**CS 3452 - THEORY OF COMPUTATION**

**UNIT I AUTOMATA AND REGULAR EXPRESSIONS**

**Q1. Prove by induction for all n≥0n \geq 0n≥0:**

**i. 12+22+32+⋯+n2=n(n+1)(2n+1)61^2 + 2^2 + 3^2 + \dots + n^2 = \dfrac{n(n+1)(2n+1)}{6}12+22+32+⋯+n2=6n(n+1)(2n+1)​**

**Step 1: Base Case (n = 1)**

LHS = 12=11^2 = 112=1

RHS = 1(1+1)(2⋅1+1)6=1⋅2⋅36=1\dfrac{1(1+1)(2 \cdot 1 + 1)}{6} = \dfrac{1 \cdot 2 \cdot 3}{6} = 161(1+1)(2⋅1+1)​=61⋅2⋅3​=1

Since LHS = RHS, the base case holds.

**Step 2: Inductive Hypothesis**

Assume that the formula holds true for some n=kn = kn=k, i.e.,

12+22+⋯+k2=k(k+1)(2k+1)61^2 + 2^2 + \dots + k^2 = \dfrac{k(k+1)(2k+1)}{6}12+22+⋯+k2=6k(k+1)(2k+1)​

**Step 3: Inductive Step**

We must prove it holds for n=k+1n = k+1n=k+1. That is,

12+22+⋯+k2+(k+1)2=(k+1)(k+2)(2k+3)61^2 + 2^2 + \dots + k^2 + (k+1)^2 = \dfrac{(k+1)(k+2)(2k+3)}{6}12+22+⋯+k2+(k+1)2=6(k+1)(k+2)(2k+3)​

Start with the LHS using the hypothesis:

(k(k+1)(2k+1)6)+(k+1)2\left( \dfrac{k(k+1)(2k+1)}{6} \right) + (k+1)^2(6k(k+1)(2k+1)​)+(k+1)2

Factor out (k+1)(k+1)(k+1):

=(k+1)6[k(2k+1)+6(k+1)]= \dfrac{(k+1)}{6} \left[ k(2k+1) + 6(k+1) \right]=6(k+1)​[k(2k+1)+6(k+1)]

Now simplify inside the brackets:

=(k+1)6[2k2+k+6k+6]=(k+1)6(2k2+7k+6)= \dfrac{(k+1)}{6} \left[ 2k^2 + k + 6k + 6 \right] = \dfrac{(k+1)}{6} (2k^2 + 7k + 6)=6(k+1)​[2k2+k+6k+6]=6(k+1)​(2k2+7k+6)

Factor the quadratic:

2k2+7k+6=(k+2)(2k+3)2k^2 + 7k + 6 = (k+2)(2k+3)2k2+7k+6=(k+2)(2k+3)

So,

=(k+1)(k+2)(2k+3)6= \dfrac{(k+1)(k+2)(2k+3)}{6}=6(k+1)(k+2)(2k+3)​

This is the RHS for n=k+1n = k+1n=k+1. Hence, proved.

✅ **Therefore, by mathematical induction, the formula holds for all n≥1n \geq 1n≥1.**

**ii. 13+23+33+⋯+n3=n2(n+1)241^3 + 2^3 + 3^3 + \dots + n^3 = \dfrac{n^2(n+1)^2}{4}13+23+33+⋯+n3=4n2(n+1)2​**

**Step 1: Base Case (n = 1)**

LHS = 13=11^3 = 113=1

RHS = 12⋅(1+1)24=1⋅44=1\dfrac{1^2 \cdot (1+1)^2}{4} = \dfrac{1 \cdot 4}{4} = 1412⋅(1+1)2​=41⋅4​=1

LHS = RHS → base case holds.

**Step 2: Inductive Hypothesis**

Assume the result holds for n=kn = kn=k:

13+23+⋯+k3=k2(k+1)241^3 + 2^3 + \dots + k^3 = \dfrac{k^2(k+1)^2}{4}13+23+⋯+k3=4k2(k+1)2​

**Step 3: Inductive Step**

Prove it for n=k+1n = k+1n=k+1:

13+23+⋯+k3+(k+1)3=(k+1)2(k+2)241^3 + 2^3 + \dots + k^3 + (k+1)^3 = \dfrac{(k+1)^2(k+2)^2}{4}13+23+⋯+k3+(k+1)3=4(k+1)2(k+2)2​

Start from the LHS:

(k2(k+1)24)+(k+1)3\left( \dfrac{k^2(k+1)^2}{4} \right) + (k+1)^3(4k2(k+1)2​)+(k+1)3

Factor out (k+1)2(k+1)^2(k+1)2:

=(k+1)24[k2+4(k+1)]= \dfrac{(k+1)^2}{4} \left[ k^2 + 4(k+1) \right]=4(k+1)2​[k2+4(k+1)]

Simplify:

k2+4k+4=(k+2)2k^2 + 4k + 4 = (k + 2)^2k2+4k+4=(k+2)2

So:

=(k+1)2(k+2)24= \dfrac{(k+1)^2(k+2)^2}{4}=4(k+1)2(k+2)2​

Which is RHS for n=k+1n = k+1n=k+1. Hence, proved.

✅ **Therefore, by mathematical induction, the formula holds for all n≥1n \geq 1n≥1.**

**✅ Final Answer:**

Both the identities have been proven using **mathematical induction**, which includes:

* Base Case Verification
* Inductive Hypothesis
* Inductive Step with simplification and algebraic factorization

## ****2. i. Construct DFA to accept the language****

**L = { w | w is of even length and begins with 11 }**

### ✳️ ****Language Description:****

* The string must **start with "11"**
* The **total length must be even**
* The alphabet is assumed to be **{0,1}**

### ✅ ****Approach to Constructing DFA:****

We design states based on:

1. Ensuring the string starts with "11"
2. Keeping track of **even/odd length** of the string.

Let’s define the states:

* **q0**: Initial state
* **q1**: Read first '1' (may be invalid unless second '1' follows)
* **q2**: Read "11" → beginning of a valid string of length 2 (even)
* **q3**: Odd length after "11" (odd after reading one extra symbol)
* **q4**: Even length after "11" (even length ≥ 4) ✅ Accepting state

### 🚦 ****State Transitions:****

| **Current State** | **Input** | **Next State** | **Explanation** |
| --- | --- | --- | --- |
| q0 | 1 | q1 | First '1' seen |
| q0 | 0 | Dead | Doesn't start with 11 |
| q1 | 1 | q2 | Now we have "11" |
| q1 | 0 | Dead | Invalid start |
| q2 | 0/1 | q3 | Now length = 3 (odd) |
| q3 | 0/1 | q4 | Length = 4 (even) |
| q4 | 0/1 | q3 | Toggle to odd |
| Dead | 0/1 | Dead | Trap state |

### ✅ ****Accepting States:****

* **q2** (length = 2)
* **q4** (even length ≥ 4)

But since we only want even length, and string must start with "11", **q2 is valid only if we allow strings of length 2**, which we do. So:

* Accepting states: **q2 and q4**

### ✏️ ****Sketch the DFA:****

(Draw this on paper in an exam. Nodes: q0, q1, q2, q3, q4, Dead)

text

CopyEdit

q0 --1--> q1 --1--> q2 (Accepting)

q2 --0/1--> q3

q3 --0/1--> q4 (Accepting)

q4 --0/1--> q3

Invalid paths go to Dead state

## ****2. ii. Write a note on NFA and compare with DFA****

### ✅ ****Definition of NFA (Nondeterministic Finite Automaton):****

An NFA is a **finite state machine** where:

* For a given state and input symbol, **multiple transitions are allowed**.
* It may also have **ε-transitions** (transitions without consuming input).
* If **any one path** leads to acceptance, the NFA accepts the string.

### ⚙️ ****Formal Definition of NFA:****

An NFA is a 5-tuple (Q,Σ,δ,q0,F)(Q, \Sigma, \delta, q\_0, F)(Q,Σ,δ,q0​,F), where:

* QQQ is a finite set of states
* Σ\SigmaΣ is the input alphabet
* δ:Q×Σ∪{ε}→2Q\delta: Q \times \Sigma \cup \{\varepsilon\} \rightarrow 2^Qδ:Q×Σ∪{ε}→2Q (can go to multiple states)
* q0∈Qq\_0 \in Qq0​∈Q is the initial state
* F⊆QF \subseteq QF⊆Q is the set of accepting states

### 🔄 ****Comparison Between DFA and NFA:****

| **Feature** | **DFA** | **NFA** |
| --- | --- | --- |
| Determinism | Only one possible transition per input | Multiple or zero transitions allowed |
| Transition | δ:Q×Σ→Q\delta: Q \times \Sigma \rightarrow Qδ:Q×Σ→Q | δ:Q×Σ→2Q\delta: Q \times \Sigma \rightarrow 2^Qδ:Q×Σ→2Q |
| ε-transitions | Not allowed | Allowed |
| Complexity | Simple to simulate | Easier to design, harder to simulate |
| Acceptance | Single path must be valid | Any one path can lead to acceptance |
| Speed | Fast, linear-time processing | May explore multiple paths (nondeterministic) |
| Power | **Same** — both accept Regular Languages | **Same** — both accept Regular Languages |

### ✅ ****Conclusion:****

* DFA is deterministic, simple, and efficient for real-time systems.
* NFA provides flexibility and is often easier to construct for complex patterns.
* **Every NFA has an equivalent DFA**, though the DFA might have exponentially more states.

## ****3. Construct a minimized DFA from the regular expression:****

### ****RE = (x + y) x (x + y)\*****

### ✅ ****Step 1: Understand the Language****

The regular expression can be broken down as:

1. **(x + y)** → First symbol is either **x or y**
2. **x** → Second symbol must be **x**
3. **(x + y)\*** → Followed by **any number (including zero) of x or y**

### ✅ ****Accepted Strings****

The string must:

* Be of length ≥ 2
* **Second symbol is x**
* First symbol can be x or y
* Rest can be anything (or even nothing)

### ✅ ****Examples of valid strings:****

* xx ✅
* yx ✅
* xxyy ✅
* yxxxx ✅

## 🎯 Language L = { w ∈ {x, y} | second symbol is x and length ≥ 2 }\*

Now we’ll design a **DFA** that accepts exactly this.

## ✅ ****Step 2: Construct the DFA****

We’ll define the states and transitions.

### 💡 ****Idea:****

Track the position of symbols until we check if the **second symbol is x**.

### 🔁 ****States:****

* **q0**: Start state (waiting for the first symbol)
* **q1x**: First symbol is **x**
* **q1y**: First symbol is **y**
* **qacc**: Accepting state (second symbol is **x**, now anything can follow)
* **qrej**: Rejecting state (second symbol is **not x**)

### 🧭 ****Transitions:****

| **Current State** | **Input** | **Next State** | **Meaning** |
| --- | --- | --- | --- |
| q0 | x | q1x | First symbol x |
| q0 | y | q1y | First symbol y |
| q1x | x | qacc | Second symbol x → accept |
| q1x | y | qrej | Second is y → reject |
| q1y | x | qacc | Second symbol x → accept |
| q1y | y | qrej | Second is y → reject |
| qacc | x/y | qacc | Loop, string already accepted |
| qrej | x/y | qrej | Dead state |

### ✅ ****Final DFA Diagram (Described):****

text

CopyEdit

q0

|--x--> q1x

|--y--> q1y

q1x

|--x--> qacc (accept)

|--y--> qrej

q1y

|--x--> qacc (accept)

|--y--> qrej

qacc --x/y--> qacc (accepting loop)

qrej --x/y--> qrej (dead state)

**Accepting state: qacc**

## ✅ ****Step 3: Minimize the DFA****

This DFA is already minimized because:

* All states are distinguishable (based on 2nd input and transitions)
* No unnecessary or unreachable states

## ✅ ****Step 4: Trace the string**** w = xxyx

We simulate step by step:

* **Start at q0**
* Read **x** → move to **q1x**
* Read **x** → move to **qacc** (second symbol is x ✅)
* Read **y** → stay in **qacc**
* Read **x** → stay in **qacc**

### ✅ Final State = qacc (Accepting) ⇒ ****Accepted****

## ✅ ****Final Answer Summary:****

* DFA accepts strings where the **second symbol is x**, and the string is **at least 2 characters long**.
* The DFA has 5 states: **q0, q1x, q1y, qacc (accept), qrej (reject)**
* The string **"xxyx"** is accepted.

**UNIT II REGULAR EXPRESSIONS AND LANGUAGES**

## ****1. State and explain the conversion of DFA into Regular Expression using Arden’s Theorem. Illustrate with an example.****

## ✅ ****Arden’s Theorem:****

Let **P** and **Q** be two regular expressions over an alphabet. The equation of the form:

R=Q+RPR = Q + RPR=Q+RP

has a unique solution:

R=QP∗R = QP^\*R=QP∗

**Conditions:**

* P must **not contain ε** (empty string)
* The equation must be **linear in R**

## ✅ ****Use in DFA to RE Conversion:****

We apply Arden’s theorem to solve the set of equations derived from DFA transitions. These equations represent the language accepted from each state.

### ✳️ ****Steps to Convert DFA to Regular Expression:****

1. **Label** each state (e.g., q0, q1, ..., qn)
2. **Write equations** for each state representing strings that take you from that state to a final state
3. **Use Arden’s theorem** to solve the equations step-by-step
4. The regular expression obtained for the **initial state** (after solving) is the **RE equivalent of the DFA**

## ✅ ****Example: Convert DFA to RE using Arden's Theorem****

### 🧠 ****DFA:****

States = {q0, q1}  
Alphabet = {0,1}  
Start state = q0  
Final state = q1

Transitions:

* δ(q0, 0) = q0
* δ(q0, 1) = q1
* δ(q1, 0) = q0
* δ(q1, 1) = q1

### 🧮 ****Step 1: Write equations****

Let:

* **R0** = regular expression for language from q0 to final state
* **R1** = regular expression from q1 to final state

Since q1 is a final state, we add **ε** to its equation.

From transitions:

* R0 = 0R0 + 1R1
* R1 = 0R0 + 1R1 + ε

### 🧮 ****Step 2: Solve R1 using Arden’s theorem****

R1=0R0+1R1+ε⇒R1=0R0+ε+1R1⇒R1=(0R0+ε)(1)∗R1 = 0R0 + 1R1 + ε \Rightarrow R1 = 0R0 + ε + 1R1 \Rightarrow R1 = (0R0 + ε)(1)^\*R1=0R0+1R1+ε⇒R1=0R0+ε+1R1⇒R1=(0R0+ε)(1)∗

### 🧮 ****Step 3: Substitute R1 in R0****

R0 = 0R0 + 1R1  
Substitute R1:

R0=0R0+1(0R0+ε)1∗=0R0+(10R0+1)1∗R0 = 0R0 + 1(0R0 + ε)1^\* = 0R0 + (10R0 + 1)1^\*R0=0R0+1(0R0+ε)1∗=0R0+(10R0+1)1∗

Now group R0 terms:

R0=0R0+10R0(1∗)+1(1∗)=R0(0+10(1∗))+1(1∗)R0 = 0R0 + 10R0(1^\*) + 1(1^\*) = R0(0 + 10(1^\*)) + 1(1^\*)R0=0R0+10R0(1∗)+1(1∗)=R0(0+10(1∗))+1(1∗)

Now apply Arden’s theorem:

R0=1(1∗)(0+10(1∗))∗R0 = 1(1^\*)(0 + 10(1^\*))^\*R0=1(1∗)(0+10(1∗))∗

### ✅ ****Final RE for DFA:****

Since q0 is the start state and q1 is final, the equivalent **Regular Expression** is:

R0=1(1∗)(0+10(1∗))∗R0 = 1(1^\*)(0 + 10(1^\*))^\*R0=1(1∗)(0+10(1∗))∗

This regular expression represents all strings that reach final state q1 from start state q0.

## 🟩 ****Conclusion:****

* Arden’s Theorem is a powerful method to convert DFA into a Regular Expression.
* We form equations from state transitions.
* We solve the system using substitution and Arden’s theorem.
* Final solution for the start state gives the equivalent RE.

**Q2. Construct Finite Automaton (FA) equivalent to the Regular Expression (ab + a)\* c**

**✅ Step 1: Understand the Regular Expression**

(ab+a)∗c(ab + a)^\* c(ab+a)∗c

This expression describes:

* **(ab + a)\*** → zero or more repetitions of:
  + Either **"ab"** or **"a"**
* Followed by a **single 'c'**

**✅ Language Examples:**

* c ✅
* ac ✅
* abc ✅
* aabc ✅
* ababac ✅
* But **ab or a** alone are ❌ (no c at the end)

**✅ Step 2: Break Down into Sub-Automata**

We'll create small automata and combine them:

**🔹 A. For "a"**

Simple transition:

* **q0 --a→ q1**

**🔹 B. For "ab"**

* **q0 --a→ q1 --b→ q2**

**🔹 C. For (ab + a)**

We combine transitions from A and B using union:

* **q0 --a→ q1 (for "a")**
* **q0 --a→ q3 --b→ q4 (for "ab")**

(You need two different paths for "a" and "ab")

**🔹 D. Apply Kleene star (\*) on (ab + a)**

* Create a loop from accepting states (q1, q4) back to q0

**🔹 E. Final 'c'**

* From q0 (after any number of repetitions), move via **c** to accepting state

**✅ Step 3: Build Final FA**

We'll define:

**🔸 States:**

* q0 → Start
* q1 → after 'a' (accepting 'a' path)
* q2 → after 'ab' (accepting 'ab' path)
* q3 → middle of 'ab' (after 'a')
* qf → final accepting state after reading 'c'

**🔸 Transitions:**

| **From** | **Input** | **To** |
| --- | --- | --- |
| q0 | a | q1 |
| q0 | a | q3 |
| q3 | b | q2 |
| q1 | ε | q0 |
| q2 | ε | q0 |
| q0 | c | qf |

**✅ Accepting State: qf**

**✅ Step 4: Description of Final FA Flow**

text

CopyEdit

a a

q0 ──► q1 ──► q3 ──► q2

▲ │ε b │ε │

│ └──────────┘ │

└───────────────┐ │

▼ ▼

q0 --c--> qf (final)

**✅ Conclusion:**

The finite automaton accepts strings generated by the regular expression:

(ab+a)∗c(ab + a)^\*c(ab+a)∗c

It allows repeated combinations of "a" and "ab" in any order, followed by a single 'c' to accept.

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

**1. What is Ambiguous Grammar? Explain with an Example.**

**Definition:**

A **grammar is said to be ambiguous** if there exists **at least one string** in the language generated by the grammar that **has more than one distinct leftmost derivation, rightmost derivation, or parse tree**.

**✅ Formal Definition:**

A context-free grammar (CFG) **G = (V, Σ, P, S)** is **ambiguous** if:

∃ string **w ∈ L(G)** such that **w has more than one parse tree** (or derivation).

**✅ Why Ambiguity Matters:**

* Ambiguity causes **confusion** in interpreting the structure of strings.
* Compilers and interpreters **require unambiguous grammars** to parse code correctly.
* **Ambiguity makes parsing difficult** and leads to **incorrect program interpretation.**

**✅ Example of Ambiguous Grammar:**

Consider the grammar **G**:

**Productions:**

mathematica

CopyEdit

E → E + E

E → E \* E

E → (E)

E → id

**Let’s parse the string:**

id+id∗idid + id \* idid+id∗id

**✅ First Parse Tree (Addition First):**

mathematica

CopyEdit

E

/|\

E + E

| |

id E \* E

| |

id id

* Interpretation: **(id + id) \* id**

**✅ Second Parse Tree (Multiplication First):**

mathematica

CopyEdit

E

/|\

E \* E

| |

E + E id

| |

id id

* Interpretation: **id + (id \* id)**

**✅ Conclusion:**

* The string **id + id \* id** has **two different parse trees**.
* Therefore, the grammar is **ambiguous**.

**✅ How to Remove Ambiguity (Briefly):**

To remove ambiguity, rewrite the grammar by introducing rules that respect **operator precedence and associativity**.

Example unambiguous version:

r

CopyEdit

E → E + T | T

T → T \* F | F

F → (E) | id

This ensures:

* \* has **higher precedence** than +
* Left associativity is maintained

**✅ Summary:**

* An **ambiguous grammar** generates at least one string with **more than one parse tree**.
* It creates problems in parsing and must be **rewritten to be unambiguous**, especially in **compiler design**.
* Example: The expression id + id \* id shows ambiguity with multiple interpretations.

2**. Construct a CFG to generate even and odd set of palindromes over the alphabet {a, b}.**

**✅ What is a Palindrome?**

A **palindrome** is a string that reads the same forward and backward.

Examples over {a, b}:

* Even-length palindromes: aa, bb, abba, baab, etc.
* Odd-length palindromes: a, b, aba, bab, aabaa, etc.

**✅ Context-Free Grammar (CFG) Definition**

A CFG is defined as:

G=(V,Σ,P,S)G = (V, \Sigma, P, S)G=(V,Σ,P,S)

Where:

* **V** = set of variables (non-terminals)
* **Σ** = set of terminals = {a, b}
* **P** = set of production rules
* **S** = start symbol

**✅ CFG to Generate All Palindromes over {a, b}**

We’ll define separate rules to generate:

**🔸 1. Even-length palindromes**

Even palindromes have matching characters from both ends and an empty center.

**Grammar:**

nginx

CopyEdit

S → aSa | bSb | ε

* Starts and ends with the same character
* Repeats recursively
* Stops with **ε** (empty string)

**🔸 2. Odd-length palindromes**

Odd palindromes have a **single middle character**.

**Grammar:**

less

CopyEdit

S → aSa | bSb | a | b

This handles:

* a, b as the center of the palindrome
* Grows symmetrically outward

**✅ Combined CFG (Even and Odd Palindromes)**

To generate **both even and odd** palindromes over {a, b}:

less

CopyEdit

S → aSa | bSb | a | b | ε

**✅ Explanation with Examples:**

**Even Palindromes:**

* ε → valid
* aSa → aa (S = ε)
* bSb → bb
* aSa → abba (S → bSb → ε)

**Odd Palindromes:**

* a and b → base cases
* aSa → aba (S = b)
* bSb → bab
* aSa → aabaa (S = aba)

**✅ Conclusion:**

* The grammar:

less

CopyEdit

S → aSa | bSb | a | b | ε

generates **all even and odd palindromes** over {a, b}.

* The CFG is **recursive**, capturing the symmetric nature of palindromes.

**UNIT IV NORMAL FORMS AND TURING MACHINES**

**Q: Convert the given grammar into Chomsky Normal Form (CNF)**

*(E Nov–Dec 2017)*

**✅ Given Grammar:**

less

CopyEdit

S → bA | aB

A → bAA | aS | a

B → aBB | bS | b

**✅ Step-by-Step CNF Conversion**

**✳️ Chomsky Normal Form (CNF) Rules:**

* All productions must be of the form:
  1. **A → BC** (where B, C are non-terminals)
  2. **A → a** (where a is a terminal)
* No ε-productions (unless S → ε and ε ∈ L(G))
* No unit productions (like A → B)
* No mixed terminals with non-terminals

**✅ Step 1: Replace terminals in RHS longer productions**

Introduce new variables for terminals:

css

CopyEdit

Xb → b

Xa → a

**✅ Step 2: Rewrite Grammar with new variables**

Now we rewrite all RHS with length > 1 using only variables:

Original → After Substitution

* S → bA becomes S → Xb A
* S → aB becomes S → Xa B
* A → bAA becomes A → Xb A1, where A1 → A A
* A → aS becomes A → Xa S
* A → a becomes A → Xa
* B → aBB becomes B → Xa B1, where B1 → B B
* B → bS becomes B → Xb S
* B → b becomes B → Xb

**✅ Step 3: Final CNF Grammar**

less

CopyEdit

S → Xb A | Xa B

A → Xb A1 | Xa S | Xa

A1 → A A

B → Xa B1 | Xb S | Xb

B1 → B B

Xb → b

Xa → a

**✅ Conclusion:**

This grammar is now in **Chomsky Normal Form (CNF)**.  
Every production is either of the form:

* **A → BC** (two non-terminals)
* **A → a** (single terminal via helper variables)

**Q46. Prove that the Halting Problem is Undecidable**

**✅ What is the Halting Problem?**

The **Halting Problem** is the decision problem of determining, given a program and its input, **whether the program will halt (terminate) or run forever.**

**🔹 Formally:**

Given a Turing machine **M** and an input string **w**,  
determine whether **M halts on w**.

**✅ Statement of the Halting Problem:**

There is **no Turing machine** that can decide whether any arbitrary Turing machine **M** halts on input **w**.

**✅ Goal: Prove Halting Problem is Undecidable**

We will prove it using a **proof by contradiction**.

**✅ Proof (By Contradiction — Turing’s Original Idea):**

**🔸 Step 1: Assume a machine H exists**

Suppose there is a Turing machine **H(M, w)** that can decide the Halting Problem.

It behaves as follows:

graphql

CopyEdit

H(M, w) =

accepts, if M halts on input w

rejects, if M runs forever on input w

**🔸 Step 2: Construct another machine D**

We now build a new Turing machine **D** that uses **H** as a subroutine:

java

CopyEdit

D(M):

if H(M, M) accepts: // M halts on its own description

loop forever

else:

halt

So:

* **D(M)** does the opposite of what **M(M)** would do.

**🔸 Step 3: What happens when we run D(D)?**

Let’s analyze **D(D)** (feeding D its own description):

* If **H(D, D)** says D halts → D loops forever.
* If **H(D, D)** says D does not halt → D halts.

💥 **Contradiction!**

* If **D(D)** halts, then it must loop forever.
* If **D(D)** loops forever, then it must halt.

**✅ Conclusion:**

Our assumption that such a machine **H** exists leads to a **logical contradiction**.  
Hence, no such machine **H** can exist.

**🔴 Therefore, the Halting Problem is Undecidable.**

**✅ Implications:**

* No program can universally predict whether **any program** halts on any input.
* One of the first results showing the **limits of computation**.

**UNIT V UNDECIDABILITY**

**Q1. Prove that: *If L is a recursive language, then L′ (complement of L) is also a recursive language.***

*(E – typically refers to an exam question)*

**✅ Definitions:**

**🔹 Recursive Language:**

A language **L** is **recursive** if there exists a **Turing Machine (TM)** **M** such that:

* For every input string **w**:
  + If **w ∈ L**, then **M accepts w** and halts.
  + If **w ∉ L**, then **M rejects w** and halts.

✅ That means: **M always halts** with a **yes or no answer**.

**🔹 Complement of L (denoted L′):**

L′ = { w | w ∉ L }

**✅ To Prove:**

If **L is recursive**, then **L′** is also recursive.

**✅ Proof:**

Let **M** be a Turing Machine that decides **L**.  
So, for any input string **w**, **M halts and:**

* Accepts if **w ∈ L**
* Rejects if **w ∉ L**

**🔸 Step 1: Construct a New TM M′ to decide L′**

We define a machine **M′** as:

mathematica

CopyEdit

On input w:

Run M on w

If M accepts → reject

If M rejects → accept

**🔸 Step 2: Why is M′ a Decider?**

* Since **M is a decider**, it always halts.
* Therefore, **M′ also halts** for every input.
* Hence, **M′ is a decider** for **L′**.

**✅ Conclusion:**

Since **M′** halts on all inputs and decides L′:

✅ **L′ is recursive**

**✅ Hence Proved:**

If **L is a recursive language**, then **L′ is also recursive.**

**📝 Remarks:**

* Recursive languages are **closed under complement**.
* This is **not true for recursively enumerable (RE) languages**, where the complement may not be RE.

**2. Is the problem of determining whether a given recursively enumerable (RE) language is empty or not decidable? Justify your answer.**

*(AN = Answer)*

**✅ Definitions:**

1. **Recursively Enumerable (RE) Language:**
   * A language **L** is **recursively enumerable** (RE) if there exists a **Turing machine (TM)** **M** such that:
     + **M** accepts every string **w ∈ L**, and
     + If **w ∉ L**, then **M** may either reject **w** or loop forever.
2. **Empty Language:**
   * A language **L** is **empty** if it contains no strings, i.e., **L = ∅**.

**✅ Problem Definition:**

We are tasked with determining whether a given RE language is **empty** or not. In other words, we need to decide whether a Turing machine **M** accepts **no strings**.

**✅ Is This Problem Decidable?**

**No**, the problem of determining whether a given recursively enumerable (RE) language is empty is **undecidable**. Here's why:

**✅ Proof (Using Reduction from the Halting Problem):**

To show that this problem is undecidable, we can **reduce** the **Halting Problem** to the problem of determining whether a given RE language is empty.

**🔸 Step 1: Assume we have a decider for the empty language problem.**

Suppose there exists a Turing machine **H** that can decide if a given RE language is empty. That means for any Turing machine **M**, **H(M)** will tell us whether the language accepted by **M** is empty (i.e., whether **M** accepts no strings).

* If **H(M) = "yes"**, the language of **M** is empty (no strings are accepted by **M**).
* If **H(M) = "no"**, the language of **M** is non-empty (there exists at least one string accepted by **M**).

**🔸 Step 2: Construct a new Turing machine based on this assumption.**

We will use the machine **H** to decide the **Halting Problem**, which is known to be undecidable. Specifically, the Halting Problem asks whether a Turing machine **M** halts on a given input **w**.

We construct the following machine **M′**:

* **M′** on input **w**:
  1. Simulate **M** on input **w**.
  2. If **M** halts, then **M′** accepts the string **w** (i.e., **M′**'s language is non-empty).
  3. If **M** does not halt, then **M′** does not accept anything (i.e., **M′**'s language is empty).

**🔸 Step 3: Apply the Decider for Empty Language Problem.**

* If we apply the hypothetical decider **H** to **M′**, we get the following:
  + If **H(M′) = "no"**, then the language of **M′** is **non-empty**, which implies that **M** halts on input **w**.
  + If **H(M′) = "yes"**, then the language of **M′** is empty, which implies that **M** does not halt on input **w**.

Thus, we can use **H** to decide whether **M** halts on **w**, which is equivalent to solving the **Halting Problem**.

**✅ Conclusion:**

Since the Halting Problem is undecidable, and we have reduced it to the problem of determining whether an RE language is empty, we can conclude that the problem of determining whether a given RE language is empty is also **undecidable**.

**🔴 Therefore, the problem of determining whether a given recursively enumerable (RE) language is empty is undecidable.**

**📝 Remarks:**

* **Undecidable problems**: In computer science, many problems are undecidable, meaning no algorithm can decide the problem for all inputs.
* This result highlights the inherent limitations of computation and the fact that not all questions about Turing machines and languages can be answered algorithmically.