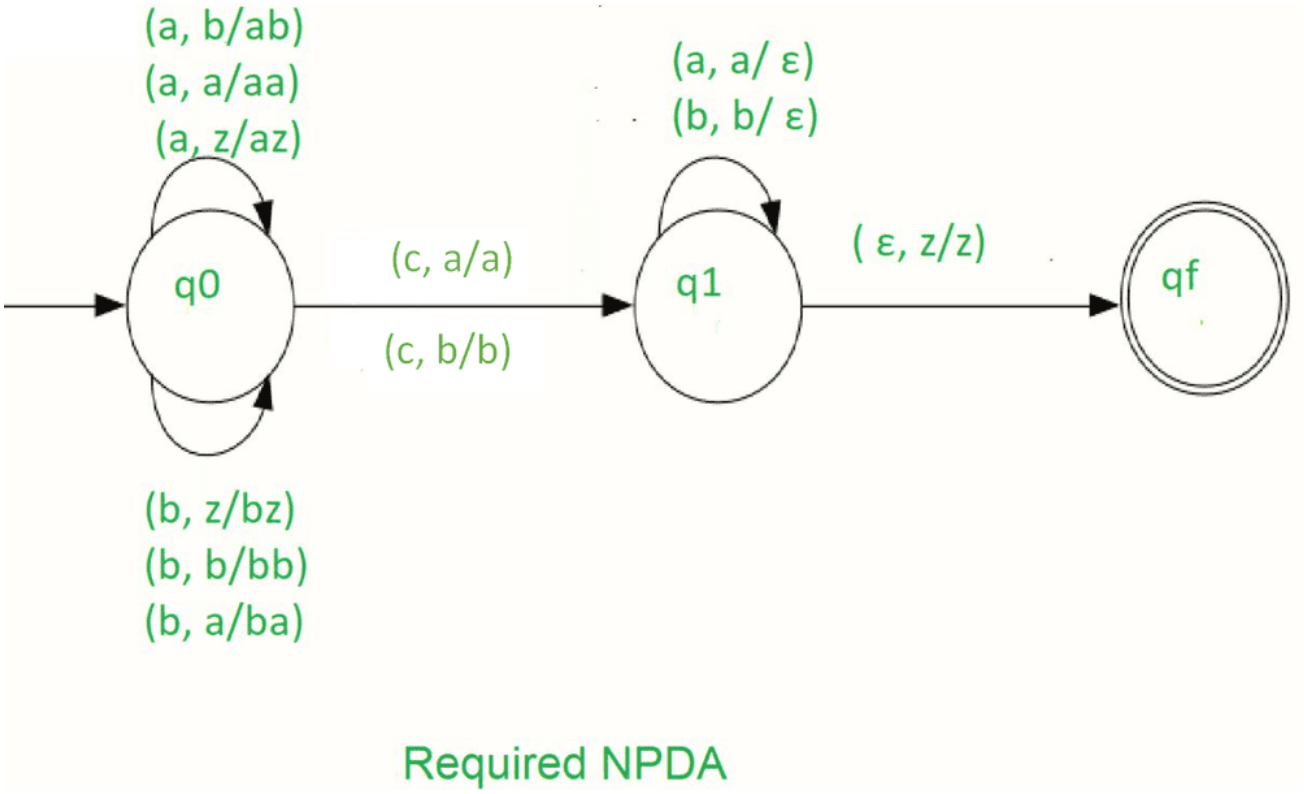
Where, q0 = Initial state
qf = Final state
z = stack start symbol
∈= indicates pop operation

Stack transition functions:

(q0, a, z) (q0, az) (q0, a, a) (q0, aa) (q0, b, z) (q0, bz) (q0, b, b) (q0, bb) (q0, a, b) (q0, ab) (q0, b, a) (q0, ba) (q0, c, a) (q1, a) (q0, c, b) (q1, b) (q1, a, a) (q1, ∈) (q1, b, b) (q1, ∈) (q1, ∈, z) (qf, z)

Where, q0 = Initial state
qf = Final state

∈ = indicates pop operation



Required NPDA

So, this is our required deterministic PDA for accepting the language,

L = { wcwR w ∈ (a, b)*}

Level Up Your GATE Prep!
Embark on a transformative journey towards GATE success by choosing Data Science & AI as your second paper choice with our specialized course. If you find yourself lost in the vast landscape of the GATE syllabus, our program is the compass you need.

Q3: (a) State decision properties of regular language. (5 Marks)

Properties of Regular Languages

So far we have seen different ways of specifying regular language: DFA, NFA, ε-NFA, regular expressions and regular grammar. We noted that all these different expressions are equal in power by showing the equivalences. Regular expressions

and grammars are considered as generators of regular language while the machines (DFA, NFA, ε-NFA) are considered as acceptors of the language.

Now we will look at the properties of regular language. The properties can be broadly classified as two parts: (A) Closure properties and (B) Decision properties

(A) Closure Properties

1. Complementation

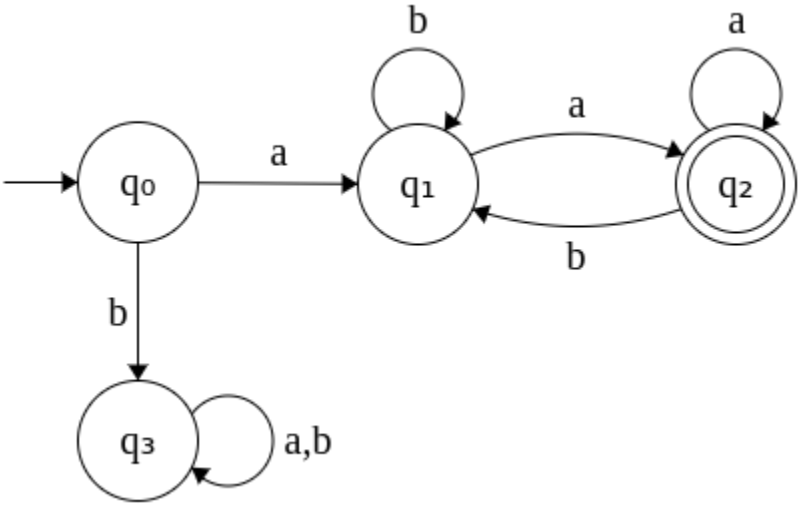
If a language L is regular its complement L' is regular.

Let DFA(L) denote the DFA for the language L. Modify the DFA as follows to obtain DFA(L').

1. Change the final states to non-final states.
2. Change the non-final states to final states.

Since there exists a DFA(L') now, L' is regular.

This can be shown by an example using a DFA. Let L denote the language containing strings that begins and ends with a. $\Sigma = \{a, b\}$. The DFA for L is given below.

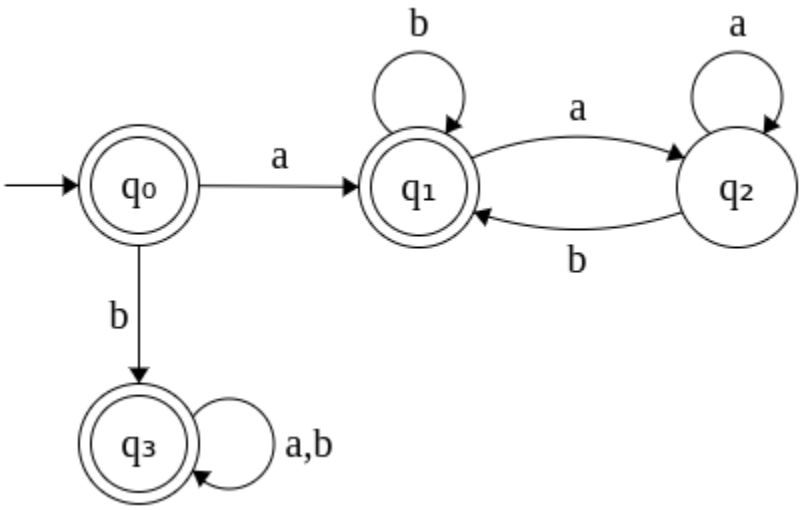


Note: q_3 denotes the dead state. Once you enter q_3, you remain in it forever.

L' denotes the language that does not contain strings that begin and end with a. This implies L' contains strings that

- begins with a and ends with b
- begins with b and ends with a
- begins with b and ends with b

The DFA for L' is obtained by flipping the final states of DFA(L) to non-final states and vice-versa. The DFA for L' is given below.



- q_0 ensures ε is accepted
- q_1 ensures all strings that begin with a and end with b are accepted.
- q_3 ensures all strings that begin with b (ending with either a or b) are accepted.

Important Note: While specifying the DFA for L, we have also included the dead state q_3. It is important to include the dead state(s) if we are going to derive the complement DFA since, the dead state(s) too would become final in the complementation. If we didn't add the dead state(s) originally, the complement will not accept all strings supposed to be accepted.

In the above example, if we didn't include q_3 originally, the complement will not accept strings starting with b. It will only accept strings that begin with a and end with b which is only a subset of the complement.

CONCLUSION: REGULAR LANGUAGES ARE CLOSED UNDER COMPLEMENTATION.

2. Union

If L_1 and L_2 are regular, then $L_1 \cup L_2$ is regular.

This is easier proved using regular expressions. If L_1 is regular, there exists a regular expression R1 to describe it. Similarly, if L_2 is regular, there exists a regular expression R2 to describe it. R1 + R2 denotes the regular expression that describe $L_1 \cup L_2$. Therefore, $L_1 \cup L_2$ is regular.

This again can be shown using an example. If L_1 is a language that contains strings that begin with a and L_2 is a language that contain strings that end with a, then $L_1 \cup L_2$ denotes the language the contain strings that either begin with a or end with a.

- a(a+b)* is the regular expression that denotes L_1.

- (a+b)*a is the regular expression that denotes L_2.

- $L_1 \cup L_2$ is denoted by the regular expression a(a+b)* + (a+b)*a. Therefore, $L_1 \cup L_2$ is regular.

In terms of DFA, we can say that a DFA($L_1 \cup L_2$) accepts those strings that are accepted by either DFA(L_1) or DFA(L_2) or both.

- DFA($L_1 \cup L_2$) can be constructed by adding a new start state and new final state.
- The new start state connects to the two start states of DFA(L_1) and DFA(L_2) by εtransitions.
- Similarly, two ε transitions are added from the final states of DFA(L_1) and DFA(L_2) to the new final state.
- Convert this resulting NFA to its equivalent DFA.

As an exercise you can try this approach of DFA construction for union for the given example.

CONCLUSION: REGULAR LANGUAGES ARE CLOSED UNDER UNION.

3. Intersection

If L_1 and L_2 are regular, then $L_1 \cap L_2$ is regular.

Since a language denotes a set of (possibly infinite) strings and we have shown above that regular languages are closed under union and complementation, by De Morgan's law can be applied to show that regular languages are closed under intersection too.

L_1 and L_2 are regular $\Rightarrow$ L_1' and L_2' are regular (by Complementation property)
$L_1' \cup L_2'$ is regular (by Union property)
$L_1 \cap L_2$ is regular (by De Morgan's law)

In terms of DFA, we can say that a DFA($L_1 \cap L_2$) accepts those strings that are accepted by both DFA(L_1) and DFA(L_2).

CONCLUSION: REGULAR LANGUAGES ARE CLOSED UNDER INTERSECTION.

4. Concatenation

If L_1 and L_2 are regular, then $L_1 . L_2$ is regular.

This can be easily proved by regular expressions. If R1 is a regular expression denoting L_1 and R2 is a regular expression denoting L_2, then we R1 . R2 denotes the regular expression denoting $L_1 . L_2$. Therefore, $L_1 . L_2$ is regular.

In terms of DFA, we can say that a DFA(L_1 . L_2) can be constructed by adding an ε-trainstion from the final state of DFA(L_1) - which now ceases to be the final state - to the start state of DFA(L_2). You can try showing this using an example.

(b) Design a DFA to accept string of 0's & 1's when interpreted as binary numbers would be multiple of 3. (5 Marks)

Given a string of binary characters, check if it is multiple of 3 or not.
Examples :
Input : 1 0 1 0

Output : NO

Explanation : (1 0 1 0) is 10 and hence

not a multiple of 3

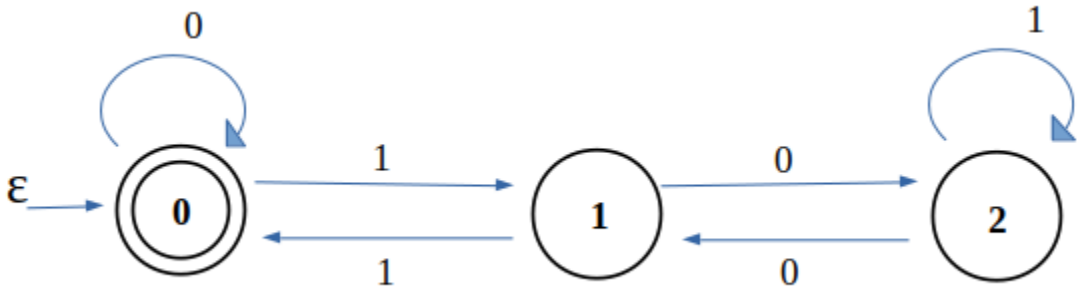
Input : 1 1 0 0

Output : YES

Explanation : (1 1 0 0) is 12 and hence

a multiple of 3

Approach 1 : One simple method is to convert the binary number into its decimal representation and then check if it is a multiple of 3 or not. Now, when it comes to **DFA (Deterministic Finite Automata)**, there is no concept of memory i.e. you cannot store the string when it is provided, so the above method would not be applicable. In simple terms, a DFA takes a string as input and process it. If it reaches final state, it is accepted, else rejected. As you cannot store the string, so input is taken character by character.
The DFA for given problem is :



As, when a number is divided by 3, there are only 3 possibilities. The remainder can be either 0, 1 or 2. Here, state 0 represents that the remainder when the number is divided by 3 is 0. State 1 represents that the remainder when the number is divided by 3 is 1 and similarly state 2 represents that the remainder when the number is divided by 3 is 2. So if a string reaches state 0 in the end, it is accepted otherwise rejected.

Below is the implementation of above approach :

C++

```
// C++ Program to illustrate
// DFA for multiple of 3
#include <bits/stdc++.h>
using namespace std;

// checks if binary characters
// are multiple of 3
bool isMultiple3(char c[], int size)
{
    // initial state is 0th
    char state = '0';

    for (int i = 0; i < size; i++) {

        // storing binary digit
        char digit = c[i];

        switch (state) {

        // when state is 0
```

```
        case '0':
            if (digit == '1')
                state = '1';
            break;

        // when state is 1
        case '1':
            if (digit == '0')
                state = '2';
            else
                state = '0';
            break;

        // when state is 2
        case '2':
            if (digit == '0')
                state = '1';
            break;
        }
    }

    // if final state is 0th state
    if (state == '0')
        return true;
    return false;
}

// Driver's Code
int main()
{
    // size of binary array
    int size = 5;

    // array of binary numbers
    // Here it is 21 in decimal
    char c[] = { '1', '0', '1', '0', '1' };

    // if binary numbers are a multiple of 3
    if (isMultiple3(c, size))
```

```
        cout << "YES\n";
    else
        cout << "NO\n";

    return 0;
}
```

Java
Python3
C#
PHP
Javascript

Output
YES

Time Complexity: O(n)
Auxiliary Space: O(1)
Approach 2: We will check if the difference between the number of non-zero odd bit positions and non-zero even bit positions is divisible by 3 or not.
Mathematically -> |odds-evens| divisible by 3.

Code:

C++

```
// C++ program to check if the binary string is divisible
// by 3.
#include <bits/stdc++.h>
using namespace std;
// Function to check if the binary string is divisible by 3.
void CheckDivisibilty(string A)
{
    int oddbits = 0, evenbits = 0;
    for (int counter = 0; counter < A.length(); counter++) {
        // checking if the bit is nonzero
        if (A[counter] == '1') {
            // checking if the nonzero bit is at even
            // position
            if (counter % 2 == 0) {
                evenbits++;
```

```
            }
            else {
                oddbits++;
            }
        }
    }
    // Checking if the difference of non-zero oddbits and
    // evenbits is divisible by 3.
    if (abs(oddbits - evenbits) % 3 == 0) {
        cout << "Yes" << endl;
    }
    else {
        cout << "No" << endl;
    }
}
// Driver Program
int main()
{
    string A = "10101";
    CheckDivisibilty(A);
    return 0;
}
```

Java
Python3
C#
Javascript

Output
Yes

Time Complexity: O(n) where n is the number of bits.
Auxiliary Space: O(1)

Feeling lost in the world of random DSA topics, wasting time without progress? It's time for a change! Join our DSA course, where we'll guide you on an exciting journey to master DSA efficiently and on schedule.
Ready to dive in? Explore our Free Demo Content and join our DSA course, trusted by over 100,000 geeks!

Q4: Define the following: (10 Marks)

- Application of context free grammar.

Context Free Grammar is formal grammar, the syntax or structure of a formal language can be described using context-free grammar (CFG), a type of formal grammar. The grammar has four tuples: (V,T,P,S).
V - It is the collection of variables or nonterminal symbols.
T - It is a set of terminals.
P - It is the production rules that consist of both terminals and nonterminals.
S - It is the Starting symbol.
A grammar is said to be the Context-free grammar if every production is in the form of :
G -> (V∪T)*, where G ∈ V

- And the left-hand side of the G, here in the example can only be a Variable, it cannot be a terminal.
- But on the right-hand side here it can be a Variable or Terminal or both combination of Variable and Terminal.

Above equation states that every production which contains any combination of the 'V' variable or 'T' terminal is said to be a context-free grammar.
For example the grammar A = { S, a,b, P,S} having production :

- Here S is the starting symbol.
- {a,b} are the terminals generally represented by small characters.
- P is variable along with S.

S-> aS
S-> bSa
but
a->bSa, or
a->ba is not a CFG as on the left-hand side there is a variable which does not follow the CFGs rule.
In the computer science field, context-free grammars are frequently used, especially in the areas of formal language theory, compiler development, and natural language processing. It is also used for explaining the syntax of programming languages and other formal languages.

Limitations of Context-Free Grammar

Apart from all the uses and importance of Context-Free Grammar in the Compiler design and the Computer science field, there are some limitations that are addressed, that is CFGs are less expressive, and neither English nor programming language can be expressed using Context-Free Grammar. Context-Free Grammar can be ambiguous means we can generate multiple parse trees of the same input. For some grammar,

Context-Free Grammar can be less efficient because of the exponential time complexity. And the less precise error reporting as CFGs error reporting system is not that precise that can give more detailed error messages and information.