2a, 4b, 5a, 6a

Don't do this practical for AI practical exam!!

6a only not b

PRACTICAL NO-1

A) AIM: Write a program to implement depth first search algorithm.

```
graph1 = {
    'A': set(['B', 'C']),
    'B': set(['A', 'D', 'E']),
    'C': set(['A', 'F']),
    'D': set(['B']),
    'E': set(['B', 'F']),
    'F': set(['C', 'E'])
}
def dfs(graph, node, visited):
    if node not in visited:
        visited.append(node)
        for n in graph[node]:
            dfs(graph, n, visited)
    return visited

visited = dfs(graph1, 'A', [])
print(visited)
```

OUTPUT:

```
======= RESTART: C:/Users/user/Desktop/practical Question/a:
['A', 'C', 'F', 'E', 'B', 'D']
```

B) AIM: Write a program to implement breadth first search algorithm

```
graph = {
    'A': set(['B', 'C']),
    'B': set(['A', 'D', 'E']),
    'C': set(['A', 'F']),
    'D': set(['B']),
    'E': set(['B', 'F']),
    'F': set(['C', 'E'])
}

# Implementing BFS to explore the graph
def bfs(start):
    queue = [start]
    levels = {}  # This keeps track of levels
    levels[start] = 0  # Depth of start node is 0
    visited = set(start)

    while queue:
        node = queue.pop(0)
        neighbours = graph[node]

        for neighbor in neighbours:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
                levels[neighbor] = levels[node] + 1

    print(levels)  # Print the level of each node
    return visited

# Printing nodes visited during BFS from node 'A'
print(str(bfs('A')))

# BFS to find all paths from start to goal
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]

    while queue:
        (vertex, path) = queue.pop(0)

        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
```

```
            else:
                queue.append((next, path + [next]))

# Finding and printing all paths from 'A' to 'F'
result = list(bfs_paths(graph, 'A', 'F'))
print(result)  # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]

# For finding the shortest path from start to goal
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None

# Finding and printing the shortest path from 'A' to 'F'
result1 = shortest_path(graph, 'A', 'F')
print(result1)  # ['A', 'C', 'F']
```

OUTPUT:

```
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
{'A': 0, 'B': 1, 'C': 1, 'D': 2, 'E': 2, 'F': 2}
{'D', 'F', 'C', 'A', 'E', 'B'}
[['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
['A', 'C', 'F']
```

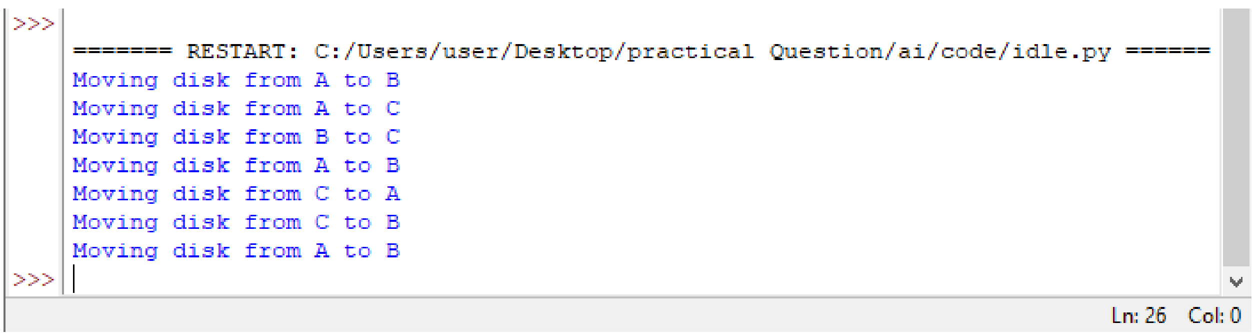
PRACTICAL NO- 2

B) AIM: Write a program to solve tower of Hanoi problem.

```
def moveTower(n, fromPole, toPole, withPole):
    if n:
        moveTower(n - 1, fromPole, withPole, toPole)
        print(f"Move disk from {fromPole} to {toPole}")
        moveTower(n - 1, withPole, toPole, fromPole)

moveTower(3, "A", "B", "C")
```

OUTPUT:



PRACTICAL NO.-3

A) AIM: Write a program to implement alpha beta search.

```
tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]
pruned = 0

def alphabeta(branch, depth, alpha, beta):
    global pruned
    for child in branch:
        if isinstance(child, list):
            alpha, beta = alphabeta(child, depth + 1, alpha, beta)
        else:
            if depth % 2 == 0: alpha = max(alpha, child)
            else: beta = min(beta, child)
        if alpha >= beta:
            pruned += 1
            break
    return alpha, beta

def search(tree, alpha=-15, beta=15):
    global pruned
    alpha, beta = alphabeta(tree, 0, alpha, beta)
    print(f"(alpha, beta): {alpha} {beta}")
    print(f"Result: {alpha}")
    print(f"Times pruned: {pruned}")

if __name__ == "__main__":
    search(tree)
```

```
========== RESTART: C:/Users/user/Desktop/practical Question/ai/3A.py ==========
(alpha, beta): 9 15
Result: 9
Times pruned: 0
```

```
>>>
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
(alpha, beta): 5 15
Result: 5
Times pruned: 1
>>>
Ln: 31   Col: 0
```

B) AIM: Write a program for Hill climbing problem.

```
import math

increment = 0.1
start = [1, 1]
points = [[1, 5], [6, 4], [5, 2], [2, 1]]

def dist(x1, y1):
    return sum((x1 - p[0]) ** 2 + (y1 - p[1]) ** 2 for p in points)

def move(x1, y1, inc):
    return dist(x1 + inc, y1), dist(x1 - inc, y1), dist(x1, y1 + inc), dist(x1, y1 - inc)

i = 1
minDist = dist(start[0], start[1])

while True:
    d1, d2, d3, d4 = move(start[0], start[1], increment)
    print(i, round(start[0], 2), round(start[1], 2))

    minMove = min(d1, d2, d3, d4)
    if minMove < minDist:
        if minMove == d1: start[0] += increment
        elif minMove == d2: start[0] -= increment
        elif minMove == d3: start[1] += increment
        else: start[1] -= increment
        minDist = minMove
        i += 1
    else: break
```

OUTPUT:

```
>>>
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
1 1 1
2 1.1 1
3 1.2 1
4 1.3 1
5 1.4 1
6 1.5 1
7 1.6 1
8 1.6 1.1
9 1.7 1.1
10 1.7 1.2
11 1.7 1.3
12 1.8 1.3
13 1.8 1.4
14 1.9 1.4
15 2.0 1.4
16 2.0 1.5
17 2.1 1.5
18 2.1 1.6
19 2.2 1.6
20 2.2 1.7
21 2.3 1.7
22 2.3 1.8
23 2.3 1.9
24 2.4 1.9
25 2.5 1.9
26 2.5 2.0
27 2.6 2.0
28 2.6 2.1
29 2.7 2.1
30 2.7 2.2
31 2.8 2.2
32 2.8 2.3
33 2.9 2.3
34 2.9 2.4
35 3.0 2.4
36 3.0 2.5
37 3.1 2.5
```

```
37 3.1 2.5
38 3.1 2.6
39 3.2 2.6
40 3.2 2.7
41 3.2 2.8
42 3.3 2.8
43 3.4 2.8
44 3.4 2.9
45 3.5 2.9
46 3.5 3.0
```

PRACTICAL NO-4

A) AIM: Write a program to implement A* algorithm.

```
from simpleai.search import SearchProblem, astar

GOAL = 'HELLO WORLD'

class HelloProblem(SearchProblem):
    def actions(self, state):
        return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ') if len(state) < len(GOAL) else []

    def result(self, state, action):
        return state + action

    def is_goal(self, state):
        return state == GOAL

    def heuristic(self, state):
        return sum(1 for i in range(len(state)) if state[i] != GOAL[i]) + (len(GOAL) - len(state))

result = astar(HelloProblem(initial_state=''))
print(result.state)
print(result.path())
```

OUTPUT:

```
PS C:\Users\user\Desktop\practical Question\ai\code> python -u "c:\Users\user\Desktop\practical Question\ai\code\test.py"
HELLO WORLD
[(None, ''), ('H', 'H'), ('E', 'HE'), ('L', 'HEL'), ('L', 'HELL'), ('O', 'HELLO'), (' ', 'HELLO '), ('W', 'HELLO W'), ('O', 'HELLO WO'),
('R', 'HELLO WOR'), ('L', 'HELLO WORL'), ('D', 'HELLO WORLD')]
PS C:\Users\user\Desktop\practical Question\ai\code>
```

PRACTICAL NO-5

B) Write a program to shuffle Deck of cards.

```
import random

# Create a deck of cards
suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
cardfaces = [str(i) for i in range(2, 11)] + ["J", "Q", "K", "A"]
deck = [f"{face} of {suit}" for suit in suits for face in cardfaces]

# Shuffle and display the deck
random.shuffle(deck)
print("\n".join(deck))
```

```
>>>
=============================================== RESTART: C:/Users/user/Desktop/practical Question/ai/code/:
Q of Spades
2 of Clubs
3 of Hearts
9 of Hearts
8 of Hearts
9 of Clubs
10 of Spades
6 of Spades
3 of Diamonds
5 of Diamonds
7 of Diamonds
4 of Clubs
J of Spades
A of Diamonds
5 of Spades
10 of Diamonds
J of Clubs
K of Diamonds
J of Diamonds
5 of Hearts
Q of Diamonds
A of Hearts
Q of Clubs
8 of Diamonds
8 of Spades
J of Hearts
2 of Hearts
6 of Diamonds
7 of Hearts
A of Clubs
A of Spades
K of Spades
K of Hearts
6 of Hearts
6 of Clubs
4 of Hearts
3 of Spades
2 of Spades
```

```
2 of Diamonds
9 of Diamonds
4 of Spades
Q of Hearts
5 of Clubs
10 of Clubs
7 of Spades
K of Clubs
9 of Spades
8 of Clubs
10 of Hearts
3 of Clubs
7 of Clubs
4 of Diamonds
>>>
```

PRACTICAL No.-7

AIM: Solve constraint satisfaction problem

```
from __future__ import print_function
from simpleai.search import CspProblem, backtrack, min_conflicts,
MOST_CONSTRAINED_VARIABLE, HIGHEST_DEGREE_VARIABLE,
LEAST_CONSTRAINING_VALUE

variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')
domains = {v: ['red', 'green', 'blue'] for v in variables}

def const_different(x, y):
    return x != y

constraints = [
    (('WA', 'NT'), const_different),
    (('WA', 'SA'), const_different),
    (('SA', 'NT'), const_different),
    (('SA', 'Q'), const_different),
    (('NT', 'Q'), const_different),
    (('SA', 'NSW'), const_different),
    (('Q', 'NSW'), const_different),
    (('SA', 'V'), const_different),
    (('NSW', 'V'), const_different),
]

problem = CspProblem(variables, domains, constraints)

for heuristic in [None, MOST_CONSTRAINED_VARIABLE,
HIGHEST_DEGREE_VARIABLE]:
    print(backtrack(problem, variable_heuristic=heuristic))

print(backtrack(problem, value_heuristic=LEAST_CONSTRAINING_VALUE))
print(backtrack(problem, variable_heuristic=MOST_CONSTRAINED_VARIABLE,
value_heuristic=LEAST_CONSTRAINING_VALUE))
print(backtrack(problem, variable_heuristic=HIGHEST_DEGREE_VARIABLE,
value_heuristic=LEAST_CONSTRAINING_VALUE))
print(min_conflicts(problem))
```

OUTPUT:

```
PS C:\Users\user\Desktop\practical Question\ai\code> python -u "c:\Users\user\Desktop\practical Question\ai
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'SA': 'red', 'NT': 'green', 'Q': 'blue', 'NSW': 'green', 'WA': 'blue', 'V': 'blue', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'SA': 'red', 'NT': 'green', 'Q': 'blue', 'NSW': 'green', 'WA': 'blue', 'V': 'blue', 'T': 'red'}
{'WA': 'blue', 'NT': 'red', 'SA': 'green', 'Q': 'blue', 'NSW': 'red', 'V': 'blue', 'T': 'red'}
PS C:\Users\user\Desktop\practical Question\ai\code>
```

PRACTICAL No.-8

A) AIM: Derive the expressions based on associative law

```
def demonstrate_associative_law(a, b, c):
    # Associative Law of Addition
    sum1 = (a + b) + c
    sum2 = a + (b + c)
    # Associative Law of Multiplication
    product1 = (a * b) * c
    product2 = a * (b * c)
    print(f"Associative Law of Addition:")
    print(f"(a + b) + c = {sum1} and a + (b + c) = {sum2}")
    print(f"Result: {'Equal' if sum1 == sum2 else 'Not Equal'}\n")
    print(f"Associative Law of Multiplication:")
    print(f"(a * b) * c = {product1} and a * (b * c) = {product2}")
    print(f"Result: {'Equal' if product1 == product2 else 'Not Equal'}")# Example usage
a = 2
b = 3
c = 4
demonstrate_associative_law(a, b, c)
```

OUTPUT:

```
practical Question/ai/code/idle.py ============================================
===
Associative Law of Addition:
(a + b) + c = 9 and a + (b + c) = 9
Result: Equal

Associative Law of Multiplication:
(a * b) * c = 24 and a * (b * c) = 24
Result: Equal
>>>
                                                              Ln: 254    Col: 0
```

B) AIM: Derive the expressions based on distributive law

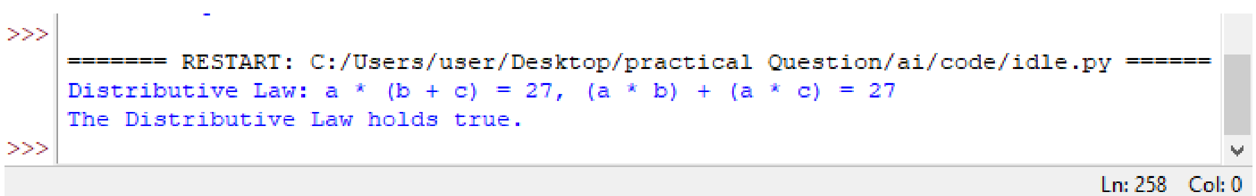
```
def distributive_law(a, b, c):
    # Calculate both sides of the distributive law
    left_side = a * (b + c)          # Left side: a * (b + c)
    right_side = (a * b) + (a * c)    # Right side: (a * b) + (a * c)
    return left_side, right_side

# Test values
a = 3
b = 4
c = 5

# Check Distributive Law
left_result, right_result = distributive_law(a, b, c)
print(f"Distributive Law: a * (b + c) = {left_result}, (a * b) + (a * c) = {right_result}")

# Verify if both sides are equal
if left_result == right_result:
    print("The Distributive Law holds true.")
else:
    print("The Distributive Law does not hold true.")
```

OUTPUT:

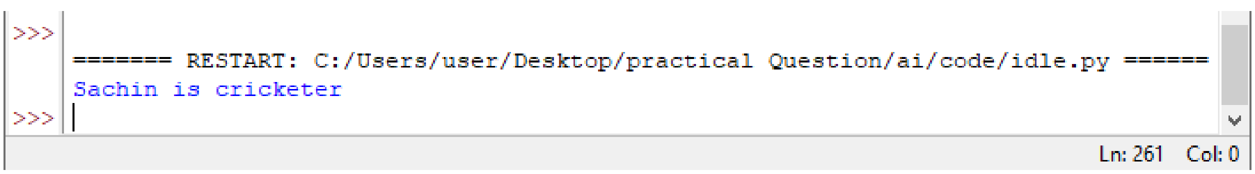


PRACTICAL No.-9

AIM: derive the predicate .(for eg: sachin is batman , batman is cricketer)-> schin is cricketer

```
# Define facts as a dictionary where key is the subject and value is the predicate
facts = {
    'Sachin': 'batsman',
    'batsman': 'cricketer'
}
# Function to derive the final fact from the initial fact
def derive_fact(starting_subject, target_predicate):
    current_subject = starting_subject
    while current_subject in facts:
        current_subject = facts[current_subject]
        if current_subject == target_predicate:
            return f"{starting_subject} is {target_predicate}"
    return "Cannot derive the fact"
# Define the starting subject and the target predicate
starting_subject = 'Sachin'
target_predicate = 'cricketer'
# Derive and print the fact
result = derive_fact(starting_subject, target_predicate)
print(result)
```

OUTPUT:



PRACTICAL No.-10

AIM: Write a program which contains three predicates: male, female, parent.

```
# Facts about individuals
male = {'John', 'Robert', 'Michael', 'Kevin'}
female = {'Mary', 'Patricia', 'Jennifer', 'Linda'}
parents = {
   'John': ['Michael', 'Sarah'],
   'Mary': ['Michael', 'Sarah'],
   'Robert': ['Kevin'],
   'Patricia': ['Kevin']
}

def is_parent(parent, child):
   return child in parents.get(parent, [])

def is_grandparent(grandparent, grandchild):
   return any(is_parent(parent, grandchild) for parent in parents.get(grandparent, []))

def is_sibling(sibling1, sibling2):
   return sibling1 in parents.get(sibling2, [])

def is_uncle(uncle, nephew):
   return uncle in male and any(is_sibling(uncle, parent) for parent in parents.get(nephew, []))

def is_aunt(aunt, niece):
   return aunt in female and any(is_sibling(aunt, parent) for parent in parents.get(niece, []))

def is_cousin(cousin1, cousin2):
   return any(parent1 != parent2 for parent1 in parents.get(cousin1, []) for parent2 in
parents.get(cousin2, []))

# Example Queries
print("Is John the father of Michael?", is_parent('John', 'Michael'))
print("Is Mary the mother of Sarah?", is_parent('Mary', 'Sarah'))
print("Is Robert a grandfather of Michael?", is_grandparent('Robert', 'Michael'))
print("Is Patricia an aunt of Kevin?", is_aunt('Patricia', 'Kevin'))
```

OUTPUT:

```
>>>
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
Is John the father of Michael? True
Is Mary the mother of Sarah? True
Is Robert a grandfather of Michael? False
Is Patricia an aunt of Kevin? False
>>> |
Ln: 267  Col: 0
```