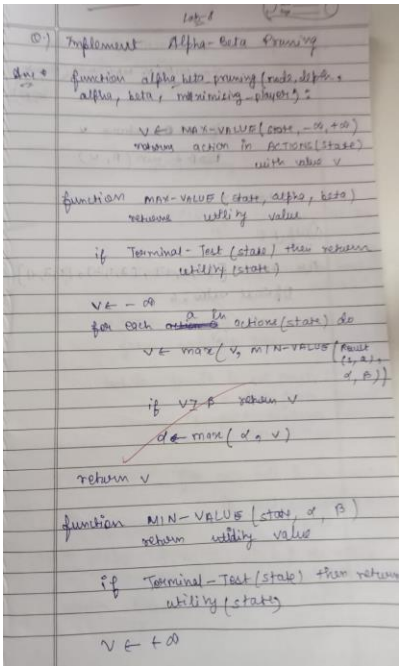
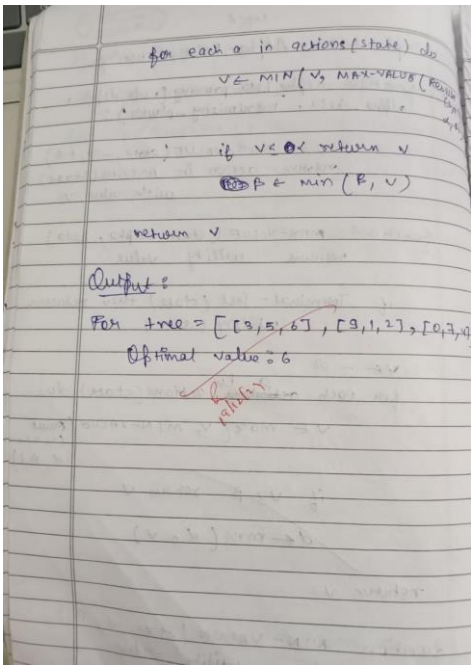
1.) ALPHA BETA PRUNING :

```python
import math

def alpha_beta_pruning(depth, node_index, is_maximizing_player, values, alpha, beta, max_depth):
    # Base case: when the maximum depth is reached
    if depth == max_depth:
        return values[node_index]

    if is_maximizing_player:
        best = -math.inf
        # Recur for left and right children
        for i in range(2):
            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            # Prune the remaining nodes if beta <= alpha
            if beta <= alpha:
                break
        return best
    else:
        best = math.inf
        # Recur for left and right children
        for i in range(2):
            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)
            # Prune the remaining nodes if beta <= alpha
            if beta <= alpha:
                break
        return best
```

```python
print("Vyom Gupta (1BM22CS333):")

# Example usage
if __name__ == "__main__":
    # Example tree represented as a list of leaf node values
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    max_depth = 3  # Height of the tree
    result = alpha_beta_pruning(0, 0, True, values, -math.inf, math.inf, max_depth)
    print("The optimal value is:", result)
```

```
Vyom Gupta (1BM22CS333):
The optimal value is: 5


...Program finished with exit code 0
Press ENTER to exit console.
```

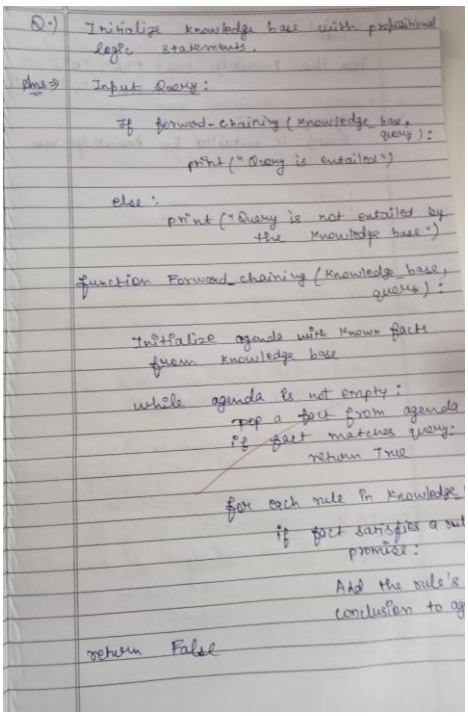


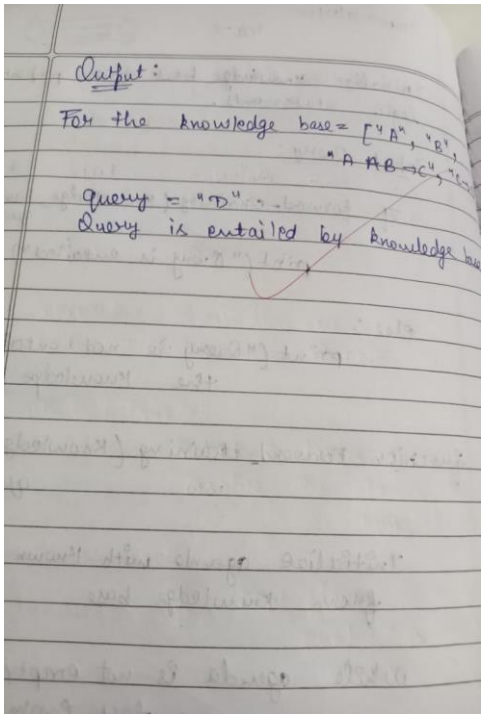


2.) Propositoinal Logic Statement Entailment

```python
from sympy.logic.boolalg import Or, And, Not
from sympy.abc import A, B, C, D, E, F
from sympy import simplify_logic

def is_entailment(kb, query):
    negated_query = Not(query)
    kb_with_negated_query = And(*kb, negated_query)  # Combine all KB clauses and the negated query
    simplified_kb = simplify_logic(kb_with_negated_query, form="cnf")

    # If the simplified KB evaluates to False, the query is entailed
    return simplified_kb == False

# Define a larger Knowledge Base (kb)
kb = [
    Or(A, B),        # A ∨ B
    Or(Not(A), C),   # ¬A ∨ C
    Or(Not(B), D),   # ¬B ∨ D
    Or(Not(D), E),   # ¬D ∨ E
    Or(Not(E), F),   # ¬E ∨ F
]

# Query to check (C ∨ F)
query = Or(C, F)

# Check entailment
result = is_entailment(kb, query)

# Output the result with Vyom Gupta's details
print("Vyom Gupta (1BM22CS333)\n")
print(f"Is the query '{query}' entailed by the knowledge base? {'Yes' if result else 'No'}")
```

```
Vyom Gupta (1BM22CS333)

Is the query 'C | F' entailed by the knowledge base? Yes
```

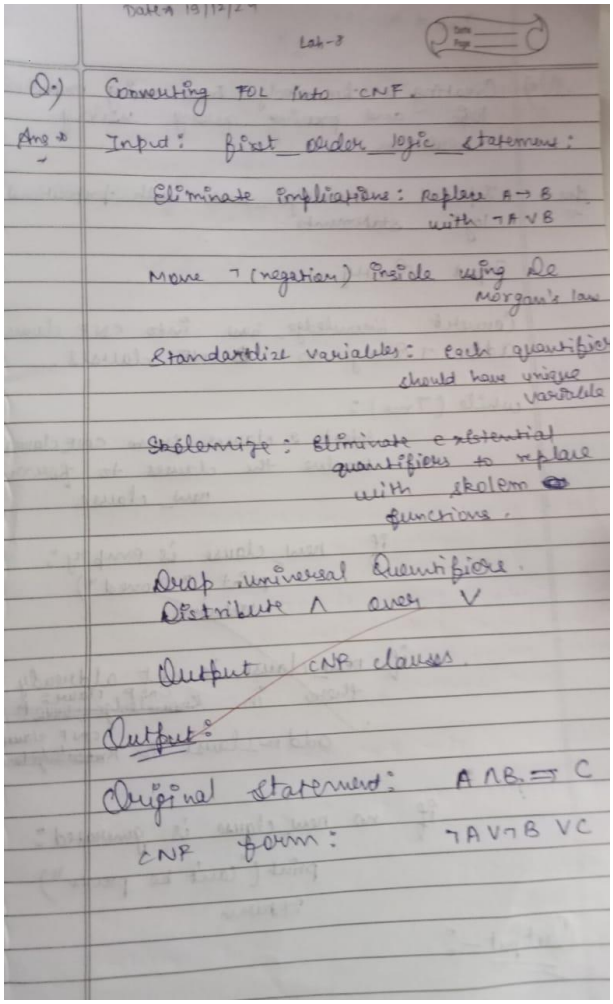




3.) FOL TO CNF

```
from sympy import symbols, Not, Or, And, Implies, Equivalent
from sympy.logic.boolalg import to_cnf
def fol_to_cnf(fol_expr):
    fol_expr = fol_expr.replace(Equivalent, lambda a, b: And(Implies(a, b), Implies(b, a)))
    fol_expr = fol_expr.replace(Implies, lambda a, b: Or(Not(a), b))
    cnf_form = to_cnf(fol_expr, simplify=True)
    return cnf_form
def main():
    P = symbols("P")
    Q = symbols("Q")
    R = symbols("R")

    fol_expr1 = Implies(P, Q)
    print("Example 1: P → Q")
    print("Original FOL Expression:")
    print(fol_expr1)
    cnf1 = fol_to_cnf(fol_expr1)
    print("\nCNF Form:")
    print(cnf1)

    fol_expr2 = Implies(Or(P, Not(Q)), Or(Q, R))
    print("\nExample 2: (P ∨ ¬Q) → (Q ∨ R)")
    print("Original FOL Expression:")
    print(fol_expr2)
    cnf2 = fol_to_cnf(fol_expr2)
    print("\nCNF Form:")
    print(cnf2)

print("Vyom Gupta (1BM22CS333)\n")
if __name__ == "__main__":
```

```
Vyom Gupta (1BM22CS333)

Example 1: P → Q
Original FOL Expression:
Implies(P, Q)

CNF Form:
Q | ~P

Example 2: (P ∨ ¬Q) → (Q ∨ R)
Original FOL Expression:
Implies(P | ~Q, Q | R)

CNF Form:
Q | R
```



4.) PROVING BY RESOLUTION :

```python
def negation(p):
    if p.startswith("~"):
        return p[1:]
    return f"~{p}"

def resolution(kb, query):
    kb.append(negation(query))
    new_clauses = set(kb)
    print(f"Initial Knowledge Base + negation of query: {kb}")
    while True:
        added_new_clause = False
        clauses = list(new_clauses)
        for i in range(len(clauses)):
            for j in range(i + 1, len(clauses)):
                clause1 = clauses[i]
                clause2 = clauses[j]
                resolvent = resolve(clause1, clause2)
                if resolvent is not None:
                    print(f"Resolving clauses: {clause1} and {clause2}")
                    print(f"Resolved to: {resolvent}")
                    if not resolvent:
                        return True
                    if resolvent not in new_clauses:
                        new_clauses.add(resolvent)
                        added_new_clause = True
        if not added_new_clause:
            break
    return False

def resolve(clause1, clause2):
```

```python
def resolve(clause1, clause2):
    literals1 = set(clause1.split(" v "))
    literals2 = set(clause2.split(" v "))
    for literal in literals1:
        neg_literal = negation(literal)
        if neg_literal in literals2:
            new_clause = literals1.union(literals2) - {literal, neg_literal}
            return " v ".join(sorted(new_clause))
    return None

kb = [
    "P v Q",
    "~P v R",
    "Q v ~R",
    "R v T"
]

query = "T"

result = resolution(kb, query)

if result:
    print(f"\nQuery '{query}' is provable from the knowledge base.")
else:
    print(f"\nQuery '{query}' is not provable from the knowledge base.")
```

```
Vyom Gupta (1BM22CS333)

Initial Knowledge Base + negation of query: ['P v Q', '~P v R', 'Q v ~R', 'R v T', '~T']
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
Resolving clauses: Q v ~R and ~P v R
Resolved to: Q v ~P
Resolving clauses: Q v ~R and R v T
Resolved to: Q v T
Resolving clauses: ~T and R v T
Resolved to: R
Resolving clauses: Q v R and Q v ~R
Resolved to: Q
Resolving clauses: P v Q and Q v ~P
Resolved to: Q
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
Resolving clauses: Q v T and ~T
Resolved to: Q
Resolving clauses: Q v ~R and ~P v R
Resolved to: Q v ~P
Resolving clauses: Q v ~R and R v T
Resolved to: Q v T
Resolving clauses: Q v ~R and R
Resolved to: Q
Resolving clauses: ~T and R v T
Resolved to: R
Resolving clauses: Q v R and Q v ~R
Resolved to: Q
Resolving clauses: P v Q and Q v ~P
Resolved to: Q
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
```

```
Resolving clauses: Q v T and ~T
Resolved to: Q
Resolving clauses: Q v ~R and ~P v R
Resolved to: Q v ~P
Resolving clauses: Q v ~R and R v T
Resolved to: Q v T
Resolving clauses: Q v ~R and R
Resolved to: Q
Resolving clauses: ~T and R v T
Resolved to: R

Query 'T' is not provable from the knowledge base.


...Program finished with exit code 0
Press ENTER to exit console.
```

Date: 19/12/24 Lab-8

Q) Creating a knowledge Base using propositional logic and proving query using resolution.

Ans ⇒

Input: Knowledge_Base with propositional logic statements

Input Query.

Convert Knowledge_base into CNF clauses
Add ¬ Query to CNF clauses.

while (True):
    Select 2 clauses from CNF clause
    Resolve the clauses to form new_clause
    if new_clause is empty:
        print ("Proved")
        return.
    if new_clause is not already there in CNF clause:
        add new_clause to CNF clause
    if no new_clause is generated:
        print ("can't be proved")
        return

Output :-

For knowledge base = ["A", "B", "A ∧ B ⇒ C", "C ⇒ D"]

Query = 'D'

Query is proven using resolution.

23/12/24