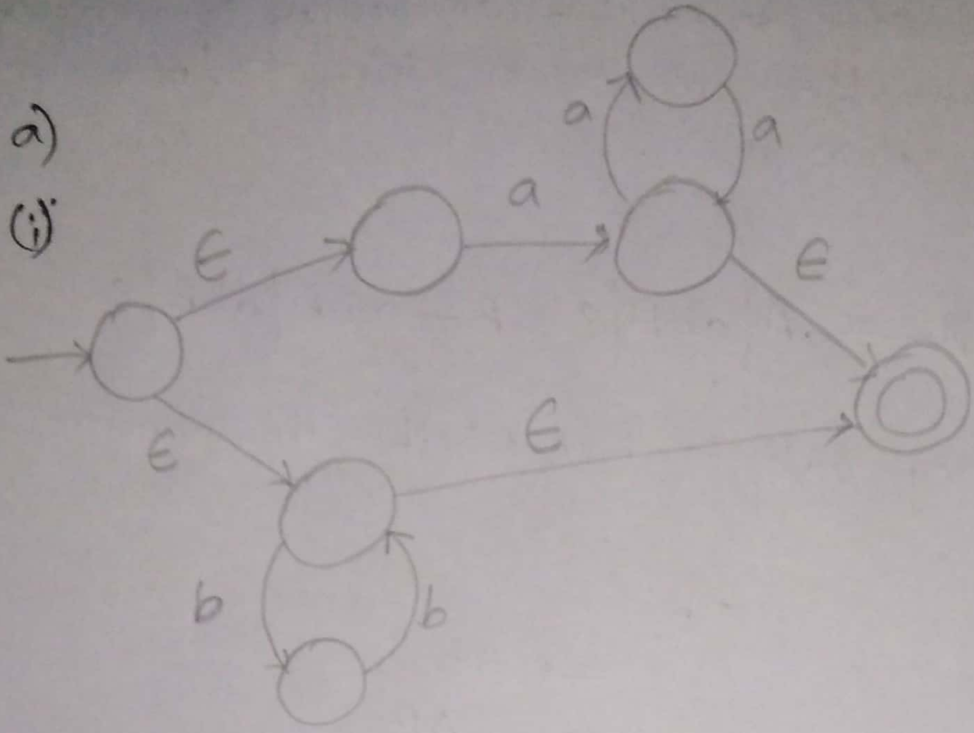
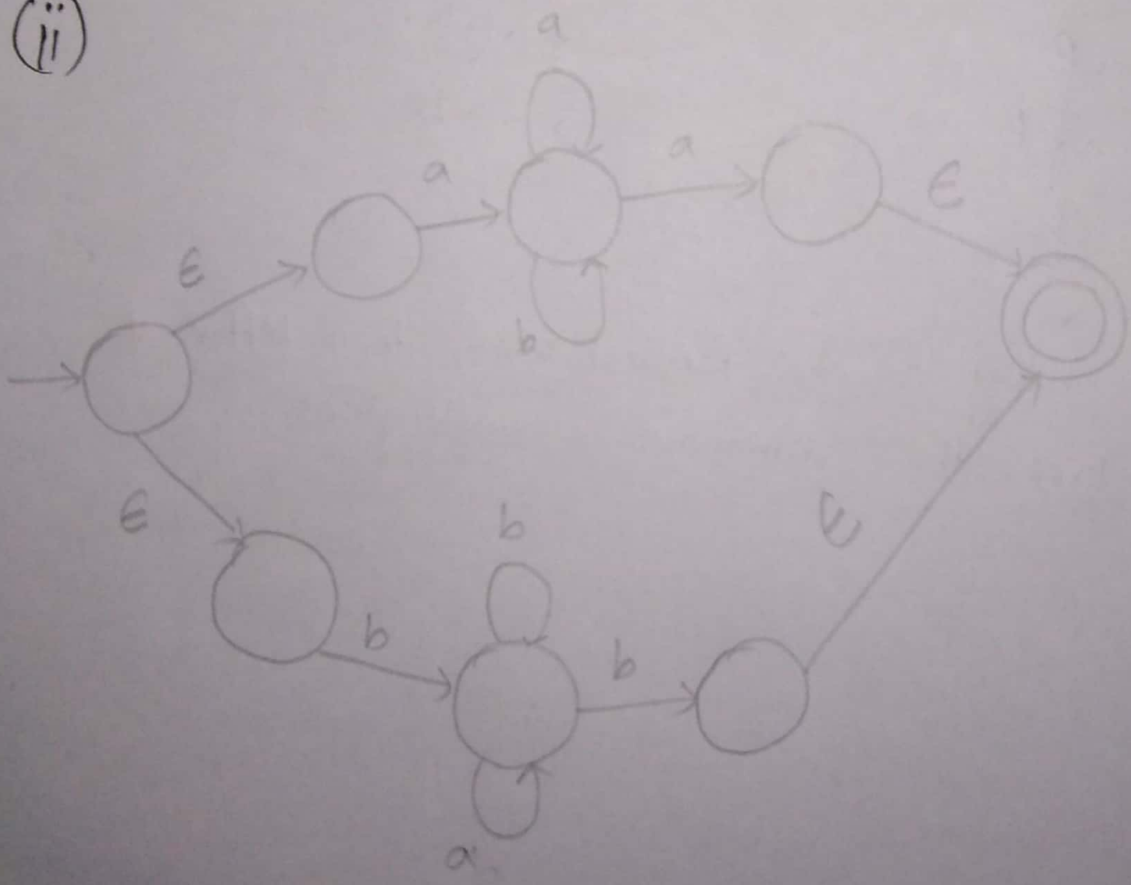
1. a)

(i)



(ii)



b) A grammar is said to exhibit ambiguity if there exists two or more derivation tree for a string ~~to~~ (means two or more left derivation tree).

Given grammar,

$S \rightarrow AB \qquad A \rightarrow aA \mid abA \mid \varepsilon \qquad B \rightarrow bB \mid abB \mid \varepsilon$

Let us take string 'aab' to check whether the grammar is ambigiuos.

Try 1

$$\begin{aligned} S &\rightarrow AB \\ &\rightarrow aAB \\ &\rightarrow aaAB \\ &\rightarrow aaB \\ &\rightarrow aabB \\ &\rightarrow aab \end{aligned}$$

Try 2

$$\begin{aligned} S &\rightarrow AB \\ &\rightarrow aAB \\ &\rightarrow aB \\ &\rightarrow aabB \\ &\rightarrow aab \end{aligned}$$

Since the string can be formed using two left-most derivation tree, the grammar is ambigiuos.

c) $S \rightarrow AB$

$\rightarrow aAB$ (using $A \rightarrow aA$)

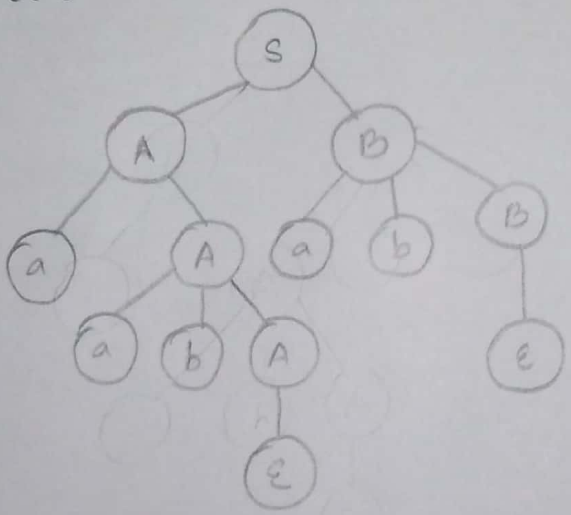
$\rightarrow aabAB$ (using $A \rightarrow abA$)

$\rightarrow aabB$ (using $A \rightarrow \varepsilon$)

$\rightarrow aababB$ (using $B \rightarrow abB$)

$\rightarrow aabab$ (using $B \rightarrow \varepsilon$).

Parse tree:



OR ⇒

c) $S \rightarrow AB$

$S \rightarrow AabB$ (using $B \rightarrow abB$)

$S \rightarrow AabbB$ (using $B \rightarrow bB$)

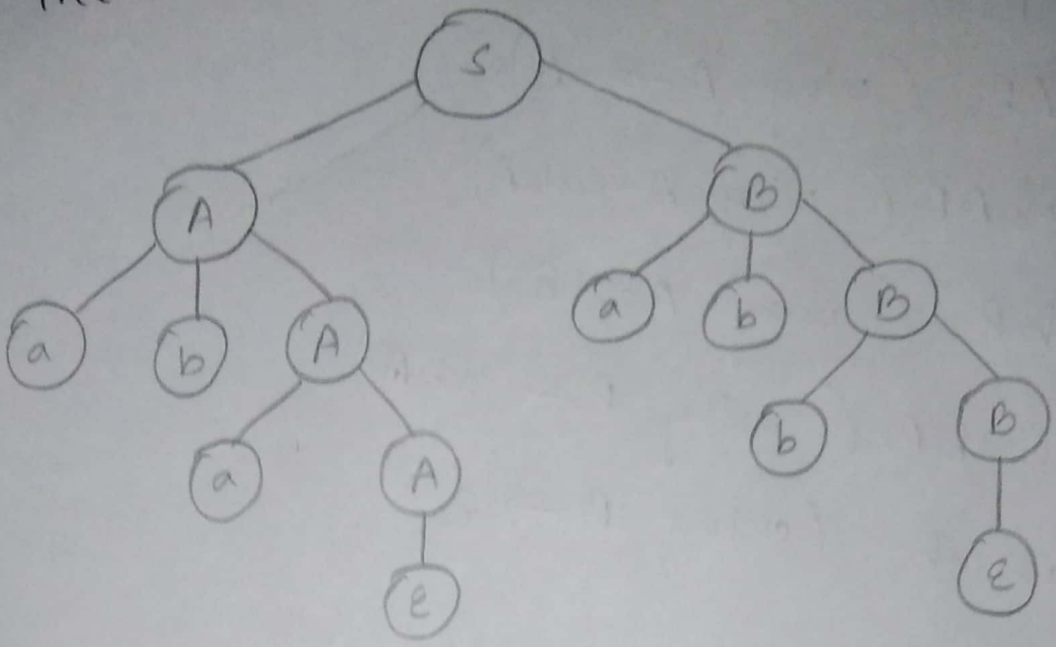
$S \rightarrow Aabb$ (using $B \rightarrow \varepsilon$)

$S \rightarrow abAabb$ (using $A \rightarrow abA$)

$S \rightarrow abaAabb$ (using $A \rightarrow aA$)

$S \rightarrow abaabb$ (using $A \rightarrow \varepsilon$)

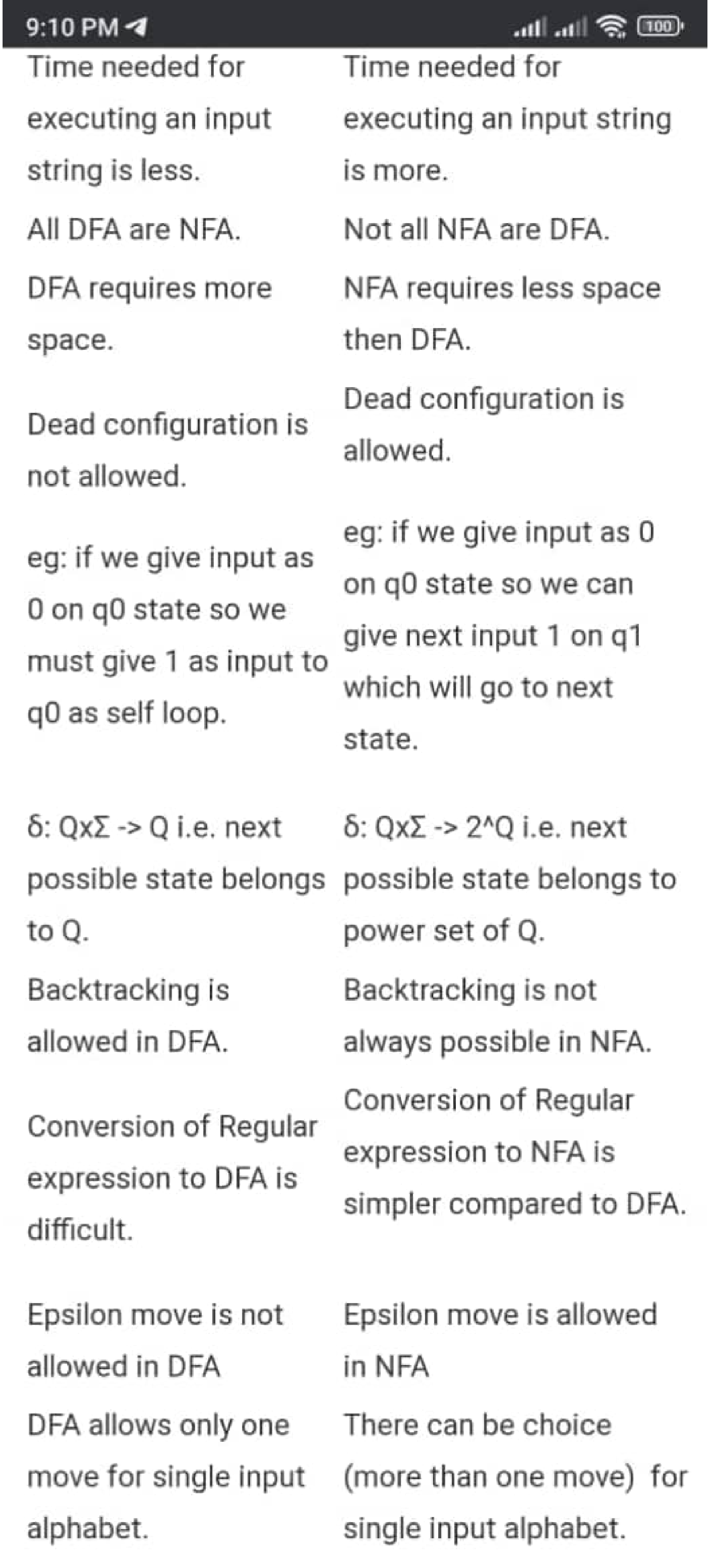
Parse tree :



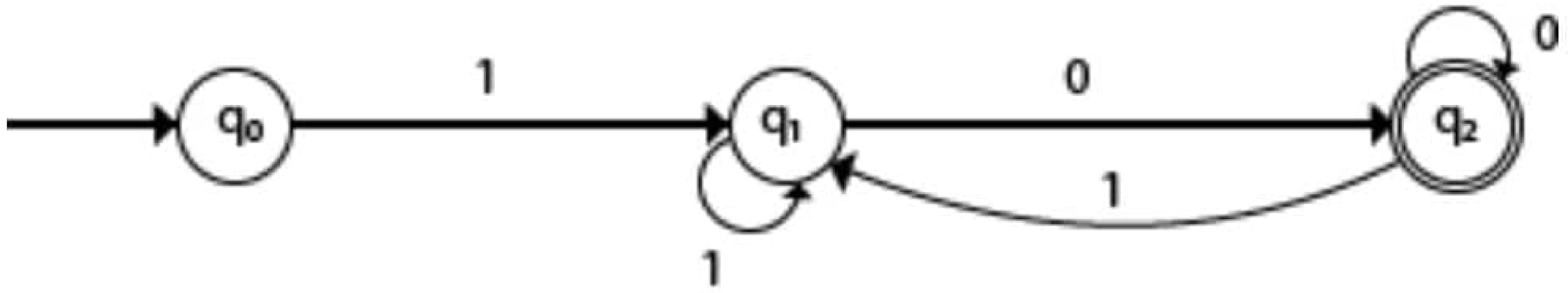
Difference between DFA and NFA :

2(a)

DFA	NFA
DFA stands for Deterministic Finite Automata.	NFA stands for Nondeterministic Finite Automata.
For each symbolic representation of the alphabet, there is only one state transition in DFA.	No need to specify how does the NFA react according to some symbol.
DFA cannot use Empty String transition.	NFA can use Empty String transition.
DFA can be understood as one machine.	NFA can be understood as multiple little machines computing at the same time.
In DFA, the next possible state is distinctly set.	In NFA, each pair of state and input symbol can have many possible next states.
DFA is more difficult to construct.	NFA is easier to construct.
DFA rejects the string in case it terminates in a state that is different from the accepting state.	NFA rejects the string in the event of all branches dying or refusing the string.

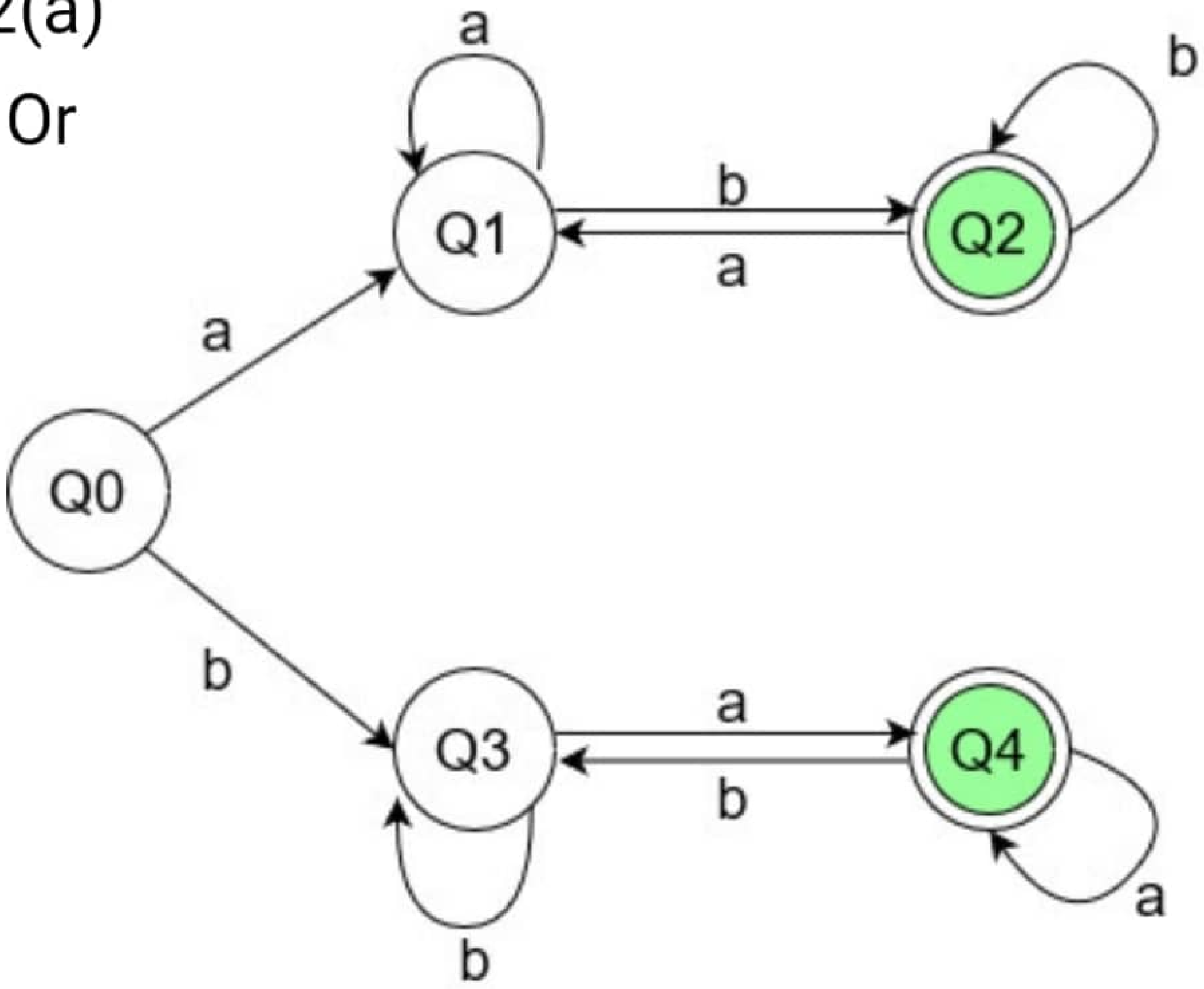


Time needed for executing an input string is less.	Time needed for executing an input string is more.
All DFA are NFA.	Not all NFA are DFA.
DFA requires more space.	NFA requires less space then DFA.
Dead configuration is not allowed.	Dead configuration is allowed.
eg: if we give input as 0 on q0 state so we must give 1 as input to q0 as self loop.	eg: if we give input as 0 on q0 state so we can give next input 1 on q1 which will go to next state.
δ: QxΣ -> Q i.e. next possible state belongs to Q.	δ: QxΣ -> 2^Q i.e. next possible state belongs to power set of Q.
Backtracking is allowed in DFA.	Backtracking is not always possible in NFA.
Conversion of Regular expression to DFA is difficult.	Conversion of Regular expression to NFA is simpler compared to DFA.
Epsilon move is not allowed in DFA	Epsilon move is allowed in NFA
DFA allows only one move for single input alphabet.	There can be choice (more than one move) for single input alphabet.


q₀
q₁
q₂
1
0
1
1
0

2(a)

Or



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

2(b)

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 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

3a.

To show that the language L = {w | w belongs to a^r * b^n * b^n} is not context-free, we can use the pumping lemma for context-free languages. The pumping lemma states that every context-free language L has a pumping length p such that any string s in L of length |s| >= p can be written as s = uvxyz, where u, v, x, y, z are strings such that:

1.|vxy| <= p

2.|vy| >= 1

3.for all i >= 0, the string uv^ixy^iz is also in L

Assume that L is a context-free language, and let p be the pumping length given by the pumping lemma for L. Consider the string s = a^p b^p b^p in L, where |s| = 3p >= p.

By the pumping lemma, we can write s as s = uvxyz, where |vxy| <= p and |vy| >= 1. We can choose v and y such that they consist entirely of b's, and vxy consists of at most two blocks of b's, one of length k and one of length l, where k + l <= p. Then u and z must consist entirely of a's, and x must consist of b's.

Since v and y consist entirely of b's, the substring vxy consists of b's only, and can be pumped any number of times without affecting the number of a's. Therefore, for any i >= 0, the string uv^ixy^iz will have the form a^(p + i) b^p b^p, which is not in L since the number of b's in the second block is different from the number of b's in the third block. This contradicts the pumping lemma assumption that all strings uv^ixy^iz are in L.

Thus, we have shown that the language L = {w | w belongs to a^r * b^n * b^n} is not context-free by contradiction.

3b.(All the questions are solved,but we have to write any one from below according to the question)

i)

To convert the given context-free grammar into an equivalent grammar in Chomsky normal form, we need to follow the following steps:

Step 1: Eliminate ε-productions if any.

There are no ε-productions in the given grammar.

Step 2: Eliminate unit productions if any.

There are no unit productions in the given grammar.

Step 3: Convert all the non-terminal symbols into binary.

We don't need to convert any non-terminal symbols because they are already binary.

Step 4: Replace all the productions of length greater than 2.

We need to replace the following productions of length greater than 2:

R --> aRb

R --> bRb

We can introduce a new non-terminal symbol C and replace the above productions as follows:

R --> CR

C --> a

C --> b

Step 5: Introduce a new start symbol if required.

We don't need to introduce a new start symbol because the given grammar already has a start symbol S.

After applying all the above steps, the resulting grammar in Chomsky normal form is:

S --> RC | C

R --> CR

C --> a

C --> b

This grammar is equivalent to the original grammar and is now in Chomsky normal form.

ii)

To convert the context-free grammar (CFG)

S --> aTa | bTa

into Chomsky normal form (CNF), we need to follow these steps:

Step 1: Eliminate the start symbol from the right-hand side of any rule.

We do not need to do anything in this step since the start symbol S does not appear on the right-hand side of any rule.

Step 2: Eliminate any rules that generate ε (the empty string).

We do not have any rules that generate ε.

Step 3: Convert all rules of the form A → B to Chomsky normal form.

The grammar does not have any rules of this form.

Step 4: Convert all rules of the form A → BC to Chomsky normal form.

We can replace the rule S --> aTa with the following three rules:

S --> X1TY1

X1 --> a

T --> S

Y1 --> a

Similarly, we can replace the rule S --> bTa with the following three rules:

S --> X2TY2

X2 --> b

T --> S

Y2 --> a

After these replacements, the grammar becomes:

S --> X1TY1 | X2TY2

T --> S

X1 --> a

X2 --> b

Y1 --> a

Y2 --> a

Step 5: Convert all rules of the form A → a to Chomsky normal form.

We do not have any rules of this form.

Therefore, the equivalent grammar in Chomsky normal form is:

S --> X1TY1 | X2TY2

T --> S

X1 --> a

X2 --> b

Y1 --> a

Y2 --> a

where S and T are non-terminal symbols, and a and b are terminal symbols.

iii)

To convert the context-free grammar (CFG)

T --> XTX | X | ε

into Chomsky normal form (CNF), we need to follow these steps:

Step 1: Eliminate the start symbol from the right-hand side of any rule.

We do not need to do anything in this step since the start symbol T does not appear on the right-hand side of any rule.

Step 2: Eliminate any rules that generate ε (the empty string).

We can eliminate the rule T --> ε by adding a new start symbol S and the rule S --> T | ε. This will allow us to derive the empty string only by using the start symbol S.

S --> T | ε

T --> XTX | X

Step 3: Convert all rules of the form A → B to Chomsky normal form.

We do not have any rules of this form.

Step 4: Convert all rules of the form A → BC to Chomsky normal form.

We can replace the rule T --> XTX with the following three rules:

T --> UD

U --> X

D --> TX

After this replacement, the grammar becomes:

S --> T | ε

T --> UD | X

U --> X

D --> TX

Step 5: Convert all rules of the form A → a to Chomsky normal form.

We do not have any rules of this form.

Therefore, the equivalent grammar in Chomsky normal form is:

S --> T | ε

T --> UD | X

U --> X

D --> TX

where S is the new start symbol, T, U, and D are non-terminal symbols, and X is a terminal symbol.

iv)

To convert the context-free grammar (CFG)

X --> a | b

into Chomsky normal form (CNF), we need to follow these steps:

Step 1: Eliminate the start symbol from the right-hand side of any rule.

We do not need to do anything in this step since the start symbol X does not appear on the right-hand side of any rule.

Step 2: Eliminate any rules that generate ε (the empty string).

We do not have any rules that generate ε.

Step 3: Convert all rules of the form A → B to Chomsky normal form.

We do not have any rules of this form.

Step 4: Convert all rules of the form A → BC to Chomsky normal form.

We do not have any rules of this form.

Step 5: Convert all rules of the form A → a to Chomsky normal form.

We can replace the rule X --> a with a new non-terminal symbol Y and the rule Y --> a.

X --> Y | b

Y --> a

Step 6: Convert all rules of the form A → aB to Chomsky normal form.

We do not have any rules of this form.

Step 7: Convert all rules of the form A → BC to Chomsky normal form.

We do not have any rules of this form.

Step 8: Convert all rules of the form A → ε to Chomsky normal form.

We do not have any rules of this form.

Therefore, the equivalent grammar in Chomsky normal form is:

X --> Y | B

Y --> a

B --> b

where X and B are non-terminal symbols, Y is a new non-terminal symbol, and a and b are terminal symbols.

3(OR).

Theorem: A language is context-free if and only if some pushdown automata recognize it.

Proof:

Forward direction:

Let L be a context-free language. By definition, there exists a context-free grammar G = (V, Σ, R, S) such that L = L(G). We will construct a pushdown automaton P = (Q, Σ, Γ, δ, q0, Z, F) that recognizes L.

First, we need to create a new start state q0 and a new stack symbol Z. We set the initial state of the PDA to q0 and the initial stack symbol to Z.

Next, we need to define the transition function δ of the PDA. For each rule A → β in R, we add a transition (q, a, B) → (q', γ) to δ, where a ∈ Σ, B ∈ Γ, and β →* γ is a sequence of rules in R. This transition means that if the PDA reads input symbol a in state q with symbol B at the top of the stack, it can replace B with γ on the stack and transition to state q'.

Finally, we add an accepting state F to the PDA if the stack is empty when the PDA reaches the end of the input string.

It can be shown that this PDA recognizes the language L. Therefore, if a language is context-free, there exists some pushdown automaton that recognizes it.

Backward direction:

Let L be a language that is recognized by some pushdown automaton P = (Q, Σ, Γ, δ, q0, Z, F). We will construct a context-free grammar G = (V, Σ, R, S) such that L = L(G).

First, we create a new start symbol S and a new non-terminal symbol Aq for each state q in Q.

Next, we add the following rules to R:

- S → Aq, where q is the initial state of P

- Aq → aBpC, where (p, a, B) → (q, C) is a transition in P and B ∈ Γ, a ∈ Σ, and C ∈ Γ

- Aq → ε, where q ∈ F

It can be shown that any string in L is derivable by using the rules of G, and any string derivable by using the rules of G is in L. Therefore, if a pushdown automaton recognizes a language, the language is context-free.

Therefore, we have shown both directions of the proof, and the theorem is proved.

3b)

To remove null productions from the given grammar, we need to follow these steps:

Step 1: Identify all nullable variables (variables that can derive ε).

From the given grammar, we can see that A, B, and C are nullable.

Step 2: Remove all rules that generate ε, except for the start symbol.

From the given grammar, the rule A --> ε generates ε, but A is the start symbol. Therefore, we cannot remove this rule.

Step 3: For each remaining rule that contains a nullable variable A, create a new rule without A.

From the given grammar, we have two rules containing nullable variable A:

S --> ABAC

A --> aA | ε

We can create two new rules:

S --> BAC | BC | AC | C

A --> aA | a

Step 4: Replace any occurrences of the nullable variables with their new productions.

From the given grammar, we have two rules containing nullable variable B:

S --> BAC | BC | AC | C

B --> bB | b

We can create four new rules:

S --> AC | C | BC | BAC | BA | AC | A | C

B --> bB | b

Finally, we have one rule containing nullable variable C:

S --> AC | C | BC | BAC | BA | AC | A | C

C --> c

Therefore, the new grammar with null productions removed is:

S --> AC | C | BC | BAC | BA | AC | A | C

A --> aA | a

B --> bB | b

C --> c

3b(or).

i)

To convert the context-free grammar (CFG) into an equivalent pushdown automaton (PDA), we can follow these steps:

Given a CFG G = (V, Σ, R, S), where:

S → XRX | S

X → a | b

1. Create a new PDA P = (Q, Σ, Γ, δ, q0, Z, F), where:

   - Q = {q0, q1, q2, q3}

   - Σ = {a, b}

   - Γ = {X, R, Z}

   - F = {q3}

   - Z = X

   - q0 is the start state.

2. Define the transition function δ(q, w, s) as follows:

- δ(q0, ε, X) = {(q1, RX)}
- δ(q1, ε, S) = {(q1, XRX), (q2, S)}
- δ(q1, a, X) = {(q2, ε)}
- δ(q1, b, X) = {(q2, ε)}
- δ(q2, ε, R) = {(q3, ε)}

The intuition behind the above transition function is as follows:

- In the initial state q0, the PDA pushes X onto the stack.
- In state q1, the PDA non-deterministically chooses whether to expand X into a or b, or to expand S into XRX.
- If X is expanded into a or b, the PDA moves to state q2 and pops X from the stack.
- If S is expanded into XRX, the PDA pushes R and X onto the stack, moves to state q1, and repeats the process.
- When there are no more symbols to read and the stack is empty, the PDA accepts the input.

3. Check that the PDA P accepts the same language as the CFG G.

Therefore, we have successfully converted the CFG into an equivalent PDA.

ii)

To convert the context-free grammar (CFG) into an equivalent pushdown automaton (PDA), we can follow these steps:

Given a CFG G = (V, Σ, R, S), where:

S → aTa | bTb

T → bTa | abTb | X | ε

1. Create a new PDA P = (Q, Σ, Γ, δ, q0, Z, F), where:

   - Q = {q0, q1, q2, q3, q4, q5}

   - Σ = {a, b}

   - Γ = {a, b, X, Z}

   - F = {q5}

   - Z = Z (the stack symbol)

   - q0 is the start state.

2. Define the transition function δ(q, w, s) as follows:

   - δ(q0, ε, Z) = {(q1, Z)}

   - δ(q1, a, Z) = {(q2, aZ)}

   - δ(q1, b, Z) = {(q4, bZ)}

   - δ(q2, ε, Z) = {(q3, Z)}

   - δ(q3, ε, T) = {(q5, ε)}

   - δ(q2, ε, X) = {(q3, X)}

   - δ(q3, ε, T) = {(q2, ε), (q4, ε)}

   - δ(q4, ε, Z) = {(q3, Z)}

   - δ(q4, ε, X) = {(q3, X)}

   - δ(q3, a, a) = {(q3, ε)}

   - δ(q3, b, b) = {(q3, ε)}

   The intuition behind the above transition function is as follows:

- In the initial state q0, the PDA pushes Z onto the stack.
- In state q1, the PDA non-deterministically chooses whether the input starts with an 'a' or a 'b', and pushes the corresponding symbol onto the stack.
- In state q2, the PDA reads 'a' and pushes T onto the stack.
- In state q3, the PDA non-deterministically chooses whether to expand T into bTa, abTb, X, or ε.
- If T is expanded into bTa or abTb, the PDA moves to state q2, pushes the corresponding symbols onto the stack, and repeats the process.
- If T is expanded into X or ε, the PDA moves to state q3 and pops T from the stack.
- In state q4, the PDA reads 'b' and pushes T onto the stack.
- When the PDA reaches the end of the input and the stack is empty, the PDA accepts the input.

3. Check that the PDA P accepts the same language as the CFG G.

Therefore, we have successfully converted the CFG into an equivalent PDA.

iii)

To convert the context-free grammar (CFG) into an equivalent pushdown automaton (PDA), we can follow these steps:

Given a CFG G = (V, Σ, R, S), where:

T → bTa | abTb | X | ε

X → a | b

1. Create a new PDA P = (Q, Σ, Γ, δ, q0, Z, F), where:

- Q = {q0, q1, q2, q3, q4, q5, q6}
- Σ = {a, b}
- Γ = {a, b, Z}
- F = {q6}
- Z = Z (the stack symbol)
- q0 is the start state.

2. Define the transition function δ(q, w, s) as follows:

- δ(q0, ε, Z) = {(q1, Z)}
- δ(q1, ε, Z) = {(q2, Z)}
- δ(q2, a, Z) = {(q3, aZ)}
- δ(q2, b, Z) = {(q4, bZ)}
- δ(q3, ε, T) = {(q5, ε)}
- δ(q4, ε, T) = {(q5, ε)}
- δ(q2, ε, X) = {(q5, X)}
- δ(q5, ε, T) = {(q2, ε), (q4, ε), (q3, ε)}
- δ(q3, b, b) = {(q5, ε)}
- δ(q4, a, a) = {(q5, ε)}

The intuition behind the above transition function is as follows:

- In the initial state q0, the PDA pushes Z onto the stack.
- In state q1, the PDA moves to state q2 and pops Z from the stack.
- In state q2, the PDA non-deterministically chooses whether the input starts with an 'a' or a 'b', and pushes the corresponding symbol onto the stack.

- If the PDA reads 'a', it moves to state q3 and pushes T onto the stack.

- If the PDA reads 'b', it moves to state q4 and pushes T onto the stack.

- If the PDA reads X, it moves to state q5 and pushes X onto the stack.

- In state q5, the PDA non-deterministically chooses whether to expand T into bTa, abTb, or ε. If T is expanded into bTa, the PDA moves to state q2 and pushes the corresponding symbols onto the stack. If T is expanded into abTb, the PDA moves to state q2 and pushes the corresponding symbols onto the stack. If T is expanded into ε, the PDA pops T from the stack.

- If the PDA reads 'a' in state q3 or 'b' in state q4, it pops T from the stack and moves back to state q5.

- When the PDA reaches the end of the input and the stack is empty, the PDA accepts the input.

3. Check that the PDA P accepts the same language as the CFG G.

Therefore, we have successfully converted the CFG into an equivalent PDA.

iv)

To convert the context-free grammar (CFG) into an equivalent pushdown automaton (PDA), we can follow these steps:

Given a CFG G = (V, Σ, R, S), where:

$X \rightarrow a \mid b$

1. Create a new PDA P = (Q, Σ, Γ, δ, q0, Z, F), where:

   - Q = {q0, q1, q2}

- Σ = {a, b}
- Γ = {a, b, Z}
- F = {q2}
- Z = Z (the stack symbol)
- q0 is the start state.

2. Define the transition function δ(q, w, s) as follows:

- δ(q0, ε, Z) = {(q1, Z)}
- δ(q1, a, Z) = {(q2, aZ)}
- δ(q1, b, Z) = {(q2, bZ)}
- δ(q2, ε, X) = {(q2, X)}
- δ(q2, a, a) = {(q2, ε)}
- δ(q2, b, b) = {(q2, ε)}

The intuition behind the above transition function is as follows:

- In the initial state q0, the PDA pushes Z onto the stack.
- In state q1, the PDA non-deterministically chooses whether the input starts with an 'a' or a 'b', and pushes the corresponding symbol onto the stack.
- In state q2, the PDA non-deterministically chooses whether to expand X into 'a' or 'b'. If X is expanded into 'a', the PDA pops 'a' from the stack. If X is expanded into 'b', the PDA pops 'b' from the stack.
- When the PDA reaches the end of the input and the stack is empty, the PDA accepts the input.

3. Check that the PDA P accepts the same language as the CFG G.

Therefore, we have successfully converted the CFG into an equivalent PDA.

Answer to the question no. 4

(a)

(i)

L = { w | w contains three times as many 1s as 0s }.

Let M be a Turing machine that decides L. The machine M works as follows:

1. Start in state q0 with the tape head pointing to the leftmost symbol on the tape.
2. Scan the tape from left to right, counting the number of 0s and 1s on the tape.
3. If the tape contains an odd number of 0s, reject the input by moving to state q-reject and halting.
4. If the tape contains 3 times as many 1s as 0s, move to state q1.
5. If the tape contains a 1 followed by a 0, move to state q2.
6. If the tape contains a 0 followed by a 1, move to state q3.
7. If the tape contains a 1 followed by another 1, move to state q4.
8. If the tape contains a 0 followed by another 0, move to state q5.
9. If the tape contains a blank symbol, move to state q-accept and halt.
10. If the machine is in state q1, q2, q3, q4, or q5, move the tape head one symbol to the right and return to step 4.

The states q2, q3, q4, and q5 are used to ensure that the machine has counted the correct number of 1s and 0s. If the tape contains an incorrect symbol sequence, the machine will reject the input.

(ii)

B = { 0^m 1^n 2^(m+n) | m, n ≥ 0 and m > n }.

Let M be a Turing machine that decides B. The machine M works as follows:

1. Start in state q0 with the tape head pointing to the leftmost symbol on the tape.
2. Scan the tape from left to right, counting the number of 0s until a 1 is reached. If there are no 1s, move to state q-accept and halt.
3. Move the tape head to the rightmost symbol of the current block of 0s.
4. Scan the tape from this position to the right, counting the number of 1s until a 2 is reached. If there are no 2s, reject the input by moving to state q-reject and halting.
5. Move the tape head to the rightmost symbol of the current block of 1s.
6. Scan the tape from this position to the right, counting the number of 2s until the end of the tape is reached. If the number of 2s is not equal to the sum of the number of 0s and 1s, reject the input by moving to state q-reject and halting.
7. If the number of 0s is greater than the number of 1s, move to state q-accept and halt.
8. If the number of 0s is less than or equal to the number of 1s, repeat from step 2.

The key idea behind the implementation is to count the number of 0s, 1s, and 2s on the tape and compare them according to the language specification. If the number of 0s is greater than the number of 1s, then the machine accepts the input. If the number of 0s is less than or equal to the number of 1s, then the machine repeats the process until it either accepts or rejects the input.

(iii)

L = { w | w is a string with 0s and 1s and contains 1s in a multiple of 3 }.

Let M be a Turing machine that decides L. The machine M works as follows:

1. Start in state q0 with the tape head pointing to the leftmost symbol on the tape.
2. Scan the tape from left to right, counting the number of 1s until the end of the tape is reached.
3. If the number of 1s is not a multiple of 3, reject the input by moving to state q-reject and halting.
4. If the tape contains a 0 symbol, move the tape head one symbol to the right and return to step 2.
5. If the tape contains a 1 symbol, move to state q1.

6. If the tape contains a blank symbol, move to state q-accept and halt.
7. If the machine is in state q1, move the tape head one symbol to the right and check for the next symbol.
8. If the next symbol is a 0, move the tape head one symbol to the right and return to step 2.
9. If the next symbol is a 1, move to state q2.
10. If the machine is in state q2, move the tape head one symbol to the right and check for the next symbol.
11. If the next symbol is a 0, move the tape head one symbol to the right and return to step 2.
12. If the next symbol is a 1, move to state q3.
13. If the machine is in state q3, move the tape head one symbol to the right and return to step 2.

The idea behind the implementation is to first count the number of 1s in the input string, and reject the input if this number is not a multiple of 3. Then, the machine scans the input string for consecutive groups of 1s, and checks that each group contains exactly three 1s. If any group contains a different number of 1s, the machine rejects the input. If the machine reaches the end of the tape and has not rejected the input, it accepts the input.

Note that the above implementation assumes that the input is represented on the tape as a string of 0s and 1s, separated by blank symbols. If the input is represented in a different form, the implementation would need to be modified accordingly.

(iv)

B = {0^i 1^j 2^k | i+k=2*j and i,j,k>0}:

Let M be a Turing machine that decides B. The machine M works as follows:

1. Start in state q0 with the tape head pointing to the leftmost symbol on the tape.
2. Scan the tape from left to right, replacing each 0 symbol with X and counting the number of 0s until the end of the tape is reached. If the tape does not contain any 0s, reject the input by moving to state q-reject and halting.
3. If the tape contains a 1 symbol, move the tape head one symbol to the right and enter state q1.
4. If the tape contains a 2 symbol, move the tape head one symbol to the right and enter state q2.

5. If the tape contains a blank symbol, reject the input by moving to state q-reject and halting.
6. If the machine is in state q1, replace each 1 symbol with Y and count the number of 1s until the end of the tape is reached. If the tape does not contain any 1s, reject the input by moving to state q-reject and halting.
7. If the machine is in state q2, replace each 2 symbol with Z and count the number of 2s until the end of the tape is reached. If the tape does not contain any 2s, reject the input by moving to state q-reject and halting.
8. Move the tape head back to the leftmost symbol on the tape.
9. If the tape contains an X symbol, move the tape head one symbol to the right and enter state q3.
10. If the tape contains a Y symbol, move the tape head one symbol to the right and enter state q4.
11. If the tape contains a Z symbol, move the tape head one symbol to the right and enter state q5.
12. If the machine is in state q3, replace each X symbol with 0 and count the number of 0s until the end of the tape is reached.
13. If the machine is in state q4, replace each Y symbol with 1 and count the number of 1s until the end of the tape is reached.
14. If the machine is in state q5, replace each Z symbol with 2 and count the number of 2s until the end of the tape is reached.
15. If the number of 0s is greater than 0 and the number of 2s is equal to twice the number of 1s, accept the input by moving to state q-accept and halting.
16. If any of the above conditions are not satisfied, reject the input by moving to state q-reject and halting.

The idea behind the implementation is to first replace each 0 symbol with X, and each 1 symbol with Y, and each 2 symbol with Z. Then, the machine counts the number of Xs, Ys, and Zs on the tape. Finally, it checks if the number of Xs and Zs satisfy the condition i+k=2*j, where i, j, and k are the number of Xs, Ys, and Zs on the tape, respectively. If the condition is satisfied, the input is accepted; otherwise, the input is rejected.

(b)

Decidable Language: In the field of theoretical computer science, a decidable language is a language for which there exists a Turing machine that can always decide whether any given string belongs to the language or not. In other words, a decidable language is a language that can be recognized by a Turing machine that always halts and produces the correct answer (either accept or reject) for any input string.

More formally, a language L is decidable if there exists a Turing machine M such that for any input string w in the language L, M accepts w and halts in a finite number of steps; and for any input string w not in the language L, M rejects w and halts in a finite number of steps. A decidable language is also known as a computable language, recursive language, or effectively enumerable language.

The problem of determining whether a given number 'm' is prime is decidable or not:

Prime numbers = {2, 3, 5, 7, 11, 13, …………..}

Divide the number **'m'** by all the numbers between '2' and '$\sqrt{m}$' starting from '2'.

If any of these numbers produce a remainder zero, then it goes to the "Rejected state", otherwise it goes to the "Accepted state". So, here the answer could be made by 'Yes' or 'No'.

Hence, it is a decidable problem.

Autumn 2021 – Question number 5

5. a) Differentiate between a Finite automation and a Turing machine.

Answer:

Finite Automata	Touring Machine
It recognizes the language called regular language	It will recognize not only regular language but also context-free language, context-sensitive language, and recursively enumerable languages.
In this, the input tape is of finite length from both the left and right sides.	In this, the input tape is of finite length from the left but is of infinite length from the right.
In this head is able to move in the right direction only. In two-way automata, the head is able to move in both directions.	In this, the head can move in both directions.
It is weak as compared to Turing Machine.	It is more powerful than Finite Automata.
Designing finite automata is easier.	Designing turing machine is difficult and as well as complex.

b) Define the classes P, NP, NP-complete. Why NP-complete class is significant regarding the question whether P=NP?

Answer:

P Class: The P in the P class stands for Polynomial Time. It is the collection of decision problems (problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.

NP Class: The NP in NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

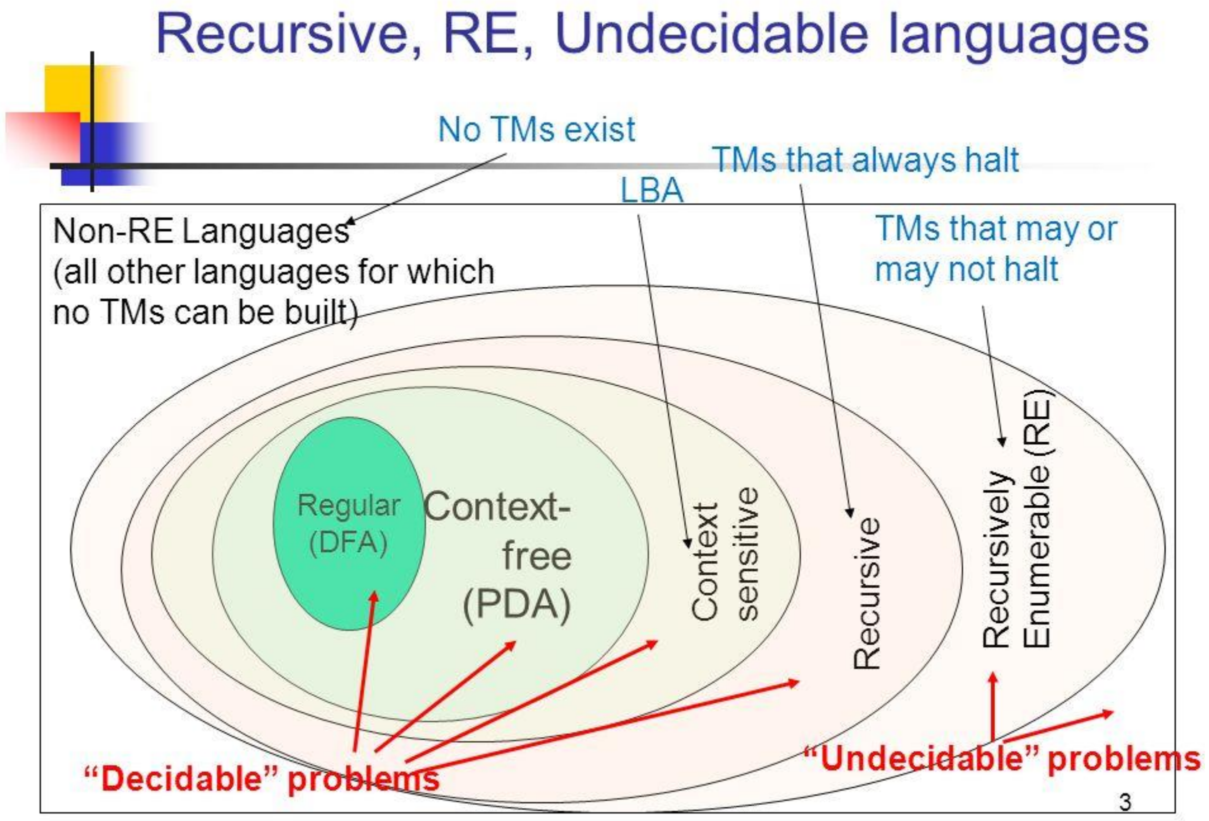
NP Complete Class: A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

NP-complete languages are significant because all NP-complete languages are thought of having similar hardness, in that process solving one implies that others are solved as well. If some NP-complete languages are proven to be in P, then all of NPs are proven to be in P.

Any NP language can be reduced to an NP-complete language. Use this reduction and the algorithm for the given NP-complete problem to solve any problem in NP. Hence all of them will have polynomial time solutions. The statement P=NP means that if a problem takes polynomial time on a non-deterministic TM, then one can build a deterministic TM which would solve the same problem also in polynomial time. So far nobody has been able to show that it can be done, but nobody has been able to prove that it cannot be done, either.

c) Show the relationship among the following types of language in a diagram: Regular Language, context free language, decidable language.

Answer:



d) Can you run a nondeterministic algorithm on a deterministic machine instead of a nondeterministic one? If your answer is yes, then explain how you can do it and how the running time will be affected. If your ans is no, then explain why it will not be possible. (Not Completed)

Answer: Yes, nondeterministic algorithm can successfully run on deterministic machine.

A non-deterministic algorithm usually has two phases and output steps. The first phase is the guessing phase, which makes use of arbitrary characters to run the problem. The second phase is the verifying phase, which returns true or false for the chosen string.

A non-deterministic algorithm can provide different outputs for the same input on different executions. Unlike a deterministic algorithm which produces only a single output for the same input even on different runs.