AI LAB RECORD

Potana kundana sai priya

1BM19CS112

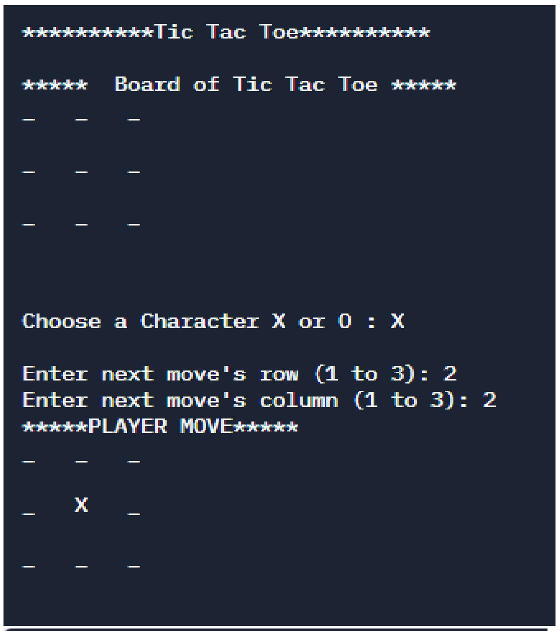
5-c

1. **Implement tic-tac-toe game.**

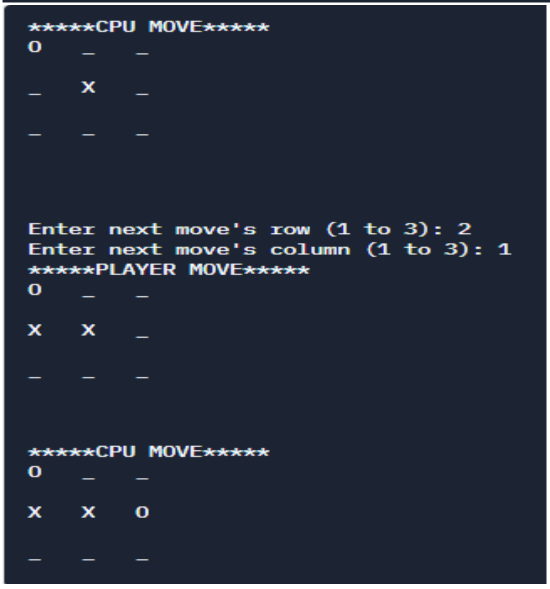
- **Program**

```
import random as r
ai,player='O','X'
board=[['_','_','_'],['_','_','_'],['_','_','_']]
weights=[[3,2,3],[2,4,2],[3,2,3]]
def init():
   global ai,player,board,weights
   ai,player='O','X'
   board=[['_','_','_'],['_','_','_'],['_','_','_']]
   weights=[[3,2,3],[2,4,2],[3,2,3]]
def move(row,col,ch):
   if board[row][col]=='_':
      board[row][col],weights[row][col]=ch,0
      return True
   else : return False

def display(move_type='board'):
   if move_type=='cpu' : print('*'*5+'CPU MOVE'+'*'*5)
   elif move_type=='board': print("*"*5+'  Board of Tic Tac Toe '+'*'*5)
   else :print('*'*5+'PLAYER MOVE'+'*'*5)
   for i in range(3):
      for j in range(3):
         print(board[i][j],end='\t')
      print('\n')
   print('\n')

def compare_line(s1,ch):
   return '_' in s1 and s1.count(ch)==2

def get_position():
   max_value=max([max(x) for x in weights])
   positions=[(i,weights[i].index(max_value)) for i in range(3)  if max_value in weights[i]]
   return positions

def has_tied():
   for row in board:
    if '_' in row: return False
   return True

def attacking_positiion(ch):
      default='_'
      for i in range(3):
          col=[board[0][i],board[1][i],board[2][i]]
          if compare_line(board[i],ch): return (i,board[i].index(default))
          elif compare_line(col,ch): return (col.index(default),i)
      diag1,diag2=[board[0][0],board[1][1],board[2][2]],[board[0][2],board[1][1],board[2][0]]
      if compare_line(diag1,ch):return (diag1.index(default),diag1.index(default))
      elif compare_line(diag2,ch): return (diag2.index(default),2-diag2.index(default))
      return False

def ai_move():
   global ai,player
   pos,f=attacking_positiion(ch=ai),False
   if pos!=False:(row,col),f=pos,True
   else :
      pos=attacking_positiion(ch=player)
      if pos!=False: row,col=pos
      else: row,col=r.choice(get_position())
   move(row,col,ai)
   return f

def run():
   global ai,player
   end,tied,move_type=False,False,None
   print('*'*10+ 'Tic Tac Toe'+'*'*10+'\n')
   display()
   ch=input('Choose a Character X or O : ')
   if ch=='O': ai,player=player,ai
   while(True):
      if tied:
          print('*'*10+'The match is tied'+'*'*10)
          return
      elif end:
          print('*'*10+move_type+' has own '+'*'*10)
          return
      move_type='player'
      r=int(input("\nEnter next move's row (1 to 3): "))
      c=int(input("Enter next move's column (1 to 3): "))
      if not move(r-1,c-1,player):
          print('\nEnter proper positions!!')
      else:
        display(move_type=move_type)
        tied=has_tied()
        if tied: continue
        move_type='cpu'
        end=ai_move()
        display(move_type=move_type)
        tied=has_tied()
def main():
   run()
   f='Y'
   while(f=='Y'or f=='y'):
      f=input('Do you want to play again Y or N: ')
      init()
      if f=='Y' or f=='y':run()
   print('\n\n'+'*'*10+' Thank You '+'*'*10)
main()
```

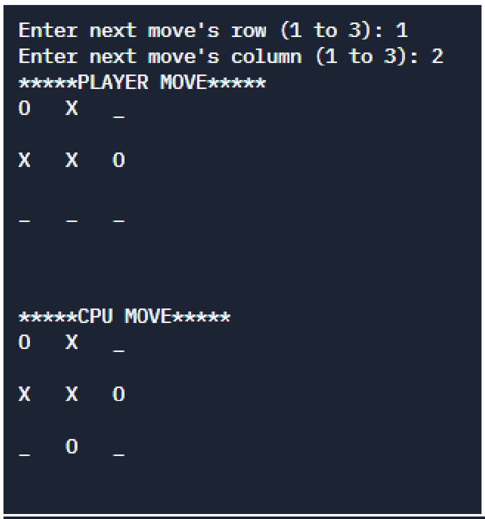
➢ **Output**



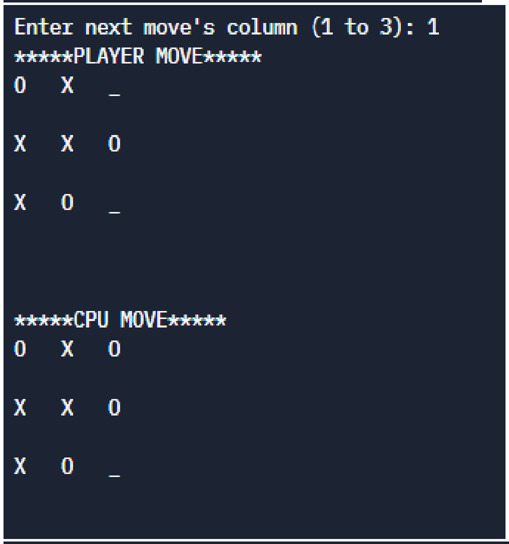
```
**********Tic Tac Toe**********

*****  Board of Tic Tac Toe *****
_   _   _

_   _   _

_   _   _



Choose a Character X or O : X

Enter next move's row (1 to 3): 2
Enter next move's column (1 to 3): 2
*****PLAYER MOVE*****
_   _   _

_   X   _

_   _   _
```



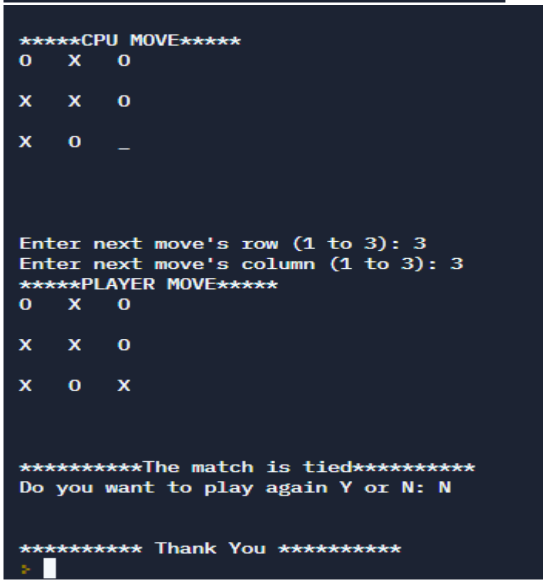
```
*****CPU MOVE*****
O   _   _

_   X   _

_   _   _



Enter next move's row (1 to 3): 2
Enter next move's column (1 to 3): 1
*****PLAYER MOVE*****
O   _   _

X   X   _

_   _   _



*****CPU MOVE*****
O   _   _

X   X   O

_   _   _
```



```
Enter next move's row (1 to 3): 1
Enter next move's column (1 to 3): 2
*****PLAYER MOVE*****
O   X   _

X   X   O

_   _   _


*****CPU MOVE*****
O   X   _

X   X   O

_   O   _
```



```
Enter next move's column (1 to 3): 1
*****PLAYER MOVE*****
O   X   _

X   X   O

X   O   _


*****CPU MOVE*****
O   X   O

X   X   O

X   O   _
```



```
*****CPU MOVE*****
O   X   O

X   X   O

X   O   _


Enter next move's row (1 to 3): 3
Enter next move's column (1 to 3): 3
*****PLAYER MOVE*****
O   X   O

X   X   O

X   O   X


**********The match is tied**********
Do you want to play again Y or N: N

********** Thank You **********
> 
```

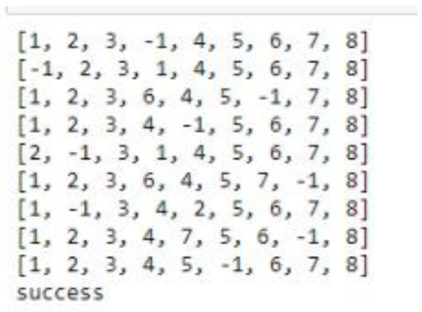
2. Solve 8 puzzle program. (bfs)

- **Program**

```
def bfs(src, target):
  queue = []
  queue.append(src)

  exp = []

  while len(queue) > 0:
    source = queue.pop(0)
    exp.append(source)

    print(source)

    if source == target:
      print("success")
      return

    poss_moves_to_do = []
    poss_moves_to_do = possible_moves(source, exp)

    for move in poss_moves_to_do:

      if move not in exp and move not in queue:
        queue.append(move)


def possible_moves(state, visited_states):
  # Find index of empty spot and assign it to b
  b = state.index(-1)

  # 'd' for down, 'u' for up, 'r' for right, 'l' for left - directions array
  d = []
  # Add all possible direction into directions array - Hint using if statements

  if b not in [0, 1, 2]:
    d.append('u')
  if b not in [6, 7, 8]:
    d.append('d')
  if b not in [0, 3, 6]:
    d.append('l')
  if b not in [2, 5, 8]:
    d.append('r')

  # If direction is possible then add state to move
  pos_moves_it_can = []
```

```
    # for all possible directions find the state if that move is played
    ### Jump to gen function to generate all possible moves in the given directions

    for i in d:
        pos_moves_it_can.append(gen(state, i, b))

    return [move_it_can for move_it_can in pos_moves_it_can if move_it_can not in
visited_states]


# Generate move for given direction
def gen(state, m, b):
    temp = state.copy()

    # if move is to slide empty spot to the left and so on

    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]

    if m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]

    if m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]

    if m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]

    # return new state with tested move to later check if "src == target"
    return temp



src = [1, 2, 3, -1, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5, -1, 6, 7, 8]

bfs(src, target)
```

- **output**



```
[1, 2, 3, -1, 4, 5, 6, 7, 8]
[-1, 2, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, -1, 7, 8]
[1, 2, 3, 4, -1, 5, 6, 7, 8]
[2, -1, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 7, -1, 8]
[1, -1, 3, 4, 2, 5, 6, 7, 8]
[1, 2, 3, 4, 7, 5, 6, -1, 8]
[1, 2, 3, 4, 5, -1, 6, 7, 8]
success
```

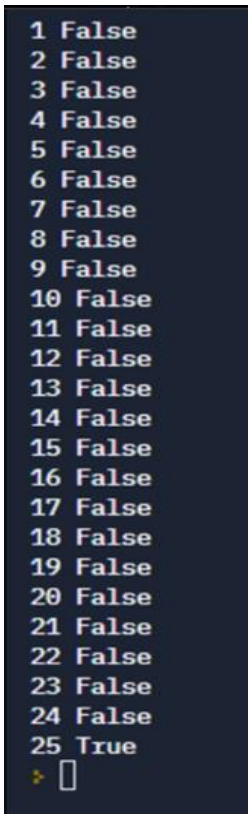
3. Implement iterative deepening search algorithm. (8 puzzle problem)

➢ **Program**

```
def dfs(src,target,limit,visited_states):
   if src == target:
      return True
   if limit <= 0:
      return False
   visited_states.append(src)
   moves = possible_moves(src,visited_states)
   for move in moves:
      if dfs(move, target, limit-1, visited_states):
         return True
   return False

def possible_moves(state,visited_states):
   b = state.index(-1)
   d = []
   if b not in [0,1,2]:
      d += 'u'
   if b not in [6,7,8]:
      d += 'd'
   if b not in [2,5,8]:
      d += 'r'
   if b not in [0,3,6]:
      d += 'l'
   pos_moves = []
   for move in d:
      pos_moves.append(gen(state,move,b))
   return [move for move in pos_moves if move not in visited_states]


def gen(state, move, blank):
   temp = state.copy()
   if move == 'u':
      temp[blank-3], temp[blank] = temp[blank], temp[blank-3]
   if move == 'd':
      temp[blank+3], temp[blank] = temp[blank], temp[blank+3]
   if move == 'r':
      temp[blank+1], temp[blank] = temp[blank], temp[blank+1]
   if move == 'l':
      temp[blank-1], temp[blank] = temp[blank], temp[blank-1]
   return temp

def iddfs(src,target,depth):
   for i in range(depth):
      visited_states = []
```

```
        if dfs(src,target,i+1,visited_states):
            return True
    return False

#Test 1
#src = [1,2,3,-1,4,5,6,7,8]
# target = [1,2,3,4,5,-1,6,7,8]

# depth = 1
# if (iddfs(src, target, depth)):
#     print("True")
# else:
#     print("false")



src = [1, 2, 3, 4, 5, 6, 7, 8, -1]
target = [-1, 1, 2, 3, 4, 5, 6, 7, 8]

for i in range(1, 100):
    val = iddfs(src,target,i)
    print(i, val)
    if val == True:
        break
```

- **output**



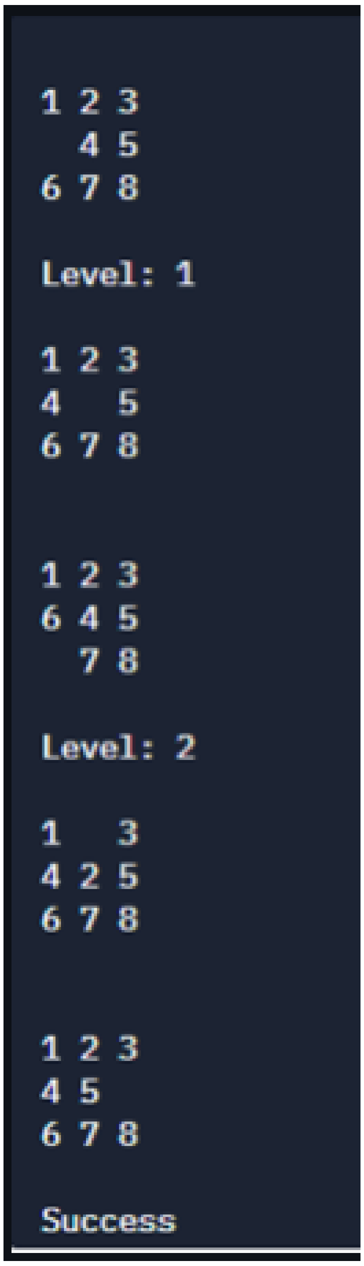
```
1 False
2 False
3 False
4 False
5 False
6 False
7 False
8 False
9 False
10 False
11 False
12 False
13 False
14 False
15 False
16 False
17 False
18 False
19 False
20 False
21 False
22 False
23 False
24 False
25 True
>
```

4. Implement A* algorithm. (8 puzzle problem)

➢ **Program**

```
def print_b(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(
        f"""
{state[0]} {state[1]} {state[2]}
{state[3]} {state[4]} {state[5]}
{state[6]} {state[7]} {state[8]}
        """
    )
def h(state, target):
    count=0
    i=0
    for j in state:
        if state[i]!= target[i]:
            count=count+1
    return count
def astar(state,target):# Add inputs if more are required
    states = [src]
    g = 0
    visited_states =[]
    while len(states):
        print(f"Level: {g}")
        moves = []
        for state in states:
            visited_states.append(state)
            print_b(state)
            if state == target:
                print("Success")
                return
            moves += [move for move in possible_moves(state, visited_states) if move not in
moves]
        costs = [g + h(move, target) for move in moves]
        states = [moves[i] for i in range(len(moves)) if costs[i] == min(costs)]
        g += 1
    print("Fail")
def possible_moves(state,visited_state):# Add inputs if more are required
    # Find index of empty spot and assign it to b
    b = state.index(-1);

    #'d' for down, 'u' for up, 'r' for right, 'l' for left - directions array
    d = []

    #Add all possible direction into directions array - Hint using if statements
    if b - 3 in range(9):
      d.append('u')
    if b not in [0,3,6]:
      d.append('l')
    if b not in [2,5,8]:
      d.append('r')
    if b + 3 in range(9):
      d.append('d')

    # If direction is possible then add state to move
    pos_moves = []

    # for all possible directions find the state if that move is played
    ### Jump to gen function to generate all possible moves in the given directions
    for m in d:
      pos_moves.append(gen(state, m, b))

    # return all possible moves only if the move not in visited_states
    return [move for move in pos_moves if move not in visited_state]
def gen(state,m, b):
  temp = state.copy()

  # if move is to slide empty spot to the left and so on
  if m == 'u': temp[b-3] , temp[b] = temp[b], temp[b-3]
  if m == 'l': temp[b-1] , temp[b] = temp[b], temp[b-1]
  if m == 'r': temp[b+1] , temp[b] = temp[b], temp[b+1]
  if m == 'd': temp[b+3] , temp[b] = temp[b], temp[b+3]

    # return new state with tested move to later check if "src == target"
  return temp
#Test 1
src = [1,2,3,-1,4,5,6,7,8]
target = [1,2,3,4,5,-1,6,7,8]




astar(src, target)
```

➢ **output**



```
1 2 3
  4 5
6 7 8

Level: 1

1 2 3
4   5
6 7 8


1 2 3
6 4 5
  7 8

Level: 2

1   3
4 2 5
6 7 8


1 2 3
4 5
6 7 8

Success
```

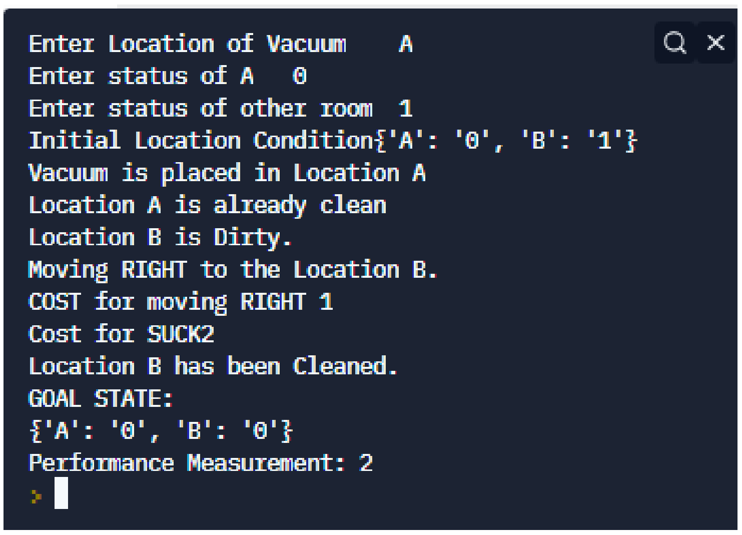
5. **Implement vacuum cleaner problem**

➢ **Program**

```
def vacuum_world():
        # initializing goal_state
        # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum \t") #user_input of
location vacuum is placed
    status_input = input("Enter status of"+" " + location_input + "\t")
#user_input if location is dirty or clean
    status_input_complement = input("Enter status of other room \t")
    initial_state = {'A' : status_input , 'B' : status_input_complement}
    print("Initial Location Condition" + str(initial_state))

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['A'] = '0'
            cost += 1                      #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                # if B is Dirty
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1                       #cost for moving right
                print("COST for moving RIGHT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1                       #cost for suck
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action" + str(cost))
                # suck and mark clean
                print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':# if B is Dirty
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1                       #cost for moving right
                print("COST for moving RIGHT " + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1                       #cost for suck
                print("Cost for SUCK" + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action " + str(cost))
                print(cost)
                # suck and mark clean
                print("Location B is already clean.")

    else:
        print("Vacuum is placed in location B")
        # Location B is Dirty.
        if status_input == '1':
            print("Location B is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['B'] = '0'
            cost += 1  # cost for suck
            print("COST for CLEANING " + str(cost))
            print("Location B has been Cleaned.")

            if status_input_complement == '1':
                # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1  # cost for moving right
                print("COST for moving LEFT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("COST for SUCK " + str(cost))
                print("Location A has been Cleaned.")

        else:
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

            if status_input_complement == '1':  # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1  # cost for moving right
                print("COST for moving LEFT " + str(cost))
                # suck the dirt and mark it as clean
```

```
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("Cost for SUCK " + str(cost))
                print("Location A has been Cleaned. ")
            else:
                print("No action " + str(cost))
                # suck and mark clean
                print("Location A is already clean.")

    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))


vacuum_world()
```

➢ **output**



```
Enter Location of Vacuum    A
Enter status of A   0
Enter status of other room  1
Initial Location Condition{'A': '0', 'B': '1'}
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
> 
```

6. **Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.**

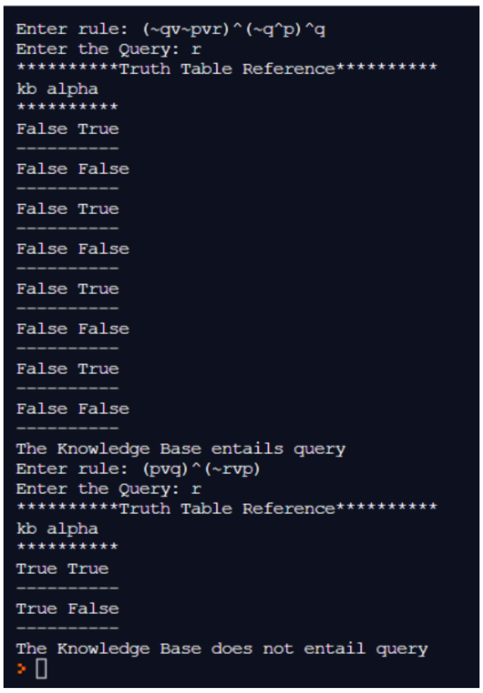
- **Program**

```
combinations=[(True,True,
True),(True,True,False),(True,False,True),(True,False, False),(False,True,
True),(False,True, False),(False, False,True),(False,False, False)]
variable={'p':0,'q':1, 'r':2}
kb=''
q=''
priority={'~':3,'v':1,'^':2}
def input_rules():
    global kb, q
    kb = (input("Enter rule: "))
    q = input("Enter the Query: ")
def entailment():
    global kb, q
    print('*'*10+"Truth Table Reference"+'*'*10)
    print('kb','alpha')
    print('*'*10)
    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb), comb)
        f = evaluatePostfix(toPostfix(q), comb)
        print(s, f)
        print('-'*10)
        if s and not f:
            return False
    return True
def isOperand(c):
    return c.isalpha() and c!='v'

def isLeftParanthesis(c):
    return c == '('

def isRightParanthesis(c):
    return c == ')'

def isEmpty(stack):
    return len(stack) == 0

def peek(stack):
    return stack[-1]

def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1]<=priority[c2]
    except KeyError:
        return False
```

```
def toPostfix(infix):
    stack = []
    postfix = ''
    for c in infix:
        if isOperand(c):
            postfix += c
        else:
            if isLeftParanthesis(c):
                stack.append(c)
            elif isRightParanthesis(c):
                operator = stack.pop()
                while not isLeftParanthesis(operator):
                    postfix += operator
                    operator = stack.pop()
            else:
                while (not isEmpty(stack)) and hasLessOrEqualPriority(c,
peek(stack)):
                    postfix += stack.pop()
                stack.append(c)
    while (not isEmpty(stack)):
        postfix += stack.pop()

    return postfix
def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i,val2,val1))
    return stack.pop()
def _eval(i, val1, val2):
    if i == '^':
        return val2 and val1
    return val2 or val1
#Test 1
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
#Test 2
```

```
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

- **output**



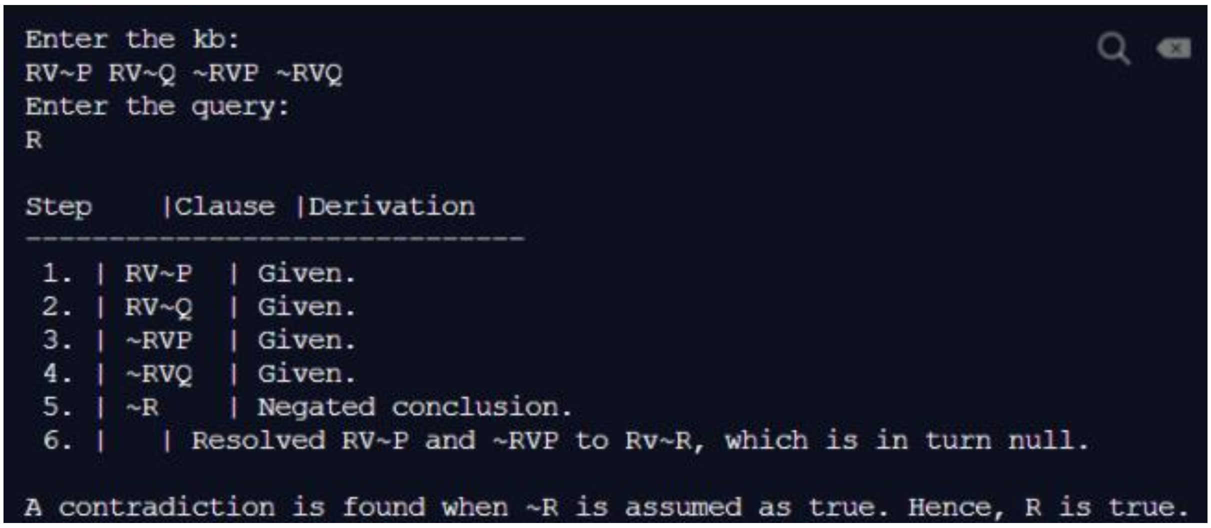
```
Enter rule: (~qv~pvr)^(~q^p)^q
Enter the Query: r
**********Truth Table Reference**********
kb alpha
**********
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
The Knowledge Base entails query
Enter rule: (pvq)^(~rvp)
Enter the Query: r
**********Truth Table Reference**********
kb alpha
**********
True True
----------
True False
----------
The Knowledge Base does not entail query
> 
```

7. **Create a knowledge base using prepositional logic and prove the given query using resolution.**

➢ **Program**

```
import re
def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''
def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms
def contradiction(query, clause):
    contradictions = [ f'{query}v{negate(query)}', f'{negate(query)}v{query}']
    return clause in contradictions or reverse(clause) in contradictions
def resolve(kb, query):
    temp = kb.copy()
    temp += [negate(query)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(query)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if contradiction(query,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and {temp[j]}
to {temp[-1]}, which is in turn null. \
                                \nA contradiction is found when
{negate(query)} is assumed as true. Hence, {query} is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
                        if contradiction(query,f'{terms1[0]}v{terms2[0]}'):
                            temp.append(f'{terms1[0]}v{terms2[0]}')
                            steps[''] = f"Resolved {temp[i]} and {temp[j]} to
{temp[-1]}, which is in turn null. \
                            \nA contradiction is found when {negate(query)} is
assumed as true. Hence, {query} is true."
                            return steps
            for clause in clauses:
                if clause not in temp and clause != reverse(clause) and
reverse(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps
def resolution(kb, query):
    kb = kb.split(' ')
    steps = resolve(kb, query)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1
def main():
    print("Enter the kb:")
    kb = input()
    print("Enter the query:")
    query = input()
    resolution(kb,query)
#test 1
#(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
main()
#test 2
#(P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
main()
```

➢ **output**



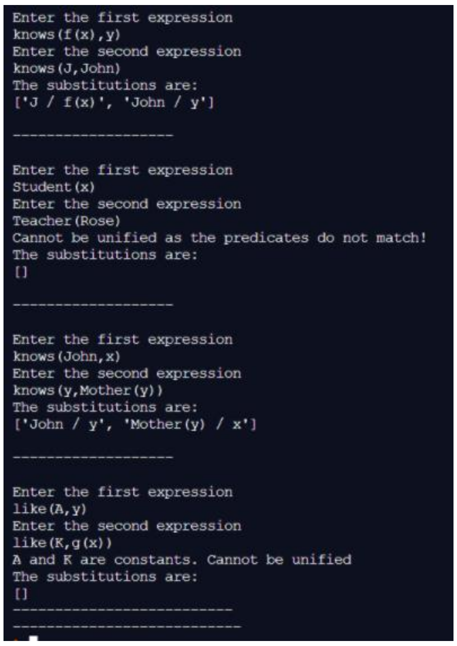
```
Enter the kb:
RV~P RV~Q ~RVP ~RVQ
Enter the query:
R

Step    |Clause |Derivation
------------------------------
 1. | RV~P  | Given.
 2. | RV~Q  | Given.
 3. | ~RVP  | Given.
 4. | ~RVQ  | Given.
 5. | ~R    | Negated conclusion.
 6. |    | Resolved RV~P and ~RVP to Rv~R, which is in turn null.

A contradiction is found when ~R is assumed as true. Hence, R is true.
```

8. **Implement unification in first order logic.**

➢ **Program**

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

def getInitialPredicate(expression):
    return expression.split("(")[0]
def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1
def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp
def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True


def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]


def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2}
do not match. Cannot be unified")
        return []

    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return []
    if attributeCount1 == 1:
        return initialSubstitution

    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)
    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return []

    return initialSubstitution + remainingSubstitution
def main():
    print("Enter the first expression")
    e1 = input()
    print("Enter the second expression")
    e2 = input()
    substitutions = unify(e1, e2)
    print("The substitutions are:")
    print([' / '.join(substitution) for substitution in substitutions])
main()
print(" ")
print("------------------- ")
print(" ")
main()
print(" ")
print("------------------- ")
print(" ")
main()
print(" ")
print("------------------- ")
print(" ")
main()
print("-------------------------- ")
print("---------------------------")
```

- **output**



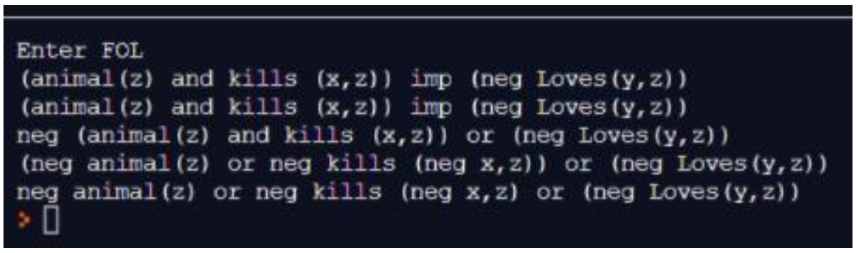
```
Enter the first expression
knows(f(x),y)
Enter the second expression
knows(J,John)
The substitutions are:
['J / f(x)', 'John / y']

-------------------

Enter the first expression
Student(x)
Enter the second expression
Teacher(Rose)
Cannot be unified as the predicates do not match!
The substitutions are:
[]

-------------------

Enter the first expression
knows(John,x)
Enter the second expression
knows(y,Mother(y))
The substitutions are:
['John / y', 'Mother(y) / x']

-------------------

Enter the first expression
like(A,y)
Enter the second expression
like(K,g(x))
A and K are constants. Cannot be unified
The substitutions are:
[]
--------------------------
---------------------------
```

9. Convert given first order logic statement into conjunctive normal form (CNF).

- **Program**

```
import re

print("Enter FOL")
def remove_brackets(source, id):
    reg = '\(([^\(]*?)\)'
    m = re.search(reg, source)
    if m is None:
        return None, None
    new_source = re.sub(reg, str(id), source, count=1)
    return new_source, m.group(1)


class logic_base:
    def __init__(self, input):
        self.my_stack = []
        self.source = input
        final = input
        while 1:
            input, tmp = remove_brackets(input, len(self.my_stack))
            if input is None:
                break
            final = input
            self.my_stack.append(tmp)
        self.my_stack.append(final)

    def get_result(self):
        root = self.my_stack[-1]
        m = re.match('\s*([0-9]+)\s*$', root)
        if m is not None:
            root = self.my_stack[int(m.group(1))]
        reg = '(\d+)'
        while 1:
            m = re.search(reg, root)
            if m is None:
                break
            new = '(' + self.my_stack[int(m.group(1))] + ')'
            root = re.sub(reg, new, root, count=1)
        return root

    def merge_items(self, logic):
        reg0 = '(\d+)'
        reg1 = 'neg\s+(\d+)'
        flag = False
        for i in range(len(self.my_stack)):
            target = self.my_stack[i]
            if logic not in target:
                continue
            m = re.search(reg1, target)
            if m is not None:
                continue
            m = re.search(reg0, target)
            if m is None:
                continue
            for j in re.findall(reg0, target):
                child = self.my_stack[int(j)]
                if logic not in child:
                    continue
                new_reg = "(^|\s)" + j + "(\s|$)"
                self.my_stack[i] = re.sub(new_reg, ' ' + child + ' ',
self.my_stack[i], count=1)
                self.my_stack[i] = self.my_stack[i].strip()
                flag = True
        if flag:
            self.merge_items(logic)


class ordering(logic_base):
    def run(self):
        flag = False
        for i in range(len(self.my_stack)):
            new_source = self.add_brackets(self.my_stack[i])
            if self.my_stack[i] != new_source:
                self.my_stack[i] = new_source
                flag = True
        return flag

    def add_brackets(self, source):
        reg = "\s+(and|or|imp|iff)\s+"
        if len(re.findall(reg, source)) < 2:
            return source
        reg_and = "(neg\s+)?\S+\s+and\s+(neg\s+)?\S+"
        m = re.search(reg_and, source)
        if m is not None:
            return re.sub(reg_and, "(" + m.group(0) + ")", source, count=1)
        reg_or = "(neg\s+)?\S+\s+or\s+(neg\s+)?\S+"
        m = re.search(reg_or, source)
        if m is not None:
            return re.sub(reg_or, "(" + m.group(0) + ")", source, count=1)
        reg_imp = "(neg\s+)?\S+\s+imp\s+(neg\s+)?\S+"
        m = re.search(reg_imp, source)
        if m is not None:
            return re.sub(reg_imp, "(" + m.group(0) + ")", source, count=1)
        reg_iff = "(neg\s+)?\S+\s+iff\s+(neg\s+)?\S+"
        m = re.search(reg_iff, source)
        if m is not None:
            return re.sub(reg_iff, "(" + m.group(0) + ")", source, count=1)


class replace_iff(logic_base):
    def run(self):
        final = len(self.my_stack) - 1
        flag = self.replace_all_iff()
        self.my_stack.append(self.my_stack[final])
        return flag

    def replace_all_iff(self):
        flag = False
        for i in range(len(self.my_stack)):
            ans = self.replace_iff_inner(self.my_stack[i], len(self.my_stack))
            if ans is None:
                continue
            self.my_stack[i] = ans[0]
            self.my_stack.append(ans[1])
            self.my_stack.append(ans[2])
            flag = True
        return flag

    def replace_iff_inner(self, source, id):
        reg = '^(.*?)\s+iff\s+(.*?)$'
        m = re.search(reg, source)
        if m is None:
            return None
        a, b = m.group(1), m.group(2)
        return (str(id) + ' and ' + str(id + 1), a + ' imp ' + b, b + ' imp '
+ a)


class replace_imp(logic_base):
    def run(self):
        flag = False
        for i in range(len(self.my_stack)):
            ans = self.replace_imp_inner(self.my_stack[i])
            if ans is None:
                continue
            self.my_stack[i] = ans
            flag = True
        return flag

    def replace_imp_inner(self, source):
        reg = '^(.*?)\s+imp\s+(.*?)$'
        m = re.search(reg, source)
        if m is None:
            return None
        a, b = m.group(1), m.group(2)
        if 'neg ' in a:
            return a.replace('neg ', '') + ' or ' + b
        return 'neg ' + a + ' or ' + b


class de_morgan(logic_base):
    def run(self):
        reg = 'neg\s+(\d+)'
        flag = False
        final = len(self.my_stack) - 1
        for i in range(len(self.my_stack)):
            target = self.my_stack[i]
            m = re.search(reg, target)
            if m is None:
                continue
            flag = True
            child = self.my_stack[int(m.group(1))]
            self.my_stack[i] = re.sub(reg, str(len(self.my_stack)), target,
count=1)
            self.my_stack.append(self.doing_de_morgan(child))
            break
        self.my_stack.append(self.my_stack[final])
        return flag

    def doing_de_morgan(self, source):
        items = re.split('\s+', source)
        new_items = []
        for item in items:
            if item == 'or':
                new_items.append('and')
            elif item == 'and':
                new_items.append('or')
            elif item == 'neg':
                new_items.append('neg')
            elif len(item.strip()) > 0:
                new_items.append('neg')
                new_items.append(item)
        for i in range(len(new_items) - 1):
            if new_items[i] == 'neg':
                if new_items[i + 1] == 'neg':
                    new_items[i] = ''
                    new_items[i + 1] = ''
        return ' '.join([i for i in new_items if len(i) > 0])


class distributive(logic_base):
    def run(self):
        flag = False
        reg = '(\d+)'
        final = len(self.my_stack) - 1
        for i in range(len(self.my_stack)):
            target = self.my_stack[i]
            if 'or' not in self.my_stack[i]:
                continue
            m = re.search(reg, target)
            if m is None:
                continue
            for j in re.findall(reg, target):
                child = self.my_stack[int(j)]
                if 'and' not in child:
                    continue
                new_reg = "(^|\s)" + j + "(\s|$)"
                items = re.split('\s+and\s+', child)
                tmp_list = [str(j) for j in range(len(self.my_stack),
len(self.my_stack) + len(items))]
                for item in items:
                    self.my_stack.append(re.sub(new_reg, ' ' + item + ' ',
target).strip())
                self.my_stack[i] = ' and '.join(tmp_list)
                flag = True
            if flag:
                break
        self.my_stack.append(self.my_stack[final])
        return flag


class simplification(logic_base):
    def run(self):
        old = self.get_result()
        for i in range(len(self.my_stack)):
            self.my_stack[i] = self.reducing_or(self.my_stack[i])
        # self.my_stack[i] = self.reducing_and(self.my_stack[i])
        final = self.my_stack[-1]
        self.my_stack[-1] = self.reducing_and(final)
        return len(old) != len(self.get_result())

    def reducing_and(self, target):
        if 'and' not in target:
            return target
        items = set(re.split('\s+and\s+', target))
        for item in list(items):
            if ('neg ' + item) in items:
                return ''
            if re.match('\d+$', item) is None:
                continue
            value = self.my_stack[int(item)]
            if self.my_stack.count(value) > 1:
                value = ''
                self.my_stack[int(item)] = ''
            if value == '':
                items.remove(item)
        return ' and '.join(list(items))

    def reducing_or(self, target):
        if 'or' not in target:
            return target
        items = set(re.split('\s+or\s+', target))
        for item in list(items):
            if ('neg ' + item) in items:
                return ''
        return ' or '.join(list(items))


def merging(source):
    old = source.get_result()
    source.merge_items('or')
    source.merge_items('and')
    return old != source.get_result()


def run(input):
    all_strings = []
    # all_strings.append(input)
    zero = ordering(input)
    while zero.run():
        zero = ordering(zero.get_result())
    merging(zero)

    one = replace_iff(zero.get_result())
    one.run()
    all_strings.append(one.get_result())
    merging(one)

    two = replace_imp(one.get_result())
    two.run()
    all_strings.append(two.get_result())
    merging(two)

    three, four = None, None
    old = two.get_result()
    three = de_morgan(old)
    while three.run():
        pass
    all_strings.append(three.get_result())
    merging(three)
    three_helf = simplification(three.get_result())
    three_helf.run()

    four = distributive(three_helf.get_result())
    while four.run():
        pass
    merging(four)
    five = simplification(four.get_result())
    five.run()
    all_strings.append(five.get_result())
    return all_strings



inputs = input().split('\n')
for input in inputs:
    for item in run(input):
        print(item)
    # output.write('\n')
```

➢ **output**



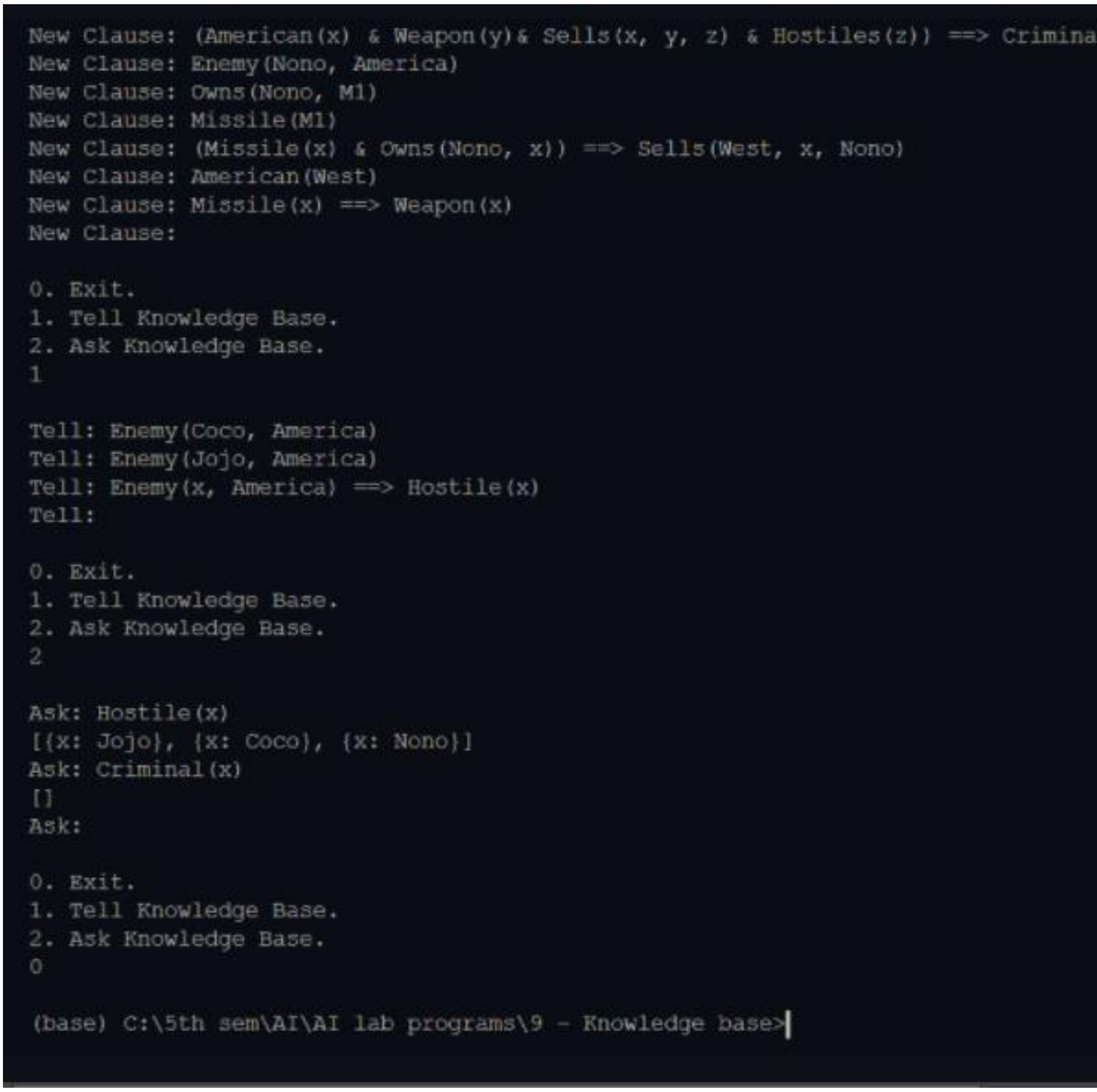
```
Enter FOL
(animal(z) and kills (x,z)) imp (neg Loves(y,z))
(animal(z) and kills (x,z)) imp (neg Loves(y,z))
neg (animal(z) and kills (x,z)) or (neg Loves(y,z))
(neg animal(z) or neg kills (neg x,z)) or (neg Loves(y,z))
neg animal(z) or neg kills (neg x,z) or (neg Loves(y,z))
> 
```

10. **Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.**

- **Program**

```python
import aima.logic
import aima.utils


def get_clauses():
    clauses = list()

    clause = input("New Clause: ")
    # Add first-order logic clauses (rules and fact)
    while clause != "":
        clauses.append(aima.utils.expr(clause))
        clause = input("New Clause: ")

    print()
    return clauses


def tell_kb(kb):
    tell = input("\nTell: ")
    # Add rules and facts with tell
    while tell != "":
        kb.tell(aima.utils.expr(tell))
        tell = input("Tell: ")
    print()


def ask_kb(kb):
    ask = input("\nAsk: ")
    # Ask Knowledge Base
    while ask != "":
        print(list(aima.logic.fol_fc_ask(kb, aima.utils.expr(ask))))
        ask = input("Ask: ")
    print()


def main():
    # Create an array to hold clauses
    clauses = get_clauses()

    # Create a first-order logic knowledge base (KB) with clauses
    kb = aima.logic.FolKB(clauses)

    menu = "0. Exit.\n1. Tell Knowledge Base.\n2. Ask Knowledge Base.\n"
    entry = input(menu)
    while entry != '0':
        if entry == '1':
            tell_kb(kb)
        elif entry == '2':
            ask_kb(kb)
        else:
            print()
        entry = input(menu)


# Tell python to run main method
if __name__ == "__main__":
    # Clause
    # (American(x) & Weapon(y) & Sells(x, y, z) & Hostile(z)) ==> Criminal(x)
    # Enemy(Nono, America)
    # Owns(Nono, M1)
    # Missile(M1)
    # (Missile(x) & Owns(Nono, x)) ==> Sells(West, x, Nono)
    # American(West)
    # Missile(x) ==> Weapon(x)

    # Tell
    # Enemy(Coco, America)
    # Enemy(Jojo, America)
    # Enemy(x, America) ==> Hostile(x)

    # Ask
    # Hostile(x)
    # Criminal(x)
    main()
```

➢ **output**



```
New Clause: (American(x) & Weapon(y)& Sells(x, y, z) & Hostiles(z)) ==> Crimina
New Clause: Enemy(Nono, America)
New Clause: Owns(Nono, M1)
New Clause: Missile(M1)
New Clause: (Missile(x) & Owns(Nono, x)) ==> Sells(West, x, Nono)
New Clause: American(West)
New Clause: Missile(x) ==> Weapon(x)
New Clause:

0. Exit.
1. Tell Knowledge Base.
2. Ask Knowledge Base.
1

Tell: Enemy(Coco, America)
Tell: Enemy(Jojo, America)
Tell: Enemy(x, America) ==> Hostile(x)
Tell:

0. Exit.
1. Tell Knowledge Base.
2. Ask Knowledge Base.
2

Ask: Hostile(x)
[{x: Jojo}, {x: Coco}, {x: Nono}]
Ask: Criminal(x)
[]
Ask:

0. Exit.
1. Tell Knowledge Base.
2. Ask Knowledge Base.
0

(base) C:\5th sem\AI\AI lab programs\9 - Knowledge base>
```