INDEX

Name Madhasoodan Reddy Sub. AI-Lab

Std.: V Div. Sem Roll No. 1BM21CS099

Telephone No. E-mail ID.

Blood Group. Birth Day.

	Sr.No.	Title	Page No.	Sign./Remarks
3/11/23	1	Python Introduction		
10/11/23	2	Python programs		
17/11/23	3	Tic Tac Game		10 17/11
24/11/23	4	8-Puzzle Game, BFS		10 24-11-23
08/11/23	5	8-Puzzle using A* algorithm		
	6	8-puzzle using IDDFS		30 Lab 22/12/23
22/12/23	7	Vaccumer cleaner.		
29/12/23	8	Knowledge Base		10 Lab 29/12/23
19/01/24				
29/01/24	1)	Implement using FOL		
	2)	FOL to PNF conversion		10 Lab 24/1/24
	3)	create KB consists of FOL & prove the given query using Forward Reasoning.		

3/11/23

Lab-1

PAGE NO :
DATE :

Python :-

Topics Learnt :

1) Introduction
2) Functions and Getting Help.
3) Booleans and conditionals
4) Lists
5) Loops and List comprehensions
6) Strings and Dictionaries
7) Working with External Libraries.

10/11/23

Lab-2:

PAGE NO:
DATE:

Write Python Programs:

Prog

1) Program for age categorisation:

```
def age_criteria(age):
    if (age<=2):
        print("Infants")
    elif (age<=12):
        print("Kids")
    elif (age<=20)
        print("Teens")
    elif (age<=59):
        print("Adults")
    elif
    else:
        print("Senior citizens")

age=int(input("Enter the age \n"))
age_criteria(age)
```

Output:-

Enter the age
11
Kids

10/11/23

Lab-2

Write Python Programs

Prog

1) Program for age categorisation

```
def age_criteria(age):
    if (age<=2):
        print("Infants")
    elif (age<=12):
        print("Kids")
    elif (age<=20):
        print("Teens")
    elif (age<=59):
        print("Adults")
    else:
        print("Senior citizens")

age=int(input("Enter the age \n"))
age_criteria(age)
```

Output:-

```
Enter the age
11
Kids
```

2) Program for multiplication tables:-

```
def multi-table(num,k):
    for i in range(k):
        ans=num*(i+1)
        print(f'{num}*{i+1}={ans}\n')

table=int(input("Enter the number\n"))
k=int(input("till which number\n"))
multi-table(table,k)
```

Output:-

```
Enter the number        Enter till which number
5                       10
5x1=5
5x2=10
5x3=15
5x4=20
5x5=25
5x6=30
5x7=35
5x8=40
5x9=45
5x10=50
```

3) Pattern

```
1
2 2
3 3 3
```

```
def pattern1(num):
    for i in range(1,num+1):
        for j in range(i):
            print(f"{i}", end="")
        for k in range(num-i+1):
            print(" ", end="")
        print("\n")
num=int(input("Enter the Number\n"))
pattern1(num)
```

Output:

```
3
1
2 2
3 3 3
```

4) for pattern:

```
1
1 2
1 2 3
```

```
def pattern2(num):
    for i in range(1,num+1):
        for j in range(i):
            print(f"{i+1}", end="")
        for k in range(num-i+1):
            print(" ", end="")
        print('\n')
num=int(input("Enter number\n"))
pattern2(num)
```

Output:

```
1
1 2
1 2 3
```

4) Reverse of a number:

```
num=int(input("Enter the number\n"))
reversed_num=0
while num!=0:
    digit=num%10
    reversed_sum=reversed_sum*10+digit
    num//10
print("Reversed Number:"+str(reversed_num))
```

Output:

Number: 4223

Reversed number: 3224

5) Program for bubble sort:

```
def bubble_sort(list_item):
    for i in range(0,len(list_item)):
        for j in range(i+1, len(list_item)):
            if(list_item[i]>list_item[j]):
                temp=list_item[i]
                list_item[i]=list_item[j]
                list_item[j]=temp
    return list_item
n=int(input("Enter the size"))
list_item=[]
for j in range(n):
    list_item.append(int(input()))
list_item=bubble_sort(list_item)
print(list_item)
```

Output:

```
Enter the size: 3
Enter the elements= 3 8 1
[1,3,8].
```

Lab-3:

```
list=[][]
for i in range(3):
    for j in range(3):
        list[i][j]= i+j+1

def create_board(list):
    for i in range(3):
        for j in range(3):
            print(list[i][j])
            print("|")
    print("_____\n").

def check_winner(list):
    for i in range(3):
        if (list[i][0]==list[i][1] && list[i][1]
            ==list[i][2]):
            return True
    if(list[0][i]==list[1][j] && list[1][i]==
        list[2][i]):
        return True
    else:
        for j in range(3):
            y=list[0][2]
            x=list[0][0]
        count=0, count1=0
        if(i==j && list[i][j]==x):
            count++
        if(i+j==2 && list[i][j]==y):
            count1++
```

```
    if (count == 3 || count2 == 3)
        return True.

def play (k,i)
    n=k/3
    if (list[n][k%3] = 'x' || list[n][k%3] = 'o')
        print ("player", i, "has already played this position")
    else:
        if (i==1):
            list[n][k%3]='x'
        else:
            list[n][k%3]='o'

count=0
while(1):
    create_board(list)
    print("player", count%2+1, "to play")
    x=int(input())
    play(x, count%2+1)

    if(checks-winner):
        print("player", count%2+1, "won"),
        break,
    count++

    if(count==9):
        print ("It's a draw")
        break;
```

Algorithm :-

1) Fill the List with dummy numbers.
2) Write create_board function → helps in printing board.
3) create a function (check_winner) which runs after a every player plays and checks for the winning condition.
4) create a function play function which helps players to switch their chances to play.
5) Main function takes input from the players and calls the above functions.

Output :-

24/11/22

Lab 4

PAGE NO :
DATE :

Algorithm :

1) create a Puzzles class and define initial configuration & goal state & shuffle the numbers to generate a random initial board.
2) Print the current Board & implement a method for it to print.
3) For moving of Tiles → Implement a method to tiles based on (BFS) & check for validity of the moves. to avoid
4) A method to check if the current board configuration matches the goal state.
5) Game Loop:
   Create a game loop that continues till puzzle is solved.
   In each iteration of the loop:
   print the current state of board.
   Prompt the (computer) for a move direction.
   Move the tiles accordingly.
   Check if the puzzle is solved.
   If solved, break out of the loop.
   If not solved, repeat the loop.

29/11

PAGE NO :
DATE :

code :

```
import copy
from heapq import heappush, heappop.

n = 3.
row = [1, 0, -1, 0]
col = [0, -1, 0, 1]

class PriorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, k)
        heappush(self.heap, k)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        return not bool(self.heap)

class Node:
    def __init__(self, parent, mat, empty_tile_pos,
                 level):
        self.parent = parent
        self.mat = mat
        self.empty_tile_pos = empty_tile_pos
        self.cost = cost
        self.level = level
    def __lt__(self, nxt):
        return self.cost < nxt.cost
```

```
def calculatecost(mat, final) -> int:
    count = 0;
    for i in range(n):
        for j in range(n):
            if mat[i][j] and (mat[i][j] != final[i][j]):
                count += 1;
    return count.

def newNode(mat, empty_tile_pos, new_empty_tile_pos, level, parent, final) -> Node:

    new_mat = copy.deepcopy(mat)
    x1, y1 = empty_tile_pos
    x2, y2 = new_empty_tile_pos
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
    cost = calculatecost(new_mat, final)
return new_node.
```

solve1)

Output:

initial

```
1 2 3
8 0 4
7 6 5
```

final

```
2 8 1
0 4 3
2 6 5
```

Output:

```
1 2 3
0 4 6
7 5 8.
```

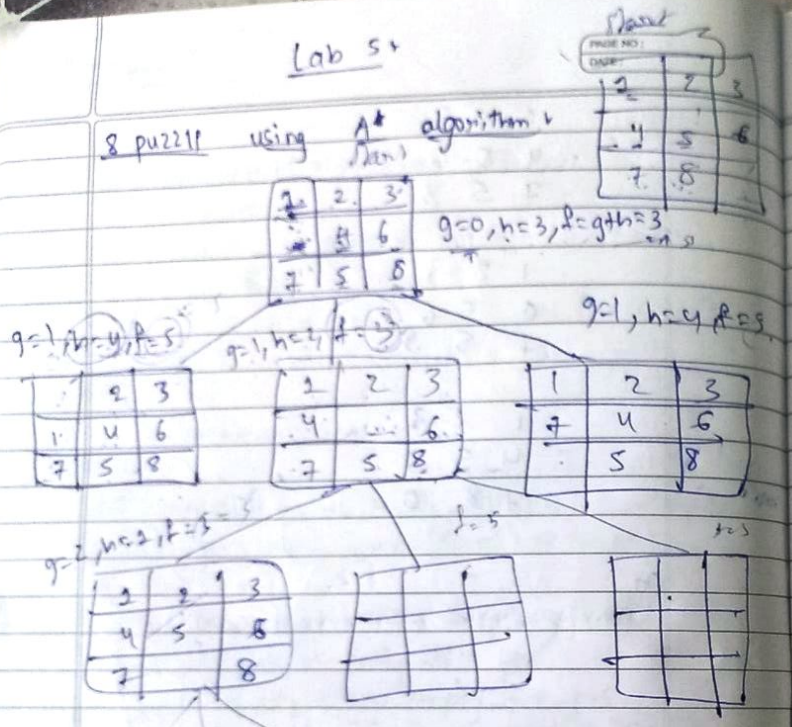
```
1 2 3
4 0 6
7 5 8.

1 2 3
0 5 6
7 5 8

1 2 3
4 5 6
7 8 0.
```

Lab 5:

8 puzzle using A* algorithm:



$f = g + h \rightarrow$

```
Class Node:
    def __init__(self.data, level, fval):
        self.data = data
        self.
```

Algorithm:

1) Class Node:

```
    def __init__(self.data, level, fval):
        " Initialize the node with data, level & missing values "

    def generate_child(self)
    """ generate child nodes from the given node by moving the blank space """

    x, y = self.find(self.data, '_')

    val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]

    children = []
    for i in val_list:
        child = self.shuffle(self.data, x, y, i[0], i[1])
    if child is not None:
        child_node = Node(child, self, level+1, 0)
        children.append(child_node)
    return children.
```

```
def shuffle(self,pu2,x1,y1,x2,y2)
if  x1>=0 & x2 <len(self.data) & y2>=0 &
    y2 <len(self.data):
      temp_pu2=[]
      temp_pu2=self.copy(pu2)

      return temp_pu2.
  else:
      return None.

def find(self,pu2,x):
  "returns index of blank space"


class Puzzle:
  Initiate
  def __init__(self,size):
      self.n=size
      self.open=[]
      self.closed=[]

def accept(self):
  "Accepts puzzle from user";

def f(self,start,goal):
    return self.h(start.data,goal)+start.level,
```

```
def h(self,start,goal):
  "calculate the ~~difference~~ no: of missing values"
  temp=0
  for i in range(0,self.n):
      for j in range(0,self.n):
        if start[i][j]!=goal[i][j]and
            start[i][j]!='_':
              temp+=1
  return temp.

def process(self):

start=self.accept()
  goal=self.accept()

  start=Node(start,0,0)
  start.fval=self.f(start,goal)
  self.open.append(start).
while True:
    cur=self.open[0]


  for i in cur.data:
      for j in i:
        print(j,end=" ")
      print("\n")

  if(self.h(cur.data,goal)==0):
      break.

  for i in cur.generate_child():
      i.fval=self.f(i,goal)
      self.open.append(i)
  self.closed.append(cur)
  del self.open[0]
```

```
stff, open.sort(key=lambda x:x[1],val,
    reverse=False)

pu2=puzzle(3)
pu2.process().
```

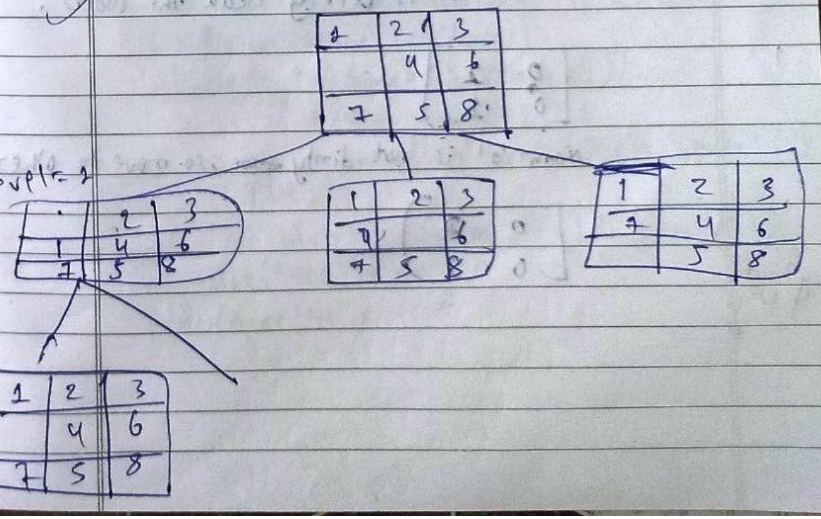
Output

IDDFS :-

```
def id_dfs(puzzle, goal, get_moves):
    import itertools

    def dfs(route, depth):
        if depth==0:
            return
        if route[-1]==goal:
            return route
        for move in get_moves[route[-1]]:
            if move not in route:
                next_route=dfs[route+[move],
                                depth-1)
                if next_route:
                    return next_route:

    for depth in itertools.count():
        route=dfs(puzzle, depth)
        if route:
            return node
```



Lab 6:

Write program to simulate vaccum cleaner for 2 rooms.

Steps:- Create room array.

Step 2:- In each cell, if room is dirty mark it as '0' and if room is 'clean' mark it as '1'.

Step 3:- Create a clean function through which the cleaner moves right, left, up down & checks the room is dirty or not.

Step 4:- After cleaning the room travel to next room & clean to cleaning.

Eg:- [0 1]
↑
Room '1' is not dirty, so move to other room
[0 1]
↑
Room '1' is dirty clean the room.

$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$

Room '1' is not dirty, so move to other

$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$

Passed

Code:-

```
def clean(floor):
    i,j, row, col = 0,0, len(floor), len(floor[0])
    for i in range(row):
        if(i%2==0):
            for j in range(col):
                if(floor[i][j]==1):
                    print_f(floor, i, j)
                    floor[i][j]=0
                print
            print_f(floor,i,j)
        else:
            for j in range(col-1,-1,-1):
                if(floor[i][j]==1):
                    print_f(floor,i,j)
                    floor[i][j]=0
            print_f(floor,i,j)

def print_F(floor,row,col):
    print("The floor matrix is as below:")
    for r in range(len(floor)):
        for c in range(len(floor[r])):
            if r==row and c==col:
                print(f"->{floor[r][c]}<",end='')
            else:
                printf(f" {floor[r][c]} ",end='')
        print(end='\n')
    print(end='\n')
```

```
def main()
    floor = []
    m = input
    n = input
print("Enter the clean status ("0" for "dirty" or "clean")")
for i in range(m):
    f = list(map(int, input.split(" ")))
    floor.append(f)
print()
clean(floor)

main()
```

Output:

Enter no. of rows & columns: 2 2

Enter clean status.

0 1

1 0

If 0 < 1

1 0

0 > 1 <

1 0

Room 0 is dirty.

0 > 0 <

1 0.

Date: 22/12/23

29/12/23

Knowledge Lab 8:-

1) Knowledge base entailment:

i) If it is raining (P), then ground is wet (Q)

ii) If the ground is wet (Q), then the plants will grow (R).

iii) It's not the case that plants will grow (~R).

Query - Whether it is raining.

Code:

```
from sympy import symbols, And, Not, Implies,
satisfiable

def create_knowledge_base():
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')
    knowledge_base = And(
        Implies(p, q),
        Implies(q, r),
        Not(r)
    )
    return knowledge_base
```

```
def query_entails(knowledge_base, query):
    entailment = satisfiable(And(knowledge_base,
                                 Not(query)))
    return not entailment.

if __name__=="__main__":
    kb = create_knowledge_base()
    query = symbol('p')
    result = query_entails(kb, query)
    print("Knowledge Base:", kb)
    print("Query:", query)
    print("Query entails Knowledge Base:", result)
```

Output :-

Query : p

query entails Knowledge Base: False.

(2) Knowledge Base Resolution :-

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()

    def add_fact(self, fact):
        self.facts.add(fact)

    def check_entailment(self, statement):
        return all(rule(statement) for rule in
                   self.facts)

def rule_0(statement):
    return "p" in statement.lower()

def rule_1(statement):
    return "np" in statement.lower() or "q"
           in statement.lower()

def rule_2(statement):
    return "p" in statement.lower() or "nq"
    in statement.lower() or "r" in statement.
        lower()

def rule_3(statement):
    return "nq" in statement.lower()
        or "r" in statement.lower()
```

```
kb = KnowledgeBase()
kb.add_fact(rule_0)

user_statement = input("Enter a statement: ")
entail_result = kb.check_entailment(user_statement)

if entail_result:
    print("The statement is entailed by KB")
else:
    print("The statement is not entailed by KB")
```

Output:

Enter the statement: P

The statement is not entailed by the KB.

29/12/23

Lab 9:

Implement unification in FOL

```
import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-2]
    expression = re.split("?<!\(.), (?!.\))", expression)
    return expression

def getInitialPredicate(expression):
    return expression.split("(")[0]

def isVariable(char):
    return char.islower() and len(char)==1

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp
```

```
def unify(exp1, exp2):
    if exp1 == exp2:
        return []
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False
    if isConstant(exp2):
        return [(exp2, exp2)]
    if isConstant(exp2):
        return [(exp2, exp2)]
    if isVariable(exp1):
        if checkOccurs(exp2, exp2):
            return False
        else:
            return [(exp2, exp1)]
    if isVariable(exp2):
        if checkOccurs(exp2, exp1):
            return False:
        else
            return [(exp1, exp2)]
if getInitialPredicate(exp1) != getInitialPredicate(exp2):
    print("Predicates do not match. Cannot be unified")
    return False.
```

```
attributeCount1 = len(getAttributes(exp1))
attributeCount2 = len(getAttributes(exp2))
if attributeCount1 != attributeCount2:
    return False

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

tail1 = getRemainingPart(exp1)
tail2 = get
if initialSubstitution != []:
    tail1 = apply(tail1, initialSubstitution)
    tail2 = apply(tail2, initialSubstitution)

remainingSubstitution = unify(tail1, tail2)
if not remainingSubstitution:
    return False

initialSubstitution.extend(remainingSubstitution)
return initialSubstitution
```

```
expr2 = "knows(A,x)"
expr2 = "knows(y,mother(y))"
substitutions = unify(expr2, expr2)
print('substitutions')
```

output:

Substitutions:
[('A','y'),('mother(y)','x')]

```
expr2 = "knows(A,x)"
expr2 = "knows(y,B)"
substitutions = unify(expr2, expr2)
print("substitutions")
```

output:

substitutions:
[('A','x'),('B','y')]

Prog. 2 Program.

```
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '[a-zA-Z]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = string.replace('~', '')
    flag = '[' in string
    string = string.replace('[', '')
    string = string.strip(']')
    for predic in getPredicates(string):
        s = list(string)
        s[s.index(predic)] = f'~{predic}'
        if flag:
            s.insert(0, '[')
            s.insert(-1, ']')
        string = ''.join(s)
    return string

def skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('\[[^\]]+\]', statement)
    for s in matches:
        statement = statement.replace(s, s[1:-1])
    for predicate in getPredicates(statement):
        attributes = getAttributes(predicate)
```

```
        if '' .join(attributes).islower():
            statement = statement.replace(match[2],
                        SKOLEM_CONSTANTS.pop(0))
        else:
            aL = [a for a in attributes if a.islower()]
            aU = [a for a in attributes if not a.islower()][0]
            statement = statement.replace(match[2], SKOLEM
    return statement

import re

def fol_to_cnf(fol):
    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i]
        statement = new_statement
    statement = statement.replace("=>", "_")
    expr = '\[([^\]]+)\]'
    statements = re.findall(expr, statement)
    for i,s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '~' in statement:
        i = statement.index('~')
        br = statement.index('[') if '[' in
             statement else 0
        new_statement = '~' + statement[br:i] + '|'
                        + statement[i+2:]
    statement = statement[:br] + new_statement
                if br>0 else new_statement.

    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2]
            = '∃', statement[i+2], '~'
        statement = ''.join(statement)

    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
        statement = ''.join(s)

    return statement

print(skolemization(fol_to_cnf('animal(y) <=> loves(x,y)')))

output:
    [~animal(y) | loves(x,y)] & [~loves(x,y) |
                                 animal(y)].

print(skolemization(fol_to_cnf("
    [american(x) & weapon(y) & sells(x,y,z) & hostile(z)]
        => criminal(x)")))

output:
[~american(x) | ~weapon(y) | ~sells(x,y,z) | ~hostile
                                              (z)]
    | criminal(x).
```

11) Write a program to create a knowledge consisting of first order logic statements & prove the given query using forward reasoning.

```
import re

def isVariable(x):  #checks whether it is variable
    return len(x)==1 & x.islower() and ...(or) not
        x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-zA-Z]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression
                            (expression)
```

```
    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params =
        return split Expression.

    def getResult(self):
        return self.result.

    def getConstants(self): # used to get constants
                            from given
                            expression.

    def getVariables(self):
        return [v if isVariable(v) else None for v in
                self.params].

    def substitute:  # used to substitute.

class Implication:

    def evaluate(self, facts): #this checks if the
        constants = {}          antecedents are
        new_lhs = []            satisfied by the
        for fact in facts:      facts in the KB.
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i,v in enumerate(val.getVariables(
                        if v:
                            constants[v] = fact.getConstants()[
                    new_lhs.append(fact).
```

```
class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))

    def query(self, e):
        for f in self.facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')

kb = KB()
kb.tell('missile(x)=>weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x,America)=>hostile(x)')
kb.tell('american(West)')
kb.tell('owns
```

```
kb = KB()
kb.tell('king(x)&greedy(x)=>evil(x)')
kb.tell('king(John)')
kb.tell('greedy(John)')
kb.tell('king(Richard)')
kb.query('evil(x)')
```

Output :-

Querying evil(x) :-
1. evil(Richard)
2. evil(John)

Safe
24/1/24