**CS 3452 - THEORY OF COMPUTATION**

**UNIT I AUTOMATA AND REGULAR EXPRESSIONS**

**Q1. i. Construct an NFA for the set of strings with {0,1} ending with 01. Draw the transition table for the same and check whether the input string 00101 is accepted by the above NFA.**

**1. Constructing NFA for strings ending with "01"**

We need to construct an NFA that accepts strings over {0, 1} that end with the substring "01."

**Steps to construct NFA:**

* Start state: **q0**
* Accepting state: **q2** (after the string has ended with "01")
* Transitions:
  + From **q0** (start state):
    - If the symbol is **0**, stay in **q0** (since we are still looking for the "01" substring).
    - If the symbol is **1**, move to **q1** (since we are looking for the "1" after a "0").
  + From **q1**:
    - If the symbol is **0**, move to **q2** (we've now seen the substring "01").
    - If the symbol is **1**, stay in **q1** (still looking for the second "1").
  + From **q2**:
    - We stay in **q2** regardless of the input (since the string has already ended with "01").

**NFA Diagram:**

lua

CopyEdit

q0 --0--> q0

q0 --1--> q1

q1 --0--> q2

q1 --1--> q1

q2 --0--> q2

q2 --1--> q2

**Transition Table:**

| **State** | **0** | **1** |
| --- | --- | --- |
| q0 | q0 | q1 |
| q1 | q2 | q1 |
| q2 | q2 | q2 |

**Checking if the input string 00101 is accepted:**

* Start at **q0**:
  + Read **0** → Stay at **q0**.
  + Read **0** → Stay at **q0**.
  + Read **1** → Move to **q1**.
  + Read **0** → Move to **q2**.
  + Read **1** → Stay at **q2**.

The machine ends at **q2** (an accepting state), so the string **00101** is accepted.

**Q1. ii. Construct an NFA for the set of all strings {0,1} that ends with three consecutive 1’s.**

**Steps to construct NFA for strings ending with three consecutive "1"s:**

* Start state: **q0**
* Accepting state: **q3** (after the string ends with "111")
* Transitions:
  + From **q0**:
    - If **0** → Stay in **q0** (still looking for the "111").
    - If **1** → Move to **q1** (first "1").
  + From **q1**:
    - If **0** → Stay in **q0** (reset if "0" appears).
    - If **1** → Move to **q2** (second "1").
  + From **q2**:
    - If **0** → Stay in **q0** (reset if "0" appears).
    - If **1** → Move to **q3** (third "1").
  + From **q3**:
    - Stay in **q3** (once we’ve seen "111", we remain in the accepting state).

**NFA Diagram:**

lua

CopyEdit

q0 --0--> q0

q0 --1--> q1

q1 --0--> q0

q1 --1--> q2

q2 --0--> q0

q2 --1--> q3

q3 --0--> q3

q3 --1--> q3

**Transition Table:**

| **State** | **0** | **1** |
| --- | --- | --- |
| q0 | q0 | q1 |
| q1 | q0 | q2 |
| q2 | q0 | q3 |
| q3 | q3 | q3 |

**Q1. iii. Construct an NFA for the set of all strings {a,b} with "abb" as a substring.**

**Steps to construct NFA for strings containing the substring "abb":**

* Start state: **q0**
* Accepting state: **q3** (once we encounter the substring "abb")
* Transitions:
  + From **q0**:
    - If **a** → Move to **q1** (beginning of the substring "abb").
    - If **b** → Stay at **q0** (no part of the substring "abb" has been seen yet).
  + From **q1**:
    - If **b** → Move to **q2** (second character of "abb").
    - If **a** → Stay at **q1** (starting the substring again).
  + From **q2**:
    - If **b** → Move to **q3** (we’ve seen the complete "abb").
    - If **a** → Stay at **q1** (start searching for "abb" again).
  + From **q3**:
    - Stay in **q3** (since we've seen the substring "abb", we remain in the accepting state regardless of the input).

**NFA Diagram:**

css

CopyEdit

q0 --a--> q1

q0 --b--> q0

q1 --b--> q2

q1 --a--> q1

q2 --b--> q3

q2 --a--> q1

q3 --a--> q3

q3 --b--> q3

**Transition Table:**

| **State** | **a** | **b** |
| --- | --- | --- |
| q0 | q1 | q0 |
| q1 | q1 | q2 |
| q2 | q1 | q3 |
| q3 | q3 | q3 |

**Conclusion:**

1. **For i**: The NFA for strings ending with "01" accepts the string **00101**.
2. **For ii**: The NFA accepts strings that end with three consecutive "1"s.
3. **For iii**: The NFA accepts strings containing "abb" as a substring.

**Q2. i. Prove that every tree has 'e' edges and 'e + 1' nodes.**

*(Nov/Dec 2014)*

**Theorem:**

For any tree with **e** edges, the number of nodes **n** is given by:

n=e+1n = e + 1n=e+1

**Proof (by Induction on the Number of Nodes):**

**Base Case:**

Consider a tree with **n = 1** (i.e., a tree with a single node).

* A single node has no edges, so **e = 0**.
* The statement holds true for this base case:  
  n=1=e+1n = 1 = e + 1n=1=e+1, where e=0e = 0e=0.

**Inductive Step:**

Assume that the statement holds for all trees with **k** nodes.  
That is, if a tree has **k** nodes, then it has **k - 1** edges. We now prove that the statement holds for **k + 1** nodes.

* Consider a tree with **k + 1** nodes. A tree is a connected acyclic graph, and it has exactly one path between any two nodes. To form a tree with **k + 1** nodes, we can add one more node to a tree with **k** nodes by connecting it with one edge to any of the existing nodes.
* Since adding one node requires exactly one edge, the number of edges in the tree with **k + 1** nodes will be the number of edges in the tree with **k** nodes plus 1.  
  By the inductive hypothesis, a tree with **k** nodes has **k - 1** edges. Therefore, a tree with **k + 1** nodes has:

(k−1)+1=kedges.(k - 1) + 1 = k \quad \text{edges.}(k−1)+1=kedges.

Thus, for **k + 1** nodes, the tree has **k** edges, which proves the statement for **k + 1** nodes.

**Conclusion:**

By the principle of mathematical induction, we conclude that for every tree, the number of nodes **n** is **e + 1**, where **e** is the number of edges.

**Therefore, for any tree, the number of edges is eee and the number of nodes is e+1e + 1e+1.**

**Q2. ii. Prove that for every integer n≥0n \geq 0n≥0, the number 42n+1+3n+242n + 1 + 3n + 242n+1+3n+2 is a multiple of 13.**

*(A)*

**Problem Statement:**

We are asked to prove that:

42n+1+3n+2is divisible by 13 for every integer n≥0.42n + 1 + 3n + 2 \quad \text{is divisible by 13 for every integer } n \geq 0.42n+1+3n+2is divisible by 13 for every integer n≥0.

Simplify the expression first:

42n+1+3n+2=(42n+3n)+(1+2)=45n+3.42n + 1 + 3n + 2 = (42n + 3n) + (1 + 2) = 45n + 3.42n+1+3n+2=(42n+3n)+(1+2)=45n+3.

Thus, we need to prove that:

45n+3is divisible by 13 for all integers n≥0.45n + 3 \quad \text{is divisible by 13 for all integers } n \geq 0.45n+3is divisible by 13 for all integers n≥0.

**Step 1: Simplifying 45n+345n + 345n+3 modulo 13.**

We first observe that:

45≡6mod  13.45 \equiv 6 \mod 13.45≡6mod13.

This is because 45÷1345 \div 1345÷13 gives a remainder of 6.

Thus, the expression becomes:

45n+3≡6n+3mod  13.45n + 3 \equiv 6n + 3 \mod 13.45n+3≡6n+3mod13.

**Step 2: Analyze the expression for different values of nnn.**

We want to prove that for all n≥0n \geq 0n≥0, 6n+3≡0mod  136n + 3 \equiv 0 \mod 136n+3≡0mod13.

Let’s evaluate this expression modulo 13 for values of nnn:

* **For n=0n = 0n=0:**

6(0)+3=3(not divisible by 13).6(0) + 3 = 3 \quad \text{(not divisible by 13)}.6(0)+3=3(not divisible by 13).

Hence, the base case doesn't hold for n=0n = 0n=0.

**Conclusion:**

The statement is **not true** for n=0n = 0n=0 since 333 is not divisible by 13.

However, there may have been a misunderstanding of your question in the context or additional constraints were needed to be defined.

**UNIT II REGULAR EXPRESSIONS AND LANGUAGES**

**1. Show that regular languages are closed under: Union, Intersection, Kleene Closure, Complement, and Difference.**

*(Nov-Dec 2017)*

**1. Closure under Union:**

A regular language is closed under union, which means that if **L₁** and **L₂** are regular languages, then **L₁ ∪ L₂** is also a regular language.

**Proof:**

* Regular languages can be represented by finite automata.
* If **L₁** is recognized by a finite automaton **M₁** and **L₂** is recognized by a finite automaton **M₂**, then we can construct a new finite automaton **M** that recognizes **L₁ ∪ L₂**.

**Construction:**

* + Create a new start state **q₀**.
  + Add **ε-transitions** (epsilon transitions) from **q₀** to the start states of **M₁** and **M₂**.
  + The accepting states of **M** will be the union of the accepting states of **M₁** and **M₂**.

Thus, by combining the two automata with epsilon transitions, we can construct a finite automaton for **L₁ ∪ L₂**. Therefore, the union of two regular languages is regular.

**2. Closure under Intersection:**

A regular language is closed under intersection, meaning that if **L₁** and **L₂** are regular languages, then **L₁ ∩ L₂** is also regular.

**Proof:**

* Let **M₁** and **M₂** be finite automata for **L₁** and **L₂**, respectively.
* We can construct a finite automaton that recognizes the intersection of the two languages by using the **product construction** method.

**Construction:**

* + Construct a new automaton **M** with states that are pairs of the states of **M₁** and **M₂** (i.e., **Q = Q₁ × Q₂**, where **Q₁** is the set of states of **M₁** and **Q₂** is the set of states of **M₂**).
  + The start state of **M** will be the pair of start states of **M₁** and **M₂**.
  + The accepting states of **M** will be the pairs where both components are accepting states of **M₁** and **M₂**, respectively.
  + The transition function of **M** will simulate the transitions of **M₁** and **M₂** simultaneously.

Thus, the intersection of two regular languages is also regular because it can be recognized by a finite automaton.

**3. Closure under Kleene Closure:**

A regular language is closed under Kleene closure, meaning that if **L** is a regular language, then **L⁺** (the Kleene closure of **L**) is also regular.

**Proof:**

* Let **M** be a finite automaton that recognizes the language **L**.
* To construct a finite automaton that recognizes **L⁺**, we can modify **M** to allow it to repeat its acceptance of strings.

**Construction:**

* + Add a new start state **q₀** to **M** with epsilon transitions to the original start state of **M** and to an accepting state of **M**.
  + The new start state can accept the empty string, ensuring that strings of length 0 (from the Kleene closure) are accepted.
  + Modify the transition function to allow **M** to go back to the start state upon reaching an accepting state, thereby enabling the automaton to accept multiple repetitions of strings from **L**.

Thus, the Kleene closure of a regular language is regular.

**4. Closure under Complement:**

A regular language is closed under complement, meaning that if **L** is a regular language, then **L'** (the complement of **L**) is also regular.

**Proof:**

* If **L** is a regular language, it is recognized by a deterministic finite automaton (DFA) **M**.
* The complement of **L** consists of all strings that are not in **L**.

**Construction:**

* + Take the DFA **M** for **L** and swap the accepting and non-accepting states.
  + This new automaton will accept all strings that **M** rejects and reject all strings that **M** accepts.

Thus, the complement of a regular language is regular because we can always construct a DFA that recognizes the complement of **L**.

**5. Closure under Difference:**

A regular language is closed under difference, meaning that if **L₁** and **L₂** are regular languages, then **L₁ - L₂** (the set difference of **L₁** and **L₂**) is also regular.

**Proof:**

* The difference of two languages **L₁ - L₂** can be expressed as the intersection of **L₁** and the complement of **L₂**:

L1−L2=L1∩L2′L₁ - L₂ = L₁ \cap L₂'L1​−L2​=L1​∩L2′​

* Since regular languages are closed under intersection and complement, and the complement of a regular language is regular, we can construct a finite automaton for **L₁ - L₂** by constructing the intersection of **L₁** and the complement of **L₂**.

**Construction:**

* + Given **L₁** as a DFA, and **L₂** as a DFA, construct the complement of **L₂** (using the procedure in the closure under complement proof).
  + Then, apply the product construction for the intersection of **L₁** and **L₂'**.

Thus, the difference of two regular languages is regular.

**Conclusion:**

Regular languages are closed under the following operations:

* **Union**
* **Intersection**
* **Kleene Closure**
* **Complement**
* **Difference**

In each case, we can construct a finite automaton or a regular expression that recognizes the resulting language, proving that the regular languages are closed under these operations.

**2. Prove that the following languages are not regular:**

*(Nov-Dec 2019)*

**i. Language: L1={w∈{a,b}∗∣w=wR}L\_1 = \{ w \in \{a, b\}^\* \mid w = w^R \}L1​={w∈{a,b}∗∣w=wR}**

This language consists of all strings over the alphabet {a,b}\{a, b\}{a,b} that are palindromes (i.e., the string is the same as its reverse).

**Proof (Using the Pumping Lemma):**

To prove that this language is **not regular**, we will use the **pumping lemma** for regular languages. The pumping lemma states that if a language LLL is regular, then there exists a constant ppp (the pumping length) such that any string s∈Ls \in Ls∈L with length ∣s∣≥p|s| \geq p∣s∣≥p can be split into three parts, s=xyzs = xyzs=xyz, satisfying the following conditions:

1. ∣xy∣≤p|xy| \leq p∣xy∣≤p,
2. ∣y∣>0|y| > 0∣y∣>0,
3. xynz∈Lxy^n z \in Lxynz∈L for all n≥0n \geq 0n≥0.

We assume that L1L\_1L1​ is regular and attempt to derive a contradiction.

* Let ppp be the pumping length for L1L\_1L1​. Consider the string s=apbps = a^p b^ps=apbp, which is a palindrome. Clearly, s∈L1s \in L\_1s∈L1​ because it is of the form w=wRw = w^Rw=wR.
* We split sss into three parts s=xyzs = xyzs=xyz, where ∣xy∣≤p|xy| \leq p∣xy∣≤p and ∣y∣>0|y| > 0∣y∣>0. Since ∣xy∣≤p|xy| \leq p∣xy∣≤p, the string xyxyxy consists only of aaa's (i.e., x=akx = a^kx=ak and y=amy = a^my=am for some kkk, mmm such that k+m≤pk + m \leq pk+m≤p).
* Now, pump yyy. Consider s′=xy2z=aka2map−mbps' = xy^2z = a^k a^{2m} a^{p-m} b^ps′=xy2z=aka2map−mbp, which is no longer a palindrome, as the number of aaa's does not match the number of bbb's at the end.

Since the pumped string is not in L1L\_1L1​, we have a contradiction, and the assumption that L1L\_1L1​ is regular is false.

Thus, L1={w∈{a,b}∗∣w=wR}L\_1 = \{ w \in \{a, b\}^\* \mid w = w^R \}L1​={w∈{a,b}∗∣w=wR} is **not regular**.

**ii. Language: L2={w∈{0,1}∗∣w begins with a 1 and the binary value of w is a prime number}L\_2 = \{ w \in \{0, 1\}^\* \mid w \text{ begins with a 1 and the binary value of } w \text{ is a prime number} \}L2​={w∈{0,1}∗∣w begins with a 1 and the binary value of w is a prime number}**

**Proof (Using the Pumping Lemma):**

The language L2L\_2L2​ consists of strings over the alphabet {0,1}\{0, 1\}{0,1} that represent binary numbers, which begin with a 1, and whose decimal value is a prime number.

We will again use the **pumping lemma** to prove that L2L\_2L2​ is not regular.

* Assume that L2L\_2L2​ is regular. Then, there exists a pumping length ppp such that any string w∈L2w \in L\_2w∈L2​ with ∣w∣≥p|w| \geq p∣w∣≥p can be split into three parts w=xyzw = xyzw=xyz satisfying the pumping lemma conditions.
* Consider the string w=1p0pw = 1^{p}0^{p}w=1p0p (this string represents the binary number 2p+12^p + 12p+1, which is prime for some ppp). Since this string starts with 1 and represents a prime number, it is in L2L\_2L2​.
* Split the string into w=xyzw = xyzw=xyz, where ∣xy∣≤p|xy| \leq p∣xy∣≤p and ∣y∣>0|y| > 0∣y∣>0. Since ∣xy∣≤p|xy| \leq p∣xy∣≤p, the string xyxyxy consists only of 111's (i.e., x=1kx = 1^kx=1k and y=1my = 1^my=1m for some kkk, mmm such that k+m≤pk + m \leq pk+m≤p).
* Now, pump yyy. Consider w′=xy2z=1k+2m0pw' = xy^2z = 1^{k + 2m} 0^pw′=xy2z=1k+2m0p. The new string represents a binary number that is no longer prime because the additional 111's create a number that is divisible by something other than 1 and itself.

Since the pumped string is not prime and does not belong to L2L\_2L2​, we have a contradiction.

Thus, L2={w∈{0,1}∗∣w begins with a 1 and the binary value of w is a prime number}L\_2 = \{ w \in \{0, 1\}^\* \mid w \text{ begins with a 1 and the binary value of } w \text{ is a prime number} \}L2​={w∈{0,1}∗∣w begins with a 1 and the binary value of w is a prime number} is **not regular**.

**Conclusion:**

Both languages L1={w∈{a,b}∗∣w=wR}L\_1 = \{ w \in \{a, b\}^\* \mid w = w^R \}L1​={w∈{a,b}∗∣w=wR} and L2={w∈{0,1}∗∣w begins with a 1 and the binary value of w is a prime number}L\_2 = \{ w \in \{0, 1\}^\* \mid w \text{ begins with a 1 and the binary value of } w \text{ is a prime number} \}L2​={w∈{0,1}∗∣w begins with a 1 and the binary value of w is a prime number} are **not regular**.

**UNIT III CONTEXT FREE GRAMMAR AND PUSH DOWN AUTOMATA**

**Deterministic Pushdown Automaton (DPDA)**

A **Deterministic Pushdown Automaton (DPDA)** is a type of **Pushdown Automaton (PDA)** where, for each state and input symbol, there is at most one transition, meaning the machine behaves deterministically. This means that for any given state and input symbol, there is only one possible action for the machine to take, making it different from a general PDA where multiple transitions can exist for the same state and input symbol.

**Formal Definition of DPDA:**

A **Deterministic Pushdown Automaton (DPDA)** can be formally defined as a 7-tuple:

M=(Q,Σ,Γ,δ,q0,Z0,F)M = (Q, \Sigma, \Gamma, \delta, q\_0, Z\_0, F)M=(Q,Σ,Γ,δ,q0​,Z0​,F)

Where:

* QQQ is the finite set of states.
* Σ\SigmaΣ is the input alphabet.
* Γ\GammaΓ is the stack alphabet.
* δ\deltaδ is the transition function δ:Q×(Σ∪{ε})×Γ→Q×Γ∗\delta: Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \to Q \times \Gamma^\*δ:Q×(Σ∪{ε})×Γ→Q×Γ∗, which determines the next state and the stack operations.
* q0q\_0q0​ is the initial state.
* Z0Z\_0Z0​ is the initial stack symbol.
* FFF is the set of accepting states.

A key difference from a non-deterministic PDA (NPDA) is that for every configuration of state, input symbol, and top stack symbol, the DPDA can have at most one transition.

**Example of a DPDA:**

Let’s look at a simple example to illustrate how a **Deterministic Pushdown Automaton (DPDA)** works:

**Problem:** Consider the language L={anbn∣n≥0}L = \{ a^n b^n \mid n \geq 0 \}L={anbn∣n≥0}, which consists of strings with an equal number of aaa's followed by bbb's.

This is a deterministic language because we can uniquely determine whether the string belongs to the language by reading from left to right, pushing aaa's onto the stack, and then popping the stack as we read bbb's.

**Steps:**

1. **Initial State:** Start in the initial state q0q\_0q0​, and begin reading the input string.
2. **Push aaa's onto the stack:** While reading aaa's, push them onto the stack. This will be the operation for all aaa's.
3. **Switch to bbb's:** When the first bbb is encountered, start popping the stack for each bbb read, matching each bbb with an aaa on the stack.
4. **Accept the string:** If all aaa's are matched with bbb's and the input string is completely consumed, the machine accepts the string.

**Formally:**

* Let the stack alphabet Γ={Z0,A}\Gamma = \{ Z\_0, A \}Γ={Z0​,A} where Z0Z\_0Z0​ is the initial stack symbol and AAA is used to represent an aaa.
* The machine starts in state q0q\_0q0​ with the stack initialized to Z0Z\_0Z0​.
* The transition function δ\deltaδ would work as follows:
  + On reading aaa from the input, push AAA onto the stack.
  + On reading bbb from the input, pop an AAA from the stack.
  + The input is accepted if all symbols are matched and the stack is empty when the input is exhausted.

**Transition Function:**

δ(q0,a,Z0)=(q0,AZ0)\delta(q\_0, a, Z\_0) = (q\_0, A Z\_0)δ(q0​,a,Z0​)=(q0​,AZ0​) δ(q0,a,A)=(q0,AA)\delta(q\_0, a, A) = (q\_0, A A)δ(q0​,a,A)=(q0​,AA) δ(q0,b,A)=(q0,ε)\delta(q\_0, b, A) = (q\_0, \varepsilon)δ(q0​,b,A)=(q0​,ε) δ(q0,ε,Z0)=(qf,Z0)\delta(q\_0, \varepsilon, Z\_0) = (q\_f, Z\_0)δ(q0​,ε,Z0​)=(qf​,Z0​)

Where:

* δ(q0,a,Z0)=(q0,AZ0)\delta(q\_0, a, Z\_0) = (q\_0, A Z\_0)δ(q0​,a,Z0​)=(q0​,AZ0​): When in state q0q\_0q0​ and reading an aaa, push AAA onto the stack.
* δ(q0,a,A)=(q0,AA)\delta(q\_0, a, A) = (q\_0, A A)δ(q0​,a,A)=(q0​,AA): When in state q0q\_0q0​ and reading another aaa, push another AAA onto the stack.
* δ(q0,b,A)=(q0,ε)\delta(q\_0, b, A) = (q\_0, \varepsilon)δ(q0​,b,A)=(q0​,ε): When in state q0q\_0q0​ and reading a bbb, pop AAA from the stack.
* δ(q0,ε,Z0)=(qf,Z0)\delta(q\_0, \varepsilon, Z\_0) = (q\_f, Z\_0)δ(q0​,ε,Z0​)=(qf​,Z0​): When all input has been consumed and the stack has only Z0Z\_0Z0​ left, transition to the final state qfq\_fqf​ and accept the string.

**Example:**

Consider the input string "aabbb".

* Start in state q0q\_0q0​ with the stack Z0Z\_0Z0​.
* Read aaa: push AAA onto the stack, so the stack is AZ0A Z\_0AZ0​.
* Read aaa: push another AAA onto the stack, so the stack is AAZ0A A Z\_0AAZ0​.
* Read bbb: pop AAA from the stack, so the stack is AZ0A Z\_0AZ0​.
* Read bbb: pop AAA from the stack, so the stack is Z0Z\_0Z0​.
* Read bbb: no more AAA's to pop, but since the input is exhausted, and the stack only contains Z0Z\_0Z0​, the string is accepted.

**Conclusion:**

A **Deterministic Pushdown Automaton (DPDA)** is a pushdown automaton where the transition function is deterministic, meaning that for each combination of current state, input symbol, and top-of-stack symbol, there is at most one possible action to take. In this example, we used a DPDA to recognize the language L={anbn∣n≥0}L = \{ a^n b^n \mid n \geq 0 \}L={anbn∣n≥0}, which is deterministic and can be processed efficiently using a DPDA.

**2.Let G=(V,T,P,S) be a CFG then prove that if the recursive inference procedure tells us that terminal string W is in the language of variable A, then there is a parse tree with root A and yield w.**

**Problem Statement:**

Let G=(V,T,P,S)G = (V, T, P, S)G=(V,T,P,S) be a context-free grammar (CFG), where:

* VVV is a set of variables (non-terminal symbols),
* TTT is a set of terminals (terminal symbols),
* PPP is a set of production rules,
* SSS is the start variable.

We need to prove that if the recursive inference procedure tells us that a terminal string www is in the language of variable AAA, i.e., A⇒∗wA \Rightarrow^\* wA⇒∗w (where ⇒∗\Rightarrow^\*⇒∗ represents derivation in the grammar), then there exists a parse tree with root AAA and yield www.

**Proof:**

We will prove the statement by induction on the number of steps in the derivation.

**Base Case (Zero steps):**

* If www is a terminal string that can be derived from AAA in **zero steps**, this means that A⇒∗wA \Rightarrow^\* wA⇒∗w is achieved without any intermediate steps. In this case, the string www must be a terminal string that is directly generated by a production of the form:

A→w(where w is a terminal string).A \rightarrow w \quad \text{(where \( w \) is a terminal string)}.A→w(where w is a terminal string).

The parse tree for this derivation would consist of a single node labeled AAA with a direct edge to the terminal string www. Since A⇒∗wA \Rightarrow^\* wA⇒∗w in this case is just A→wA \rightarrow wA→w, there is trivially a parse tree with root AAA and yield www.

**Inductive Step (Assume for kkk steps, prove for k+1k+1k+1):**

* Assume that for any string www that can be derived from AAA in kkk steps, there exists a parse tree with root AAA and yield www. We need to prove that for any string www that can be derived from AAA in k+1k+1k+1 steps, there exists a parse tree with root AAA and yield www.
* Let’s consider the derivation of www in k+1k+1k+1 steps:

A⇒∗X1⇒∗X2⇒∗⋯⇒∗Xk⇒∗w,A \Rightarrow^\* X\_1 \Rightarrow^\* X\_2 \Rightarrow^\* \dots \Rightarrow^\* X\_{k} \Rightarrow^\* w,A⇒∗X1​⇒∗X2​⇒∗⋯⇒∗Xk​⇒∗w,

where XiX\_iXi​ are intermediate strings and the last step involves applying a production of the form:

A⇒A1 A2…An(where A1,A2,…,An are variables or terminals).A \Rightarrow A\_1 \, A\_2 \dots A\_n \quad \text{(where \( A\_1, A\_2, \dots, A\_n \) are variables or terminals)}.A⇒A1​A2​…An​(where A1​,A2​,…,An​ are variables or terminals).

So, the last step of the derivation involves applying a production of the form:

A→A1A2…An.A \rightarrow A\_1 A\_2 \dots A\_n.A→A1​A2​…An​.

* In this step, the string www is derived from AAA by first deriving the strings X1,X2,…,XnX\_1, X\_2, \dots, X\_nX1​,X2​,…,Xn​ from the variables A1,A2,…,AnA\_1, A\_2, \dots, A\_nA1​,A2​,…,An​, respectively, in the previous steps. Since we assumed by induction that for A1⇒∗X1A\_1 \Rightarrow^\* X\_1A1​⇒∗X1​, A2⇒∗X2A\_2 \Rightarrow^\* X\_2A2​⇒∗X2​, and so on, each XiX\_iXi​ has a corresponding parse tree, we can now construct a parse tree for AAA as follows:
  1. The root of the tree is labeled AAA.
  2. The children of this root are labeled A1,A2,…,AnA\_1, A\_2, \dots, A\_nA1​,A2​,…,An​.
  3. Each subtree rooted at AiA\_iAi​ corresponds to the parse tree for XiX\_iXi​ (which exists by the induction hypothesis).
* Thus, we can build a parse tree for AAA where the leaves are the terminal symbols, and the structure of the tree corresponds to the derivation of www from AAA.
* Therefore, if A⇒∗wA \Rightarrow^\* wA⇒∗w in k+1k+1k+1 steps, there exists a parse tree with root AAA and yield www.

**Conclusion:**

By induction, we have shown that if the recursive inference procedure tells us that the terminal string www is in the language of variable AAA, i.e., A⇒∗wA \Rightarrow^\* wA⇒∗w, then there exists a parse tree with root AAA and yield www.

This completes the proof.

**UNIT IV NORMAL FORMS AND TURING MACHINES**

**1. Convert the following grammar into GNF A Nov/Dec 2013 S◊XY1/0, X◊00X/Y,Y◊1X1**

**Problem:**

Convert the given grammar into **Greibach Normal Form (GNF)**.

**Given Grammar:**

1. S→XY1∣0S \rightarrow XY1 \mid 0S→XY1∣0
2. X→00X∣YX \rightarrow 00X \mid YX→00X∣Y
3. Y→1X1Y \rightarrow 1X1Y→1X1

**Steps to Convert into Greibach Normal Form (GNF):**

A **grammar is in Greibach Normal Form (GNF)** if the right-hand side of each production starts with a terminal symbol and is followed by zero or more variables.

To convert the given grammar into GNF, we must ensure that each production has the form:

A→aαA \rightarrow a \alphaA→aα

where:

* aaa is a terminal symbol, and
* α\alphaα is a string of variables (which can be empty).

Let's go through the given grammar step by step and convert it into GNF.

**Step 1: Convert S→XY1∣0S \rightarrow XY1 \mid 0S→XY1∣0**

* The production S→XY1S \rightarrow XY1S→XY1 has a non-terminal XXX followed by another non-terminal YYY, which does not start with a terminal. To satisfy GNF, we need to replace XXX and YYY with productions that start with terminal symbols.
* The production S→0S \rightarrow 0S→0 is already in GNF because it starts with the terminal symbol 000.

For S→XY1S \rightarrow XY1S→XY1, we will need to expand the productions for XXX and YYY.

**Step 2: Convert X→00X∣YX \rightarrow 00X \mid YX→00X∣Y**

* The production X→00XX \rightarrow 00XX→00X starts with two terminal symbols, 000000, but then continues with a non-terminal XXX, which violates GNF. We need to expand XXX into productions that start with terminal symbols.
* The production X→YX \rightarrow YX→Y needs to be expanded as well because YYY is a non-terminal.

Let's now substitute the productions for XXX and YYY.

**Step 3: Convert Y→1X1Y \rightarrow 1X1Y→1X1**

* The production Y→1X1Y \rightarrow 1X1Y→1X1 starts with the terminal 111, which is fine for GNF. However, it also includes a non-terminal XXX, so we need to expand XXX first.

Now let's substitute the appropriate expansions into the productions for XXX and YYY.

**Final Conversion:**

1. Start with SSS:
   * S→XY1S \rightarrow XY1S→XY1
   * S→0S \rightarrow 0S→0

Since XXX and YYY are non-terminals, expand them.

1. For XXX:
   * X→00XX \rightarrow 00XX→00X
   * X→YX \rightarrow YX→Y

Replace YYY using its definition.

1. For YYY:
   * Y→1X1Y \rightarrow 1X1Y→1X1

Let's replace all non-terminal symbols by expanding them and ensure the grammar is in GNF:

1. S→XY1S \rightarrow XY1S→XY1
   * Replace XXX and YYY with their definitions:
   * S→(00X1)Y1S \rightarrow (00X1)Y1S→(00X1)Y1
   * S→0S \rightarrow 0S→0
2. X→YX \rightarrow YX→Y and Y→1X1Y \rightarrow 1X1Y→1X1
   * Replace XXX in the above and continue

**2. Give the five tuple representation of a TM and explain the representation. Define the language accepted by a TM.**

Five-Tuple Representation of a Turing Machine (TM):

A Turing Machine (TM) is a theoretical computational model that consists of an infinite tape, a tape head that can move left or right, and a finite set of states. The machine can process symbols from its input based on a set of rules (transition function).

The Five-Tuple Representation of a Turing Machine is formally defined as:

M=(Q,Σ,Γ,δ,(q0,qaccept,qreject))M = (Q, \Sigma, \Gamma, \delta, (q\_0, q\_{\text{accept}}, q\_{\text{reject}}))M=(Q,Σ,Γ,δ,(q0​,qaccept​,qreject​))

Where:

* QQQ is a finite set of states (including the initial, accept, and reject states).
* Σ\SigmaΣ is a finite set of input symbols (the alphabet that the machine can read from the input tape).
* Γ\GammaΓ is a finite set of tape symbols (which includes the input alphabet Σ\SigmaΣ, and an additional blank symbol ⊔\sqcup⊔ to represent unused positions on the tape).
* δ\deltaδ is the transition function, which is a relation that defines the machine's behavior. It is a mapping:

δ:Q×Γ→Q×Γ×{L,R}\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}δ:Q×Γ→Q×Γ×{L,R}

This means that for each state q∈Qq \in Qq∈Q and each tape symbol γ∈Γ\gamma \in \Gammaγ∈Γ, the transition function specifies:

* + The next state q′∈Qq' \in Qq′∈Q,
  + The symbol to write on the tape γ′∈Γ\gamma' \in \Gammaγ′∈Γ,
  + The direction to move the tape head LLL (left) or RRR (right).
* q0q\_0q0​ is the start state, which is the state from which the Turing Machine begins its computation.
* qacceptq\_{\text{accept}}qaccept​ is the accept state, where the machine halts and accepts the input.
* qrejectq\_{\text{reject}}qreject​ is the reject state, where the machine halts and rejects the input.

Explanation of the Five-Tuple Representation:

1. Set of States (QQQ):
   * This includes all possible states the Turing machine can be in during its computation.
   * Example: Q={q0,q1,qaccept,qreject}Q = \{ q\_0, q\_1, q\_{\text{accept}}, q\_{\text{reject}} \}Q={q0​,q1​,qaccept​,qreject​}, where q0q\_0q0​ is the start state, qacceptq\_{\text{accept}}qaccept​ is the accept state, and qrejectq\_{\text{reject}}qreject​ is the reject state.
2. Input Alphabet (Σ\SigmaΣ):
   * This is the set of symbols that the Turing machine can read from the input tape.
   * Example: Σ={0,1}\Sigma = \{ 0, 1 \}Σ={0,1} for a binary input alphabet.
3. Tape Alphabet (Γ\GammaΓ):
   * This is the set of symbols that the machine can write to the tape, which includes all the symbols in Σ\SigmaΣ and at least one special blank symbol ⊔\sqcup⊔ (representing empty cells on the tape).
   * Example: Γ={0,1,⊔}\Gamma = \{ 0, 1, \sqcup \}Γ={0,1,⊔}.
4. Transition Function (δ\deltaδ):
   * This function defines the machine's behavior: for each combination of the current state and tape symbol, it provides the next state, the symbol to write on the tape, and the direction to move the tape head.
   * Example:
     + δ(q0,0)=(q1,1,R)\delta(q\_0, 0) = (q\_1, 1, R)δ(q0​,0)=(q1​,1,R), which means that if the machine is in state q0q\_0q0​ and reads a 0, it writes a 1, moves the head right, and transitions to state q1q\_1q1​.
5. Start State (q0q\_0q0​):
   * The state in which the machine begins its computation.
6. Accept and Reject States (qaccept,qrejectq\_{\text{accept}}, q\_{\text{reject}}qaccept​,qreject​):
   * The states that represent acceptance and rejection of the input, respectively.

Language Accepted by a Turing Machine (TM):

The language accepted by a Turing Machine is defined as the set of strings for which the machine halts in the accept state.

Formally, the language L(M)L(M)L(M) accepted by a Turing machine MMM is:

L(M)={w∈Σ∗∣M halts in qaccept when started on w}L(M) = \{ w \in \Sigma^\* \mid M \text{ halts in } q\_{\text{accept}} \text{ when started on } w \}L(M)={w∈Σ∗∣M halts in qaccept​ when started on w}

Where:

* Σ∗\Sigma^\*Σ∗ is the set of all possible strings over the input alphabet Σ\SigmaΣ.
* A string www is accepted by MMM if, after processing www from the input tape, the machine halts in the accept state qacceptq\_{\text{accept}}qaccept​.

If the machine halts in the reject state qrejectq\_{\text{reject}}qreject​, then the string is rejected. If the machine does not halt (i.e., it goes into an infinite loop), the string is considered not accepted.

Summary:

* A Turing Machine is represented by a five-tuple (Q,Σ,Γ,δ,(q0,qaccept,qreject))(Q, \Sigma, \Gamma, \delta, (q\_0, q\_{\text{accept}}, q\_{\text{reject}}))(Q,Σ,Γ,δ,(q0​,qaccept​,qreject​)).
* The language accepted by a TM is the set of strings that lead the machine to the accept state after processing the input, where the machine halts. If the machine halts in the reject state or never halts, the string is not accepted.

**UNIT V UNDECIDABILITY**

**1. Discuss the difference between NP-complete and NP-hard problems. May/June 2012**

**Difference Between NP-Complete and NP-Hard Problems**

The terms **NP-complete** and **NP-hard** refer to classes of computational problems, but they have distinct meanings. Let's discuss these terms in detail:

**1. NP-Complete Problems:**

A problem is **NP-complete** if it satisfies two conditions:

1. **It is in NP**: This means the problem can be verified in polynomial time. More formally, for a given solution, it is possible to check whether it is correct in polynomial time.
2. **It is NP-hard**: The problem is as hard as any other problem in NP, meaning any problem in NP can be reduced to this problem in polynomial time.

In other words, an NP-complete problem is one that:

* Belongs to the NP class, meaning that its solution can be verified in polynomial time.
* Is at least as hard as any other NP problem, meaning every problem in NP can be reduced to it in polynomial time.

**Examples of NP-Complete Problems:**

* **3-SAT**: A version of the Boolean satisfiability problem where each clause has exactly 3 literals.
* **Traveling Salesman Problem (Decision version)**: Given a set of cities and a distance matrix, is there a route that visits each city exactly once and has a total distance less than or equal to a given value?
* **Knapsack Problem (0/1 version)**: Given a set of items, each with a weight and value, is there a subset of the items that fit into a knapsack of limited weight capacity, such that the total value is at least a given value?

**2. NP-Hard Problems:**

A problem is **NP-hard** if it satisfies the following condition:

* **It is at least as hard as the hardest problem in NP**: In other words, if a problem is NP-hard, it means that every problem in NP can be reduced to it in polynomial time. However, NP-hard problems may or may not be in NP themselves.

The key point about NP-hard problems is that they do not necessarily belong to the NP class, meaning they might not even have a solution that can be verified in polynomial time. In fact, NP-hard problems can be harder than NP-complete problems, and some may be undecidable (i.e., there is no algorithm that can solve them in a finite amount of time for all inputs).

**Examples of NP-Hard Problems:**

* **Halting Problem**: Given a program and its input, decide whether the program will halt or run forever. This problem is undecidable and therefore NP-hard, but it is not in NP.
* **Travelling Salesman Problem (Optimization version)**: Given a set of cities, find the shortest route that visits each city exactly once. This is an NP-hard problem because it's at least as hard as the decision version of the Traveling Salesman Problem (which is NP-complete), but it's not in NP because finding the shortest route cannot be verified in polynomial time.

**Key Differences:**

| **Aspect** | **NP-Complete** | **NP-Hard** |
| --- | --- | --- |
| **Class Membership** | Belongs to NP (solvable in nondeterministic polynomial time) | May or may not belong to NP (can be harder than NP problems) |
| **Definition** | A problem in NP that is as hard as any other problem in NP | A problem as hard as the hardest problem in NP (not necessarily in NP itself) |
| **Verification of Solution** | Solutions can be verified in polynomial time | Solutions may not be verifiable in polynomial time |
| **Example** | 3-SAT, Knapsack Problem (0/1), Traveling Salesman Problem (Decision version) | Halting Problem, Traveling Salesman Problem (Optimization version) |
| **Polynomial-Time Reduction** | Any NP problem can be reduced to an NP-complete problem in polynomial time | Every NP problem can be reduced to an NP-hard problem in polynomial time, but it may not be solvable in polynomial time |

**Summary:**

* **NP-complete** problems are a subset of NP problems that are both in NP and as hard as any other problem in NP. If we find a polynomial-time solution to any NP-complete problem, we could solve all problems in NP in polynomial time.
* **NP-hard** problems are at least as hard as the hardest problems in NP, but they may not belong to NP. Some NP-hard problems might not even have a solution that can be verified in polynomial time.

In essence, all NP-complete problems are NP-hard, but not all NP-hard problems are NP-complete.

**2. Explain how to measure and classify complexity.**

**Measuring and Classifying Complexity in Computation**

Complexity in computational problems refers to how the resources required to solve a problem (such as time and space) grow as the size of the input increases. **Computational complexity** helps in analyzing the efficiency of algorithms, and it's typically measured in terms of **time complexity** and **space complexity**.

**1. Time Complexity:**

Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the size of its input. It is typically expressed as a function of nnn, where nnn is the size of the input.

* **Big-O Notation**: Time complexity is most commonly represented using Big-O notation, which describes the upper bound of an algorithm's running time. It gives an idea of the worst-case scenario.
  + **Example:**
    - O(1)O(1)O(1): Constant time – The algorithm’s runtime does not depend on the input size. Example: Accessing an array element.
    - O(n)O(n)O(n): Linear time – The algorithm’s runtime grows linearly with the input size. Example: A loop that processes each element of an array.
    - O(n2)O(n^2)O(n2): Quadratic time – The runtime grows quadratically with the input size. Example: Bubble Sort.
    - O(log⁡n)O(\log n)O(logn): Logarithmic time – The algorithm’s runtime grows logarithmically as the input size increases. Example: Binary search.
    - O(nlog⁡n)O(n \log n)O(nlogn): Linearithmic time – Common in efficient sorting algorithms, such as Merge Sort and Quick Sort.
* **Best, Worst, and Average Case**:
  + **Best Case**: The scenario where the algorithm performs the fewest steps.
  + **Worst Case**: The scenario where the algorithm performs the most steps.
  + **Average Case**: The expected performance of the algorithm, considering all possible inputs.

**2. Space Complexity:**

Space complexity measures the amount of memory space an algorithm uses as a function of the size of its input.

* Just like time complexity, space complexity is typically expressed in Big-O notation.
  + **Example:**
    - O(1)O(1)O(1): Constant space – The algorithm uses a fixed amount of space regardless of the input size.
    - O(n)O(n)O(n): Linear space – The algorithm uses space proportional to the size of the input. Example: Storing input data in an array.
    - O(n2)O(n^2)O(n2): Quadratic space – The algorithm uses space that grows quadratically with input size.

**3. Classifying Computational Problems:**

Computational problems are classified based on their **complexity**. These classifications help in understanding the difficulty of solving a problem and determining whether an efficient solution exists. The primary classifications include **P**, **NP**, **NP-complete**, **NP-hard**, and **EXPTIME**, among others.

**P (Polynomial Time):**

* **P** refers to the class of problems that can be solved in polynomial time, i.e., problems for which an algorithm exists that solves them in O(nk)O(n^k)O(nk) time, where kkk is a constant.
* **Example**: Sorting an array, finding the shortest path in a graph (Dijkstra’s algorithm).

**NP (Nondeterministic Polynomial Time):**

* **NP** refers to the class of problems for which a solution, if provided, can be verified in polynomial time. However, it is not known if every problem in NP can be solved in polynomial time.
* **Example**: The Boolean satisfiability problem (SAT), the traveling salesman problem (TSP) in decision form.

**NP-Complete:**

* **NP-complete** problems are the hardest problems in NP. If a polynomial-time solution exists for any NP-complete problem, then all problems in NP can also be solved in polynomial time.
* **Example**: 3-SAT, Knapsack problem, Traveling Salesman Problem.

**NP-Hard:**

* **NP-hard** problems are at least as hard as the hardest problems in NP, but they may not be in NP. These problems might not have a solution that can be verified in polynomial time or even be decidable.
* **Example**: The Halting problem, optimization versions of NP-complete problems.

**EXPTIME (Exponential Time):**

* Problems in **EXPTIME** are those that require exponential time to solve. These problems cannot be solved in polynomial time, and their solution may require time proportional to 2n2^n2n or worse.
* **Example**: Some decision problems in logic, certain planning problems.

**Classifying Based on Reductions:**

* Problems are classified by how they relate to each other using reductions. A **reduction** is a way of transforming one problem into another. For example, if a problem AAA can be reduced to problem BBB, and problem BBB is known to be NP-complete, then problem AAA is also NP-complete.

**4. Key Complexity Classes:**

* **P**: Problems solvable in polynomial time.
* **NP**: Problems for which solutions can be verified in polynomial time.
* **NP-complete**: Problems that are both in NP and as hard as any problem in NP.
* **NP-hard**: Problems that are at least as hard as the hardest NP problems, but they may not belong to NP.
* **EXPTIME**: Problems requiring exponential time to solve.

**5. Complexity Hierarchy and Open Problems:**

The **P vs NP problem** is one of the most famous unsolved questions in computer science. It asks whether every problem for which a solution can be verified in polynomial time (NP) can also be solved in polynomial time (P). If P=NPP = NPP=NP, then all NP problems could be solved efficiently (in polynomial time), but if P≠NPP \neq NPP=NP, then there are problems that are hard to solve but easy to verify.

**Conclusion:**

Measuring and classifying complexity helps in understanding the difficulty and feasibility of solving problems. **Time complexity** and **space complexity** give an estimate of how an algorithm behaves with respect to resource usage as input size grows. **Complexity classes** categorize problems based on how hard they are to solve or verify, and this classification helps guide the development of efficient algorithms.

Bottom of Form