7/11/24

1) Tic Tac Toe

Algorithm:-

1. In the main function initialize a 'board' → nested list with space & Initialize a variable player = 'X'
   When player = "X" → user input
   " " = "O" → computer
2. Make a loop for to run 9 times which is no ^ of turns.
3. Ask user input for row & col and enter 'X' in the specified space if not taken already
   If taken print "Spot is taken") & prompt the user again.
4. When player = 'O' call another function "computer (board)" This function must generate a random number [0-2] for row & another random number [0-2) for column
   → If taken generate again.
5. After each turn call function check winner (board)
   ↓
   This function specifies the winner when pattern is achieved
6. We have another function to print board

Code:-

```
import random

def printboard(board):
    for row in board:
        print(" | ".join(row))
        print("-"*9)

def checkwinner(board):
    for row in board:
        if row.count(row[0]) == len(row) and row[0] != ' ':
            return True

    for col in range(len(board[0])):
        if board[0][col] == board[1][col] == board[2][col] != ' ':
            return True

    if board[0][0] == board[1][1] == board[2][2] != ' ' or
```

```
return False

def computermove(board):
    while True:
        row = random.randint(0,2)
        col = random.randint(0,2)
        if board[row][col] == ' ':
            return row, col

board = [[' ' for _ in range(3)] for _ in range(3)]

printboard(board)

while True:
    row = int(input("Enter row(0,1,2)"))
    col = int(input("Enter col(0,1,2)"))
    if board[row][col] == ' ':
        board[row][col] = 'X'
        printboard(board)

    if checkwinner(board):
        print("Congratulations!")
        break

comp_row, comp_col = computer_move(board)
board[comp_row][comp_col] = 'O'
print("Computer's move!")
printboard(print)

if checkwinner(board):
    print("Computer won! Better luck next time")
    break
else:
    print("That spot is already taken!")

if all(board[i][j] != ' ' for i in range(3) for j in range(3)):
    print("It's a tie!")
    break
```

7/11

```
Harshala -1BM21CS074
Would you like to go first or second? (1/2):
1
   |   |
---+---+---
   |   |
---+---+---
   |   |


Player move: (0-8):
0
 O |   |
---+---+---
   |   |
---+---+---
   |   |


 O |   |
---+---+---
   | X |
---+---+---
   |   |


Player move: (0-8):
3
 O |   |
---+---+---
 O | X |
---+---+---
   |   |
```

Program - 8

8 Puzzle problem using BFS

In the problem, square will have N+1 tiles, where N = 8, 15, etc

If N = 8, square will have 9 tiles

In the problem, initial state will be given & we have to reach goal state / goal config

↓

Solved by moving the tiles one by one empty space & thus achieve goal state

Rules:-

→ Empty can only move in 4 dirs

Can't move diagonally & only 1 step at a time.

→

o	x	o
x	#	x
o	x	o

Tiles at o — no of possible moves
" at x - " " "
" at # -

Complexity → O(b^d) where

b - branching factor
d - depth factor

Worst case = 3^20

B branching factor = all possible moves by empty tile at each position / no of total tiles

$= \frac{24}{9} \approx 3$

depth factor - initially 0

As we explore nodes depth factor ↑s

1	2	3
	4	6
7	5	8

3 possible moves

right, up, down

(2)
(3)
(4)

1	2	3
	4	6
7	5	8

	2	3
1	4	6
7	5	8

1	2	3
7	4	6
	5	8

similarly continue branching nodes

```
import numpy as np
    Pandas as pd
    os

def bfs(src,target):          (BFS code)
    queue = []
    queue.append(src)
    exp = []

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

    print(source)

    if source == target:
        print("success")
        return
    pos_moves_to_do = []
pos_moves_to_do = possible_moves(source)

for move in pos_moves_to_do:

    if move not in exp and move not in queue:

        queue.append(move)
```

```
def possible_moves(state, visited_states):
    b = state.index(0)

    d = []                          [To print possible direction when the empty node is at diff positions]
    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')

    pos_moves_it_can = []

    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]

def gen(state, m, b):          [To move down shift 3 units front & so on]
    temp = state.copy()

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

```
    return temp
initial_state = [1, 2, 3, 4, 5, 6, 7, 8, -1]
target_state = [1, 2, 3, 4, 5, 6, 7, 8, -1]

bfs(initial_state, target_state)
```

```
1 2 3
4 5 6
7 8

Success
```

Program 3

8 puzzle using A* search

~~Class Node~~

Algorithm:-

1. We move the empty in all possible directions in the start state and calculate f-score for each state.

   This is called expanding current state

2. After expanding the current state, it is pushed into the closed list & the ~~new~~ newly generated states are pushed into the open list

3. A state with the least f-score is selected and expanded again

   f score = h score + g-score

   h → heuristic score

   g → no of nodes traversed from start node to current node

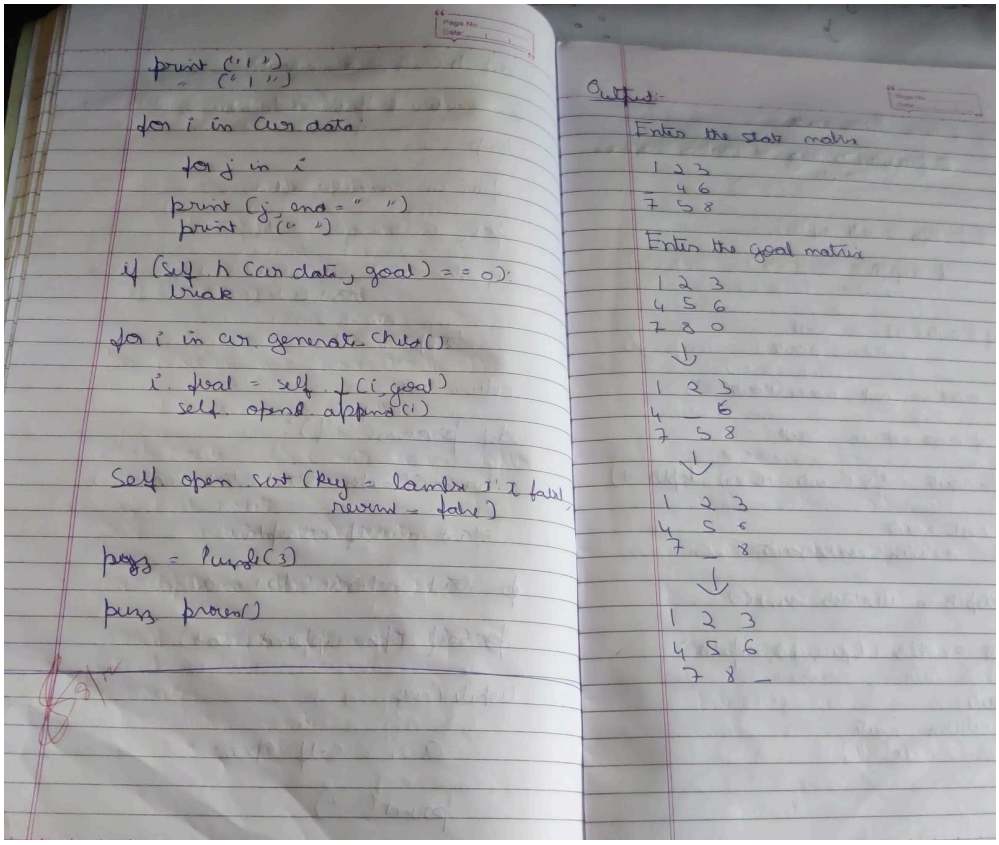
4. ~~Puzzle~~ There are 2 class - Node & Puzzle.

Node class - defines the structure of the state & also provides function to move empty space & generate child state from current state.

Puzzle class accepts initial & goal states of the N-Puzzle problem & provides function to calculate f score of any given node

```
class Node:

    def __init__(self, data, level, fval):

        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):

        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
```

```
            if child is not None:

                child_node = Node(child, self.level+1, 0)
                children.append(child_node)

        return children

    def shuffle(self, puz, x1, y1, x2, y2):

        if x2 >= 0 & x2 < len(self.data) & y2 >= 0
            & y2 < len(self.data):

            temp_puz = []
            temp_puz = self.copy(puz)


            return temp_puz

        else:
            return None

    def copy(self, root):

        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)

        return temp
```

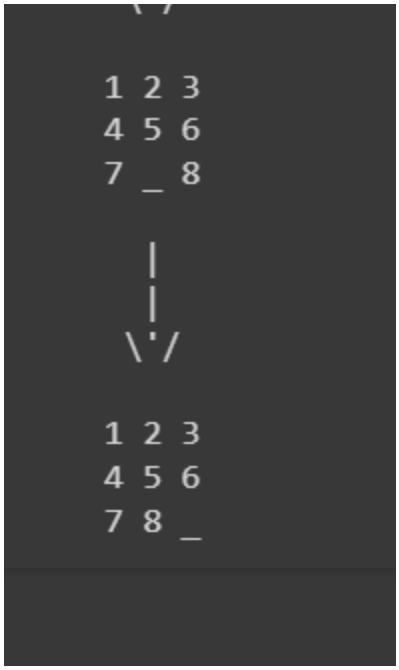


```
print ('|')
      ('|')
for i in cur_data:
    for j in i
    print (j, end = " ")
    print ('|')
if (self.h(cur_data, goal) == 0):
    break
for i in cur.generate_child():
    i.fval = self.f(i, goal)
    self.open.append(i)
self.open.sort(key = lambda x: x.fval,
               reverse = False)
puzz = Puzzle(3)
puzz.process()
```

Output:

```
Enter the start matrix
1 2 3
_ 4 6
7 5 8
Enter the goal matrix
1 2 3
4 5 6
7 8 0
|
1 2 3
4 _ 6
7 5 8
|
1 2 3
4 5 6
7 _ 8
|
1 2 3
4 5 6
7 8 _
```

```
Enter the start state matrix

1 2 3
4 5 6
_ 7 8
Enter the goal state matrix

1 2 3
4 5 6
7 8 _




  |
  |
 \'/

1 2 3
4 5 6
_ 7 8

  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
  |
 \'/
```


\'/
1 2 3
4 5 6
7 _ 8
|
|
\'/
1 2 3
4 5 6
7 8 _

Program 4

8 puzzle using Iterative Deepening search

```
def dfs(route, depth):
    if depth == 0:
        return
    if route[-1] == goal:
        return route
    for mov in get_moves(route[-1]):
        if mov not in route:
            next_route = dfs(route + [mov], depth-1)
            if next_route:
                return next_route
    for depth in itertools.count():
        route = dfs([puzzle], depth)
        if route:
            return route
```

```
def possible_moves(state):
    b = state.index(0)
    d = []
    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')
    pos_moves = []
    for i in d:
        pos_moves.append(generate(state, i, b))
    return pos_moves

def generate(state, m, b):
    temp = state.copy()
    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 temp

initial = [1,2,3,0,7,6,7,5,8]
goal = [1,2,3,4,5,6,7,8,0]

route = id_dfs(initial, goal, pos_moves)
if route:
    print("Success! 9! is possible")
    print("Path:", route)
else:
    print("Fail!")
```

Output:

Success! 9! is possible 8 puzzle program

Path: [[1,2,3,0,4,6,7,5,8],[1,2,3,4,0,6,7,5,8],
[1,2,3,4,5,6,7,0,8],[1,2,3,4,5,6,7,8,0]]

```
Harshala Rani-1bm21cs074
Success!! It is possible to solve 8 Puzzle problem
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]]
```

Programs

Vaccum Cleaner

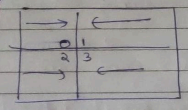
~~def clean():~~

* Algorithm:-

① We have a 2D list called 'floor'

We take input for no of rows & columns of the floor grid.

The user then inputs initial state of each cell in this grid (1 for dirty & 0 for clean)
→ You do this for each room

② 

Total of 4 rooms

0 & 2 room - Cleaner moves L to R

1 & 3 room - Cleaner moves R to L

③ In each room, ~~the~~ cleaner must check each cell in the grid.

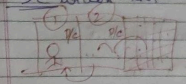
If ~~floor~~ cell is dirty, it cleans it by setting the value to 1.

floor(i)(j) = 0

④ After cleaning each cell, the current state of ~~floor~~ room is displayed.

To show ~~the~~ where the cleaner is at

><



① User inputs state of each room
↓
this is stored in a list room state

② Clean room → function that takes room state as argument

If clean room 0 to 3 and returns to 0th room

③ If states of room = 1 → dirty
clean it and makes the state = 0

④ After returning to 0th room it gives a completed message

Code:-

```
def clean_rooms(rooms):
    for i in range(len(rooms)):
        clean_room(rooms, i)

clean_room(room, 0)

def clean_room(rooms, i):
    if rooms[i] == 1:
        print(f"Cleaning room {i}")
        room[i] = 0
    else:
        print(f"Room {i} is already clean")

~~def main():~~

rooms = 4
room_state = []

print("Initial state for each room")
for room in range(rooms):
    state = int(input(f"Room {room} state (0/1): "))
    room_state.append(state)

clean_rooms(room_state)

print("All rooms have been present")
```

Output:-

```
Enter initial state for 4 rooms
Room 0 state: 0
Room 1 state: 1
Room 2 state: 1
Room 3 state: 0

Room 0 is already clean
Cleaning Room 1
Cleaning Room 2
Room 3 is already clean
Room 0 is already clean

All rooms have been present
```

```
0 indicates clean and 1 indicates dirty
harshala Rani 1BM21CS074
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Program 6

Knowledge Base - Entailment

Algorithm: $\alpha \mid= \beta$ → KB ⊢ Q

1 Knowledge Base Function

→ Define proposition symbols
→ Knowledge base using logical statements:

→ If p then q $p \rightarrow q$
→ If q then r $q \rightarrow r$
→ Not r $\sim r$

2 Query Entails Function

→ Takes knowledge base & query as input
→ Checks if KB entails query
⇒ By checking if there exists any satisfying assignment where conjunction of the KB & negation of query is satisfiable

→ If there is no satisfiable assignment, it returns 'true'

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

KB = create_knowledge_base()
query = symbols('p')
result = query_entails(KB, query)
```

KB ∧ ~Q

```
print("Knowledge Base", kb)
print("Query", query)
print("Query entails Knowledge Base:", result)
```

Program 6

Output:

Knowledge Base: ~r & (Implies(p,q) & Implies(q,r))

Query: p

Query entails Knowledge Base : False

p	q	r	p→q	q→r	~r	KB ∧ ~q
T	T	T	T	T	F	F
T	T	F	T	F	T	F
T	F	T	F	T	F	F
T	F	F	F	T	T	F
F	T	T	T	T	F	F
F	T	F	T	F	T	F
F	F	T	T	T	F	F
F	F	F	T	T	T	T

Satisfiable

Thus, not entailing.

Ex: p → It is a sunny
q → I go for a picnic
r → I carry an umbrella

p ⇒ q If it is sunny, I go for a picnic

q ⇒ r If I go for picnic, I carry an umbrella

~r ⇒ ~q I'm not carry an umbrella

Query → Is it sunny?

[NO]

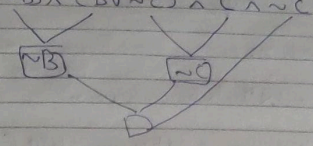
```
Harshala Rani-1BM21CS074
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

(Program-7)

Knowledge Base - Resolution

```
def negate_literal(literal)
    if literal[0] == '~'
        return literal[1:]
    else
        return '~' + literal

def resolve(C1, C2)
    resolved_clause = set(C1) | set(C2)
    for literal in C1:
        if negate_literal(literal) in C2:
            resolved_clause.remove(literal)
                        .remove(negate_literal(literal))
    return tuple(resolved_clause)

def resolution(KB)
    new_clauses = set()
    for i, C1 in enumerate(KB)
        for j, C2 in enumerate(KB)
            if i != j
                new_clause = resolve(C1, C2)
                if len(new_clause) > 0 & new_clause not in
                    new_clauses.add(new_clause)
```

KB: $(A \vee \sim B) \wedge (B \vee \sim C) \wedge C \wedge \sim C$



Algorithm:

1. Main function initializes the resolution process by splitting the input rules and calling 'resolve' function
2. Negate function negates the literal
3. Resolve function performs resolution method it iteratively considers pairs of clauses based on resolution rule until either a contradiction/a goal is reached.

```
Step    |Clause |Derivation
-----------------------------
 1.     | PvQ   | Given.
 2.     | PvR   | Given.
 3.     | ~PvR  | Given.
 4.     | RvS   | Given.
 5.     | Rv~Q  | Given.
 6.     | ~Sv~Q | Given.
 7.     | ~R    | Negated conclusion.
 8.     | QvR   | Resolved from PvQ and ~PvR.
 9.     | Pv~S  | Resolved from PvQ and ~Sv~Q.
 10.    | P     | Resolved from PvR and ~R.
 11.    | ~P    | Resolved from ~PvR and ~R.
 12.    | Rv~S  | Resolved from ~PvR and Pv~S.
 13.    | R     | Resolved from ~PvR and P.
 14.    | S     | Resolved from RvS and ~R.
 15.    | ~Q    | Resolved from Rv~Q and ~R.
 16.    | Q     | Resolved from ~R and QvR.
 17.    | ~S    | Resolved from ~R and Rv~S.
 18.    |       | Resolved ~R and R to ~RvR, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

Program -8
Unification

```
import re

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

def unify(exp1, exp2):
    if exp1 == exp2:
        return []
    if is_constant(exp1) and is_constant(exp2):
        if exp1 != exp2:
            return False
    if is_variable(exp1):
        if check_occurs(exp1, exp2):
            return False
        else
            return [(exp2, exp1)]
    if is_variable(exp2):
        if check_occurs(exp1, exp2):
            return False
        else
            return [(exp2, exp1)]
    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Predicates do not match. Cannot be unified")
        return False
    head1 = getFirst
    head2 = 6 " (exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return False
    if attributeCount(exp1) == 1:
        return initialSubstitution
    tail1 = apply(tail1, initialSubstitution)
    tail2 = " " 2
    remainingSolution = unify(tail1, tail2)
    if not remainingSolution:
        return False
    exp1 = knows(X)
    exp2 = " (Richard)
    Substitution = unify(exp1, exp2)
    print("Substitution")
```

Output: [('X', 'Richard')]

```
exp1 = knows(A,x)
exp2 = knows(y, mother(y))
```

Substitution:

[(A, y), (mother(y), x)]

Algorithm

function works by breaking down exp into their components, handling constants, variables and predicates appropriately, and recursively verifying the substitutions

→ If the 2 expressions are identical, returns an empty list (No Subs needed)

→ If 1 of the expression is a constant, unifies by substituting constant for the variable

→ If one of the expression is a variable, unifies by substituting variable for the constant

→ If the predicates of the exps don't match, unification fails

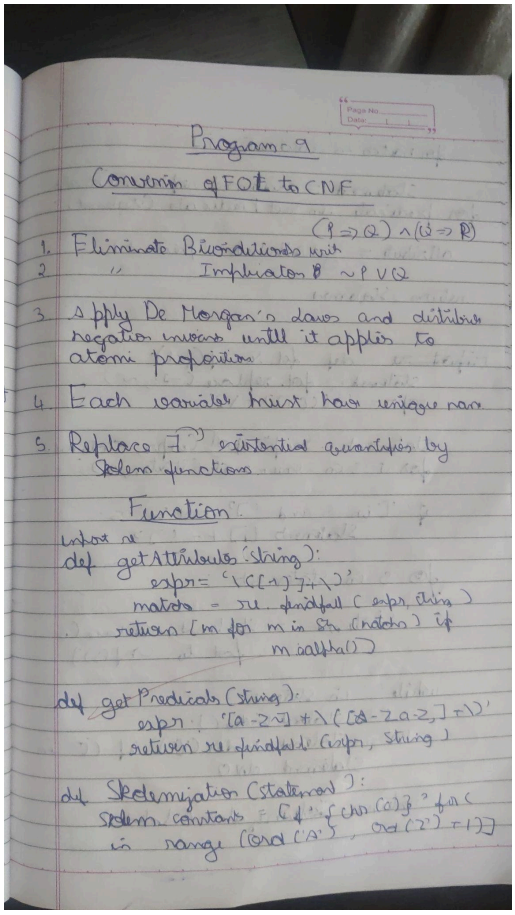
→ Recursively performs unification on the head & tail of the expressions

```
[ ]  exp1 = "knows(X)"
     exp2 = "knows(Richard)"
     substitutions = unify(exp1, exp2)
     print("Substitutions:")
     print(substitutions)
```

```
     Substitutions:
     [('X', 'Richard')]
```

```
[ ]  exp1 = "knows(A,x)"
     exp2 = "knows(y,mother(y))"
     substitutions = unify(exp1, exp2)
     print("Substitutions:")
     print(substitutions)
```

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



Program - 9

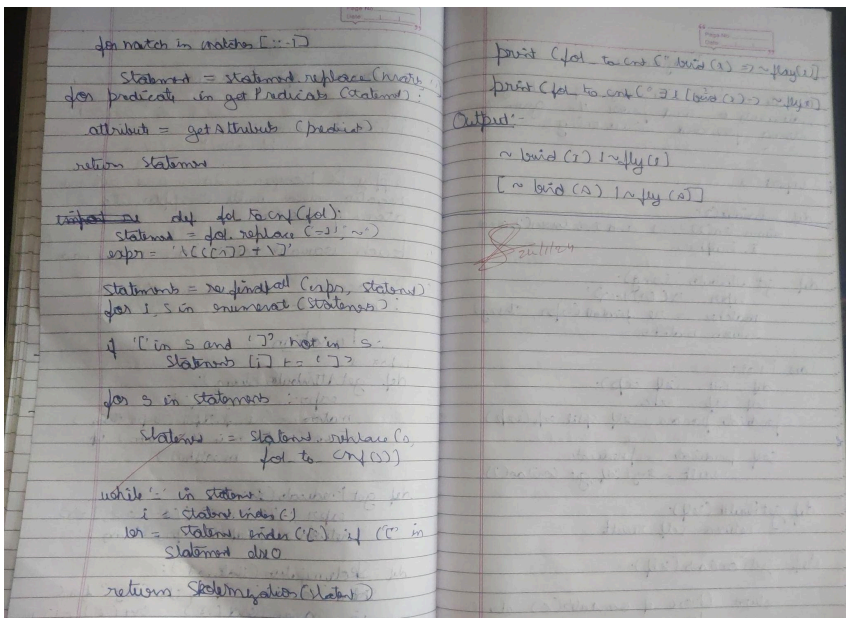
Conversion of FOL to CNF

$(P \Rightarrow Q) \wedge (\neg P \Rightarrow R)$

1. Eliminate Biconditionals and
2. " Implication $P \sim P \vee Q$
3. Apply De Morgan's laws and distribute negation inwards until it applies to atomic proposition
4. Each variable must have unique name
5. Replace $\exists$ existential quantifiers by Skolem functions

Function:

```
import re
def getAttributes(string):
    expr = '([a-zA-Z~]+)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if
            m.isalpha()]

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

def Skolemization(statement):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c
    in range(ord('A'), ord('Z')+1)]
```



```
    for match in matches[::-1]:
        statement = statement.replace(match,
    for predicate in getPredicates(statement):
        attributes = getAttributes(predicate)
    return statement

def fol_to_cnf(fol):
    statement = fol.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 None
    return Skolemization(statement)
```

```
print(fol_to_cnf("bird(x) => ~fly(x)"))
print(fol_to_cnf("~[bird(x) => ~fly(x)]"))
```

Output:-

```
~bird(x) | ~fly(x)
[~bird(A) | ~fly(A)]
```

```
print("Harshala Rani 1BM21CS074")
print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))
```

```
Harshala Rani 1BM21CS074
[~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
[animal(G(x))&~loves(x,G(x))]|[loves(F(x),x)]
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)
```

Program 10

Create a KB consisting of FOL logic statements and prove the given query using forward reasoning

```
import re

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

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

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

    def getResult(self):
        return self.result

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

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

class Implication:
    def __init__(self, exp):
        self.exp = exp
        l = exp.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getConstants()):
                        if v:
                            constants[val.params[i]] = fact.params[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.exp)[0], str(getAttributes(self.rhs.exp)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = '{}{}'.format(predicate, attributes)
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None

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))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.exp for f in self.facts])):
            print('\t{}. {}'.format(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('enemy(Nono, America)')
kb.tell('owns(Nono, M1)')
kb.tell('american(x) & weapon(y) & sells(x,y,z) & hostile(z) => criminal(x)')

kb.display()
```

Output:

```
Querying criminal(x)
    1. criminal(John)
```

Algorithm

1. Fact class:
   Represents facts in KB
   Stores result of the fact also
2. Implication class
   Represents implication in KB
   Stores LHS as a list of facts and RHS as a single fact
3. KB class
   Represents Knowledge Base info
   Provides method 'Tell' to add and 'query' to query KB

Result

```
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)')
```

```
Harshala Rani-1BM21CS074
Querying evil(x):
        1. evil(John)
```