P1.wap to implement depth first search algorithm

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = set()
def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

P1B. Wap to implement breadth first search algorithm

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = []
queue = []
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)
  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

P2 wap to implement A* algorithm

```
from collections import deque
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list
    def get_neighbors(self, v):
        return self.adjacency_list[v]
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]
    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])
        g = {}
        g[start_node] = 0
        parents = {}
        parents[start_node] = start_node
        while len(open_list) > 0:
            n = None
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;
            if n == None:
                print('Path does not exist!')
                return None
            if n == stop_node:
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]
                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path
            for (m, weight) in self.get_neighbors(n):
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
```

```
                parents[m] = n
                if m in closed_list:
                    closed_list.remove(m)
                    open_list.add(m)
            open_list.remove(n)
            closed_list.add(n)
        print('Path does not exist!')
        return None
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

P2b wap to solve water jug problem

```
from collections import deque
class State:
    def __init__(self, jug_a, jug_b, steps=None):
        self.jug_a = jug_a
        self.jug_b = jug_b
        self.steps = steps or []
    def __eq__(self, other):
        return self.jug_a == other.jug_a and self.jug_b == other.jug_b
    def __hash__(self):
        return hash((self.jug_a, self.jug_b))
def water_jug_problem(capacity_a, capacity_b, target):
    initial_state = State(0, 0, steps=[(0, 0)])
    visited = set()
    queue = deque([initial_state])
    while queue:
        current_state = queue.popleft()
        visited.add(current_state)
        if current_state.jug_a == target or current_state.jug_b == target:
            return current_state
        # Generate possible states and add steps
        possible_states = [
            State(capacity_a, current_state.jug_b, current_state.steps + [(capacity_a,
current_state.jug_b)]),  # Fill jug A
            State(current_state.jug_a, capacity_b, current_state.steps + [(current_state.jug_a,
capacity_b)]),  # Fill jug B
            State(0, current_state.jug_b, current_state.steps + [(0, current_state.jug_b)]),  #
Empty jug A
            State(current_state.jug_a, 0, current_state.steps + [(current_state.jug_a, 0)]),  #
Empty jug B
        ]
```

```
        if current_state.jug_a > 0 and current_state.jug_b < capacity_b:
            pour_amount = min(current_state.jug_a, capacity_b - current_state.jug_b)
            possible_states.append(State(current_state.jug_a - pour_amount,
current_state.jug_b + pour_amount,
                                         current_state.steps + [(current_state.jug_a - pour_amount,
current_state.jug_b + pour_amount)]))  # Pour from A to B
        if current_state.jug_b > 0 and current_state.jug_a < capacity_a:
            pour_amount = min(current_state.jug_b, capacity_a - current_state.jug_a)
            possible_states.append(State(current_state.jug_a + pour_amount,
current_state.jug_b - pour_amount,
                                         current_state.steps + [(current_state.jug_a + pour_amount,
current_state.jug_b - pour_amount)]))  # Pour from B to A
        for state in possible_states:
            if state not in visited:
                queue.append(state)
                visited.add(state)
    return None
# Example usage
capacity_a = 4
capacity_b = 3
target = 2
solution = water_jug_problem(capacity_a, capacity_b, target)
if solution:
    print("Solution found:")
    for step in solution.steps:
        print(f"Jug A: {step[0]}, Jug B: {step[1]}")
else:
    print("No solution found.")
```

P3A wap to solve 4queens problem

```
def print_board(board):
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print("\n")
def is_safe(board, row, col):
    # Check this row on the left side
    for i in range(col):
        if board[row][i]:
            return False
    # Check upper diagonal on the left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j]:
            return False
    # Check lower diagonal on the left side
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):
        if board[i][j]:
            return False
```

```
    return True
def solve_n_queen(board, col):
    if col >= len(board):
        return True
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_n_queen(board, col + 1):
                return True
            board[i][col] = 0
    return False
def solve_n_queen_problem(N):
    board = [[0] * N for _ in range(N)]
    if solve_n_queen(board, 0):
        print_board(board)
    else:
        print("No solution exists")
# Input from user
N = int(input("Enter the size of the board (N): "))
solve_n_queen_problem(N)
```

P3b wap to solve tower of hanoi

```
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, target, auxiliary)
 # Move the nth disk from source to target peg
    print(f"Move disk {n} from {source} to {target}")
    # Move (n-1) disks from auxiliary peg to target peg using the source peg
    tower_of_hanoi(n - 1, auxiliary, source, target)
if __name__ == "__main__":
    n = 3  # Change n to the number of disks you want to solve the problem for
    tower_of_hanoi(n, 'A', 'B', 'C')
```

P4A wap to solve tic tac toe problem

```
import random

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

def check_win(board, player):
    for row in board:
        if all(cell == player for cell in row):
```

```
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

def is_full(board):
    for row in board:
        if " " in row:
            return False
    return True

def get_empty_cells(board):
    empty_cells = []
    for row in range(3):
        for col in range(3):
            if board[row][col] == " ":
                empty_cells.append((row, col))
    return empty_cells

def min_max(board, depth, is_maximizing):
    if check_win(board, "X"):
        return -1
    elif check_win(board, "O"):
        return 1
    elif is_full(board):
        return 0

    if is_maximizing:
        max_eval = -float("inf")
        for row, col in get_empty_cells(board):
            board[row][col] = "O"
            eval = min_max(board, depth + 1, False)
            board[row][col] = " "
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float("inf")
        for row, col in get_empty_cells(board):
            board[row][col] = "X"
            eval = min_max(board, depth + 1, True)
            board[row][col] = " "
            min_eval = min(min_eval, eval)
        return min_eval

def get_best_move(board):
```

```
    best_move = None
    best_eval = -float("inf")
    for row, col in get_empty_cells(board):
        board[row][col] = "O"
        eval = min_max(board, 0, False)
        board[row][col] = " "
        if eval > best_eval:
            best_eval = eval
            best_move = (row, col)
    return best_move

def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    player_turn = "X"

    print("Welcome to Tic-Tac-Toe!")
    print_board(board)

    while True:
        if player_turn == "X":
            row, col = map(int, input("Enter row and column (e.g., 0 0): ").split())
            if board[row][col] != " ":
                print("Invalid move. Try again.")
                continue
        else:
            print("Computer's turn:")
            row, col = get_best_move(board)

        board[row][col] = player_turn
        print_board(board)

        if check_win(board, player_turn):
            print(f"{player_turn} wins!")
            break
        elif is_full(board):
            print("It's a draw!")
            break

        player_turn = "X" if player_turn == "O" else "O"

if __name__ == "__main__":
    main()
```

P4b wap to solve shuffle dock cards

```
import itertools
import random
```

```
# Define the ranks, suits, and create a deck of cards
ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']
suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
deck = list(itertools.product(ranks, suits))
# Shuffle the deck
random.shuffle(deck)
# Draw 5 random cards
hand = random.sample(deck, 5)

# Print the drawn cards
print("Your hand:")
for card in hand:
    print(f"{card[0]} of {card[1]}")
```

P5 wap to solve number word program

```
import copy

# Define the initial and goal states for a 3x3 sliding puzzle
initial_state = [[2, 8, 3],
                 [1, 6, 4],
                 [7, 0, 5]]

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

# Define possible moves (left, right, up, down)
moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]
move_names = ['left', 'right', 'up', 'down']

# Helper function to print the puzzle board
def print_board(board):
    for row in board:
        print(" ".join(map(str, row)))
    print()

# Helper function to find the coordinates of a value in the board
def find_value(board, value):
    for i in range(3):
        for j in range(3):
            if board[i][j] == value:
                return i, j

# Perform breadth-first search to solve the puzzle and print all steps
def bfs_search(initial_state, goal_state):
    visited = set()
    queue = [(initial_state, [])]
```

```
    while queue:
        current_state, path = queue.pop(0)

        if current_state == goal_state:
            print("Solution Steps:")
            for step in path:
                print(f"Step {step[0]}: Move {step[1]}")
                print_board(step[2])
            print("Goal state reached!")
            return

        i, j = find_value(current_state, 0)  # Find the empty space

        for move, move_name in zip(moves, move_names):
            new_i, new_j = i + move[0], j + move[1]
            if 0 <= new_i < 3 and 0 <= new_j < 3:
                new_state = copy.deepcopy(current_state)
                new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]

                if tuple(map(tuple, new_state)) not in visited:
                    visited.add(tuple(map(tuple, new_state)))
                    new_path = path + [(len(path) + 1, move_name, new_state)]
                    queue.append((new_state, new_path))

# Start the search from the initial state
print("Initial State:")
print_board(initial_state)
bfs_search(initial_state, goal_state)
```

P6 wap to constrain satisfaction

```
from constraint import Problem

# Define the variables and their domains
variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')
domains = {v: ['red', 'green', 'blue'] for v in variables}
# Create a CSP problem instance
problem = Problem()
# Add variables and domains to the problem
for variable, domain in domains.items():
    problem.addVariable(variable, domain)
# Define the constraints
def not_equal_constraint(variable1, variable2):
    return variable1 != variable2

# Add the constraints for neighboring regions not having the same color
```

```
for region1, region2 in [('WA', 'NT'), ('WA', 'SA'), ('NT', 'SA'), ('SA', 'Q'), ('SA', 'NSW'), ('SA',
'V'), ('Q', 'NSW'), ('NSW', 'V')]:
    problem.addConstraint(not_equal_constraint, (region1, region2))
# Find the solutions
solutions = problem.getSolutions()
# Print the solutions
for solution in solutions:
    print(solution)
```

P7 wap derive the expression based on distribution law

```
# Define variables
a = 3
b = 5
c = 2

# Applying the Distributive Law
left_side = a * (b + c)
right_side = (a * b) + (a * c)

# Check if both sides are equal
if left_side == right_side:
    print(f"Distributive Law holds: {a} * ({b} + {c}) = {left_side} = ({a} * {b}) + ({a} * {c})")
else:
    print(f"Distributive Law does not hold: {a} * ({b} + {c}) = {left_side}, ({a} * {b}) + ({a} * {c}) =
{right_side}")
```

P8 wap based on associative

```
# Define variables
a = 4
b = 2
c = 6
# Applying the associative law for multiplication
result1 = (a * b) * c
result2 = a * (b * c)
# Check if the results are equal
if result1 == result2:
    print("Associative law for multiplication holds.")
    print(f"({a} * {b}) * {c} = {result1}")
    print(f"{a} * ({b} * {c}) = {result2}")
else:
    print("Associative law for multiplication does not hold.")
```

P9 wap derives predicate eg- sachin is batsman

```
# Define the relationships
relationships = {
```

```
    "Sachin": "batsman",
    "batsman": "cricketer"
}

# Function to derive the final predicate
def derive_predicate(subject, relationships):
    if subject in relationships:
        middle_term = relationships[subject]  # e.g., Sachin is a batsman
        if middle_term in relationships:
            final_predicate = relationships[middle_term]  # e.g., batsman is a cricketer
            return f"{subject} is a {final_predicate}"
        else:
            return f"{subject} is a {middle_term}"
    else:
        return "Relationship not found"

# Test the function
subject = "Sachin"
result = derive_predicate(subject, relationships)
print(result)

batsman(sachin).
cricketer(X) :- batsman(X).
```