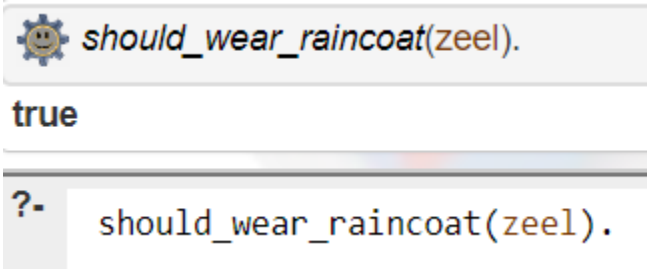
Practical No.1

Aim: Write the following in form of Facts and rules and solve the query
% Today is rainy
% Zeel is a person
% Every person should wear raincoat if it is rainy today
% Query: Should zeel wear raincoat today ?

Program Code:

```
 % Facts
rainy(today).
person(zeel).

% Rule
should_wear_raincoat(Person) :-
    person(Person),
    rainy(today).
```

Output Screenshot:



Practical No.2

Aim: **Use SWI – Prolog for answering the following questions (load the rules in the file familytree.pl):**

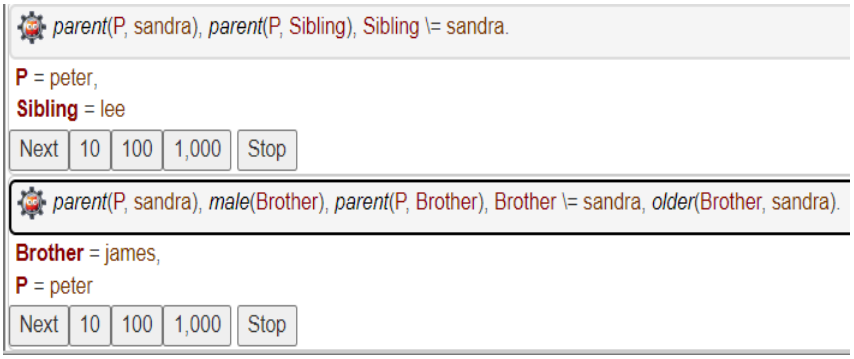
1. **Is Albert a parent of Peter?**
2. **Who is the child of Jim?**
3. **Who are the parents of Brian?**
4. **Is Irene a grandparent of Brian?**
5. **Find all the grandchildren of Irene**
6. **Now add the following rule to familytree.pl and re-consult:**
   older(Person1, Person2) :- yearOfBirth(Person1, Year1), yearOfBirth(Person2, Year2), Year2 > Year1.
7. **Who is older than Pat?**
8. **Who is younger than Darren?**
9. **List the siblings of Sandra.**
10. **Who is the older brother of Sandra?**
11. **Find the predecessors of Kyle.**
12. **Does Kate have a sister?**
13. **How many females and males are there in the knowledge base?**

Program Code:

```
% Program: family.pl

parent(albert, jim).
parent(albert, peter).
parent(jim, brian).
parent(john, darren).
parent(peter, lee).
parent(peter, sandra).
parent(peter, james).
parent(peter, kate).
parent(peter, kyle).
parent(brian, jenny).
parent(irene, jim).
parent(irene, peter).
parent(pat, brian).
parent(pat, darren).
parent(amanda, jenny).

% female(Person)
%
female(irene).
female(pat).
female(lee).
```

```
female(sandra).
female(jenny).
female(amanda).
female(kate).

% male(Person)
%
male(albert).
male(jim).
male(peter).
male(brian).
male(john).
male(darren).
male(james).
male(kyle).

% yearOfBirth(Person, Year).
%
yearOfBirth(irene, 1923).
yearOfBirth(pat, 1954).
yearOfBirth(lee, 1970).
yearOfBirth(sandra, 1973).
yearOfBirth(jenny, 2004).
yearOfBirth(amanda, 1979).
yearOfBirth(albert, 1926).
yearOfBirth(jim, 1949).
yearOfBirth(peter, 1945).
yearOfBirth(brian, 1974).
yearOfBirth(john, 1955).
yearOfBirth(darren, 1976).
yearOfBirth(james, 1969).
yearOfBirth(kate, 1975).
yearOfBirth(kyle, 1976).
older(Person1, Person2) :- yearOfBirth(Person1, Year1), yearOfBirth(Person2, Year2), Year2 > Year1.
```

Output Screenshot:


parent(P, sandra), parent(P, Sibling), Sibling \= sandra.
P = peter,
Sibling = lee
Next 10 100 1,000 Stop
parent(P, sandra), male(Brother), parent(P, Brother), Brother \= sandra, older(Brother, sandra).
Brother = james,
P = peter
Next 10 100 1,000 Stop

Practical No.3

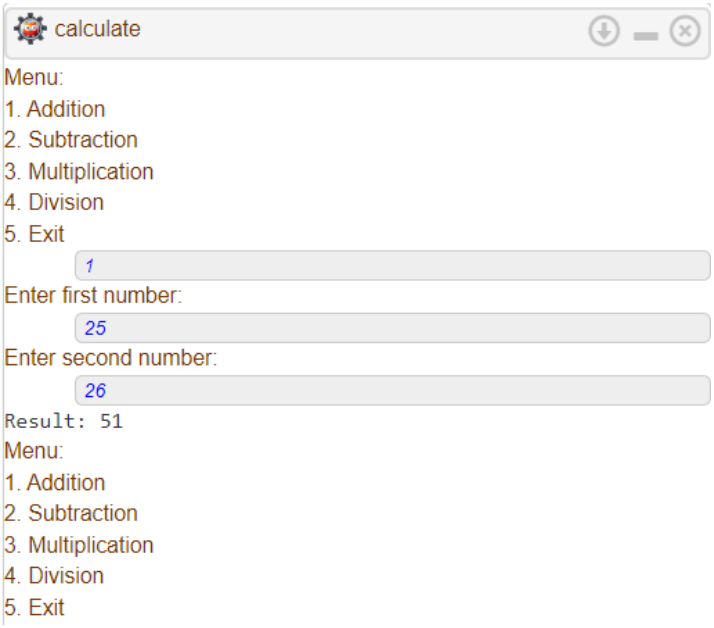
Aim: 1. Write a prolog program to implement a Menu Driven Calculator.

Program Code:

```
% Menu driven calculator program
calculate :-
   writeln('Menu:'),
   writeln('1. Addition'),
   writeln('2. Subtraction'),
   writeln('3. Multiplication'),
   writeln('4. Division'),
   writeln('5. Exit'),
   read(Choice),
   Choice \= 5, !,
   writeln('Enter first number:'),
   read(Num1),
   writeln('Enter second number:'),
   read(Num2),
   perform_calculation(Choice, Num1, Num2),
   calculate.

perform_calculation(1, Num1, Num2) :-
   Sum is Num1 + Num2,
   format('Result: ~w~n', [Sum]).
perform_calculation(2, Num1, Num2) :-
   Difference is Num1 - Num2,
   format('Result: ~w~n', [Difference]).
perform_calculation(3, Num1, Num2) :-
   Product is Num1 * Num2,
   format('Result: ~w~n', [Product]).
perform_calculation(4, Num1, Num2) :-
   (Num2 \= 0 ->
      Quotient is Num1 / Num2,
      format('Result: ~w~n', [Quotient]) ;
      writeln('Error: Division by zero')).

perform_calculation(_, _, _) :-
```

```
    writeln('Invalid choice. Please try again.').
```

Output Screenshot:



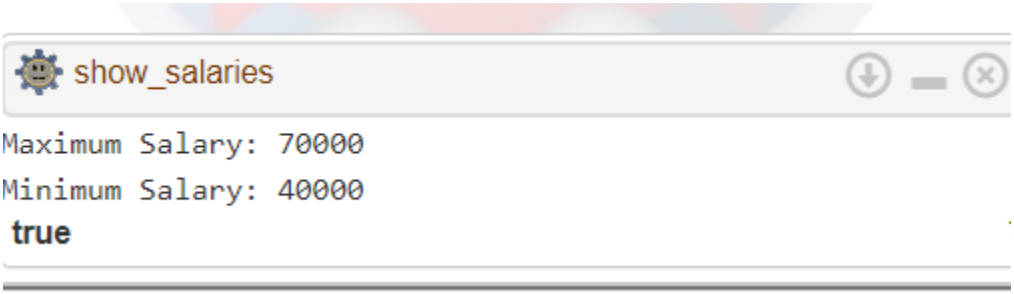
Aim: 2. Write a prolog program to find maximum and minimum salaries of given 3 employees.

Program Code:

```
employee(alice, 50000).
employee(bob, 70000).
employee(charlie, 40000).

max_salary(Max) :-
    findall(Salary, employee(_, Salary), Salaries),
    max_list(Salaries, Max).

min_salary(Min) :-
    findall(Salary, employee(_, Salary), Salaries),
    min_list(Salaries, Min).
```

```
show_salaries :-
   max_salary(Max),
   min_salary(Min),
   format('Maximum Salary: ~w~n', [Max]),
   format('Minimum Salary: ~w~n', [Min]).
```

Output Screenshot:

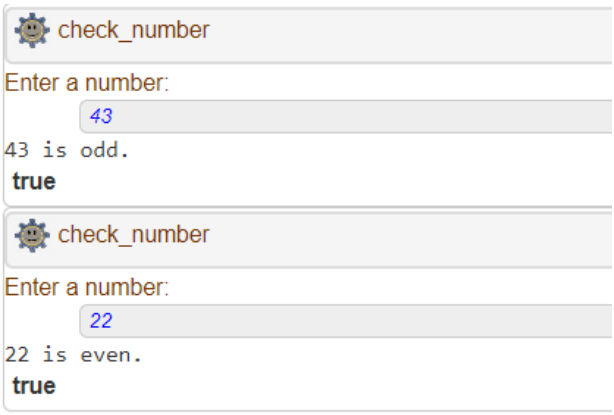


Aim: 3. Write a prolog program to check whether a given number is odd or even.

Program Code:

```
even(Number) :-
   0 is Number mod 2.

odd(Number) :-
   1 is Number mod 2.

check_number :-
   writeln('Enter a number:'),
   read(Number),
   (even(Number) ->
      format('~w is even.~n', [Number]) ;
      format('~w is odd.~n', [Number])).
```

Output Screenshot:



Practical No.4

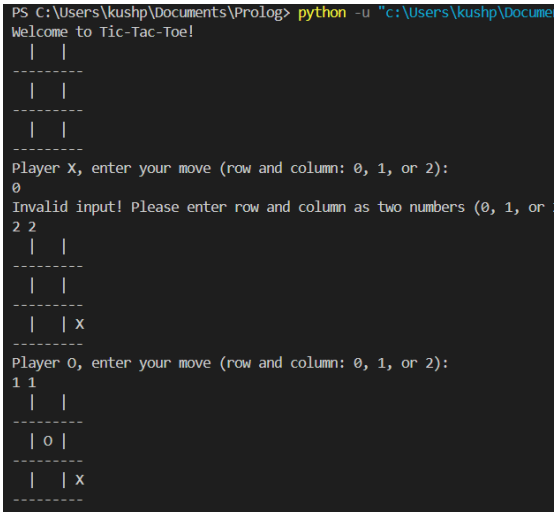
Aim: Write a program to implement Tic-Tac-Toe game problem

Program Code:

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

def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]

    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]

    return None

def is_draw(board):
    return all(cell != " " for row in board for cell in row)

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"

    while True:
        print_board(board)
        print(f"Player {current_player}, enter your move (row and column: 0, 1, or 2): ")

        # Input validation loop
        while True:
            try:
                row, col = map(int, input().split())
                if row in [0, 1, 2] and col in [0, 1, 2]:
                    break
```

```
            else:
                print("Invalid input! Please enter row and column as two numbers (0, 1, or
2).")
        except ValueError:
            print("Invalid input! Please enter row and column as two numbers (0, 1, or 2).")

    if board[row][col] == " ":
        board[row][col] = current_player
        winner = check_winner(board)

        if winner:
            print_board(board)
            print(f"Player {winner} wins!")
            break
        elif is_draw(board):
            print_board(board)
            print("It's a draw!")
            break

        # Switch players
        current_player = "O" if current_player == "X" else "X"
    else:
        print("Invalid move! Try again.")

if __name__ == "__main__":
    print("Welcome to Tic-Tac-Toe!")
    play_game()
```

Output Screenshot:



Practical No.5

Aim: Write a program to implement BFS (for 8 puzzle problem or Water Jug problem or any AI search problem)

Program Code:

```
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_position, moves=0):
        self.board = board
        self.zero_position = zero_position
        self.moves = moves

    def get_possible_moves(self):
        x, y = self.zero_position
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        possible_moves = []

        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                possible_moves.append((new_x, new_y))

        return possible_moves

    def move(self, new_position):
        x, y = self.zero_position
        new_x, new_y = new_position
        new_board = [list(row) for row in self.board]
        new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
        return PuzzleState(new_board, (new_x, new_y), self.moves + 1)

    def is_goal(self):
        return self.board == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    def __str__(self):
        return "\n".join(" ".join(str(num) for num in row) for row in self.board)

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()
    visited.add(tuple(map(tuple, initial_state.board)))

    while queue:
```

```
        current_state = queue.popleft()
        if current_state.is_goal():
            return current_state.moves

        for new_position in current_state.get_possible_moves():
            new_state = current_state.move(new_position)
            state_tuple = tuple(map(tuple, new_state.board))
            if state_tuple not in visited:
                visited.add(state_tuple)
                queue.append(new_state)

    return -1  # Goal not reachable

if __name__ == "__main__":
    initial_board = [
        [1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]
    ]
    zero_position = (1, 1)  # Position of the empty space (0)

    initial_state = PuzzleState(initial_board, zero_position)
    result = bfs(initial_state)

    if result != -1:
        print(f"Goal reached in {result} moves!")
    else:
        print("Goal not reachable.")
```

Output Screenshot:

```
PS C:\Users\kushp\Documents\Prolog> python -u "
Goal reached in 2 moves!
```

Practical No.6

Aim: Write a program to implement DFS (for 8 puzzle problem or Water Jug problem or any AI search problem)

Program Code:

```
class PuzzleState:
    def __init__(self, board, zero_position, moves=0):
        self.board = board
        self.zero_position = zero_position
        self.moves = moves

    def get_possible_moves(self):
        x, y = self.zero_position
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        possible_moves = []

        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                possible_moves.append((new_x, new_y))

        return possible_moves

    def move(self, new_position):
        x, y = self.zero_position
        new_x, new_y = new_position
        new_board = [list(row) for row in self.board]
        new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
        return PuzzleState(new_board, (new_x, new_y), self.moves + 1)

    def is_goal(self):
        return self.board == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    def __str__(self):
        return "\n".join(" ".join(str(num) for num in row) for row in self.board)

def dfs(state, visited):
    if state.is_goal():
        return state.moves
```

```
    visited.add(tuple(map(tuple, state.board)))

    for new_position in state.get_possible_moves():
        new_state = state.move(new_position)
        state_tuple = tuple(map(tuple, new_state.board))

        if state_tuple not in visited:
            result = dfs(new_state, visited)
            if result != -1:
                return result

    return -1  # Goal not reachable

if __name__ == "__main__":
    initial_board = [
        [1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]
    ]
    zero_position = (1, 1)  # Position of the empty space (0)

    initial_state = PuzzleState(initial_board, zero_position)
    visited = set()
    result = dfs(initial_state, visited)

    if result != -1:
        print(f"Goal reached in {result} moves!")
    else:
        print("Goal not reachable.")
```

Output Screenshot:

Practical No.7

Aim: Write a program to implement Single Player Game (Using Heuristic Function)

Program Code:

```
class GameState:
    def __init__(self, current_value, target_value):
        self.current_value = current_value
        self.target_value = target_value

    def heuristic(self):
        return abs(self.target_value - self.current_value)

    def get_possible_moves(self):
        return [
            self.current_value + 5,
            self.current_value * 2,
            self.current_value - 3
        ]

    def is_goal(self):
        return self.current_value == self.target_value

def find_best_move(state):
    best_move = None
    best_heuristic = float('inf')

    for move in state.get_possible_moves():
        temp_state = GameState(move, state.target_value)
        if temp_state.heuristic() < best_heuristic:
            best_heuristic = temp_state.heuristic()
            best_move = move

    return best_move

def play_game(initial_value, target_value):
    current_state = GameState(initial_value, target_value)
    moves = 0

    while not current_state.is_goal():
        best_move = find_best_move(current_state)
        if best_move is None:
            print("No valid moves available.")
```

```
            return

        current_state = GameState(best_move, target_value)
        moves += 1
        print(f"Current value: {current_state.current_value}, Moves: {moves}")

    print(f"Target reached in {moves} moves!")

if __name__ == "__main__":
    initial_value = 0  # Starting number
    target_value = 23  # Target number to reach
    play_game(initial_value, target_value)
```

Output Screenshot:

```
PS C:\Users\kushp\Documents\Prolog> python -u
Current value: 5, Moves: 1
Current value: 10, Moves: 2
Current value: 20, Moves: 3
Current value: 25, Moves: 4
Current value: 22, Moves: 5
Current value: 27, Moves: 6
Current value: 24, Moves: 7
Current value: 21, Moves: 8
Current value: 26, Moves: 9
Current value: 23, Moves: 10
Target reached in 10 moves!
```

Practical No.8

Aim: Write a program to implement A* algorithm

Program Code:

```
import heapq

class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0  # Cost from start to this node
        self.h = 0  # Heuristic cost to goal
        self.f = 0  # Total cost

    def __eq__(self, other):
        return self.position == other.position

    def __lt__(self, other):
        return self.f < other.f  # Compare nodes based on f value

def heuristic(a, b):
    # Manhattan distance
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(grid, start, goal):
    open_list = []
    closed_list = []

    start_node = Node(start)
    goal_node = Node(goal)

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        closed_list.append(current_node)

        # Goal check
        if current_node == goal_node:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path

        # Generate children
        neighbors = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # 4 possible directions
        for new_position in neighbors:
```

```
        node_position = (current_node.position[0] + new_position[0],
current_node.position[1] + new_position[1])

        # Check if it's within the grid boundaries
        if (0 <= node_position[0] < len(grid)) and (0 <= node_position[1] < len(grid[0])) and
grid[node_position[0]][node_position[1]] == 0:
            child_node = Node(node_position, current_node)

            # Check if the child node is in the closed list
            if child_node in closed_list:
                continue

            # Calculate g, h, f values
            child_node.g = current_node.g + 1
            child_node.h = heuristic(child_node.position, goal_node.position)
            child_node.f = child_node.g + child_node.h

            # Check if the child node is in the open list
            if any(child_node == item and child_node.g > item.g for item in open_list):
                continue

            heapq.heappush(open_list, child_node)

    return None  # Path not found

# Example usage
if __name__ == "__main__":
    # Define the grid (0 = walkable, 1 = obstacle)
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0]
    ]

    start = (0, 0)  # Starting position
    goal = (4, 4)   # Goal position

    path = astar(grid, start, goal)
    print("Path found:", path)
```

Output Screenshot:

```
PS C:\Users\kushp\Documents\Prolog> python -u "c:\Users\kushp\Documents\Prolog\Prac8.py"
Path found: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]
```

Practical No.9

Aim: Write a program to implement mini-max algorithm for any game development.

Program Code:

```
import math

# Constants for players
PLAYER_X = 'X'
PLAYER_O = 'O'
EMPTY = ' '

class TicTacToe:
    def __init__(self):
        self.board = [[EMPTY for _ in range(3)] for _ in range(3)]

    def print_board(self):
        for row in self.board:
            print('|'.join(row))
            print('-' * 5)

    def is_winner(self, player):
        # Check rows, columns, and diagonals
        for i in range(3):
            if all([self.board[i][j] == player for j in range(3)]) or all([self.board[j][i] == player
for j in range(3)]):
                return True
        if all([self.board[i][i] == player for i in range(3)]) or all([self.board[i][2 - i] == player
for i in range(3)]):
            return True
        return False

    def is_draw(self):
        return all([self.board[i][j] != EMPTY for i in range(3) for j in range(3)])

    def minimax(self, depth, is_maximizing):
        if self.is_winner(PLAYER_X):
            return -10 + depth  # X loses
        elif self.is_winner(PLAYER_O):
            return 10 - depth  # O wins
        elif self.is_draw():
            return 0  # Draw
```

```
        if is_maximizing:
            max_eval = -math.inf
            for i in range(3):
                for j in range(3):
                    if self.board[i][j] == EMPTY:
                        self.board[i][j] = PLAYER_O
                        eval = self.minimax(depth + 1, False)
                        self.board[i][j] = EMPTY
                        max_eval = max(max_eval, eval)
            return max_eval
        else:
            min_eval = math.inf
            for i in range(3):
                for j in range(3):
                    if self.board[i][j] == EMPTY:
                        self.board[i][j] = PLAYER_X
                        eval = self.minimax(depth + 1, True)
                        self.board[i][j] = EMPTY
                        min_eval = min(min_eval, eval)
            return min_eval

    def best_move(self):
        best_eval = -math.inf
        move = (-1, -1)
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == EMPTY:
                    self.board[i][j] = PLAYER_O
                    eval = self.minimax(0, False)
                    self.board[i][j] = EMPTY
                    if eval > best_eval:
                        best_eval = eval
                        move = (i, j)
        return move

    def play_game(self):
        current_player = PLAYER_X
        while True:
            if current_player == PLAYER_X:
```

```
            self.print_board()
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter column (0-2): "))
            if self.board[row][col] == EMPTY:
                self.board[row][col] = PLAYER_X
                if self.is_winner(PLAYER_X):
                    self.print_board()
                    print("Player X wins!")
                    break
                current_player = PLAYER_O
        else:
            print("Player O is making a move...")
            row, col = self.best_move()
            self.board[row][col] = PLAYER_O
            if self.is_winner(PLAYER_O):
                self.print_board()
                print("Player O wins!")
                break
            current_player = PLAYER_X

        if self.is_draw():
            self.print_board()
            print("It's a draw!")
            break

# Example usage
if __name__ == "__main__":
    game = TicTacToe()
    game.play_game()
```

Output Screenshot:

```
PS C:\Users\kushp\Documents\Prolog> python -u
 | |
-----
 | |
-----
 | |
-----
Enter row (0-2): 0
Enter column (0-2): 2
Player O is making a move...
 | |X
-----
 |O|
-----
 | |
-----
Enter row (0-2): 1
Enter column (0-2): 2
Player O is making a move...
 | |X
-----
 |O|X
-----
 | |O
-----
```

Practical No.10

Aim: Write a program in Prolog that will answer the question for the following facts.

Author
(name,address,age)
Publisher (name,address)
Book
(title,author,publisher)
a.What are the names of all authors?
b.What is the address of publisher abc?
c.What are the titles published by abc?

Program Code:

```
% Define facts

% Author(name, address, age)
author('Author One', 'Address One', 45).
author('Author Two', 'Address Two', 38).
author('Author Three', 'Address Three', 29).

% Publisher(name, address)
publisher('abc', 'Publisher Address One').
publisher('xyz', 'Publisher Address Two').

% Book(title, author, publisher)
book('Book One', 'Author One', 'abc').
book('Book Two', 'Author Two', 'abc').
book('Book Three', 'Author One', 'xyz').
book('Book Four', 'Author Three', 'abc').

% a. What are the names of all authors?
all_authors(Name) :-
    author(Name, _, _).

% b. What is the address of publisher abc?
publisher_address(Name, Address) :-
    publisher(Name, Address).
```

```
% c. What are the titles published by abc?
titles_published_by(Publisher, Title) :-
    book(Title, _, Publisher).

% Example Queries
% ?- all_authors(Name).
% ?- publisher_address('abc', Address).
% ?- titles_published_by('abc', Title).
```

Output Screenshot:

Practical No.11

Aim: Write a program in Prolog to find,

- **Member of a list**
- **The length of an input list.**
- **Concatenation of two**
- **Reverse of a list.**
- **Delete an item from list**

Program Code:

```
% Member of a list
member(X, [X|_]).                % X is a member if it is the head of the list.
member(X, [_|Tail]) :- member(X, Tail). % X is a member if it is in the tail.

% Length of a list
length([], 0).                   % The length of an empty list is 0.
length([_|Tail], Length) :- length(Tail, TailLength), Length is TailLength + 1. % Count the head and recurse.

% Concatenation of two lists
concat([], L, L).                % Concatenating an empty list with L results in L.
concat([Head|Tail], L, [Head|ResultTail]) :- concat(Tail, L, ResultTail). % Recur with head and tail.

% Reverse of a list
reverse_list([], []).            % The reverse of an empty list is an empty list.
reverse_list([Head|Tail], Reversed) :-
   reverse_list(Tail, ReversedTail),
   concat(ReversedTail, [Head], Reversed). % Recur and concatenate.

% Delete an item from a list
delete_item(X, [X|Tail], Tail).      % If X is the head, skip it.
delete_item(X, [Head|Tail], [Head|ResultTail]) :-
   delete_item(X, Tail, ResultTail). % Recur on tail.

% Example Queries
% ?- member(a, [a, b, c]).         % Should return true.
% ?- length([a, b, c], Length).    % Should return Length = 3.
% ?- concat([1, 2], [3, 4], Result). % Should return Result = [1, 2, 3, 4].
% ?- reverse_list([1, 2, 3], Reversed). % Should return Reversed = [3, 2, 1].
```

% ?- delete_item(b, [a, b, c], Result). % Should return Result = [a, c].

Output Screenshot:

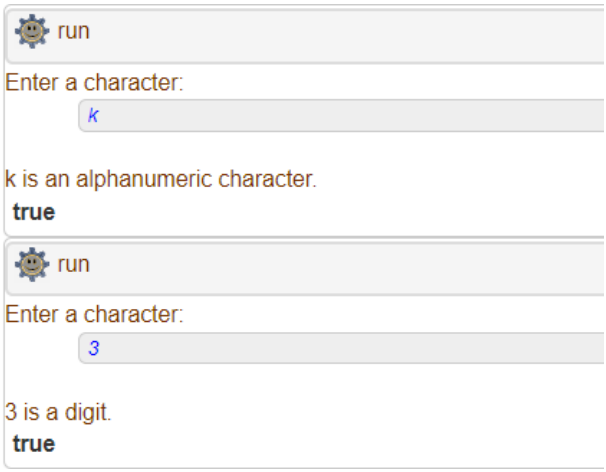
Practical No.12

Aim: Write a Program in Prolog for reading in a character and decide whether it is a digit or an alphanumeric character

Program Code:

```
% Check if a character is a digit
is_digit(Char) :-
   char_type(Char, digit).

% Check if a character is alphanumeric
is_alphanumeric(Char) :-
   char_type(Char, alnum).

% Main predicate to read a character and classify it
check_character :-
   write('Enter a character: '),     % Prompt the user
   read_line_to_string(user_input, InputString),  % Read a line of input
   (   string_length(InputString, Length),
      Length > 0 ->
      sub_string(InputString, 0, 1, _, FirstChar),  % Get the first character
      atom_string(CharAtom, FirstChar),            % Convert string to atom
      atom_chars(CharAtom, [Char])                % Convert atom to character
   ;
      write('No character entered.'), nl, fail
   ),
   nl,                          % New line for better output formatting
   (   is_digit(Char) ->
      write(Char), write(' is a digit.'), nl
   ;   is_alphanumeric(Char) ->
      write(Char), write(' is an alphanumeric character.'), nl
   ;   write(Char), write(' is neither a digit nor an alphanumeric character.'), nl
   ).

run :-
   check_character.
```

Output Screenshot:



Practical No.13

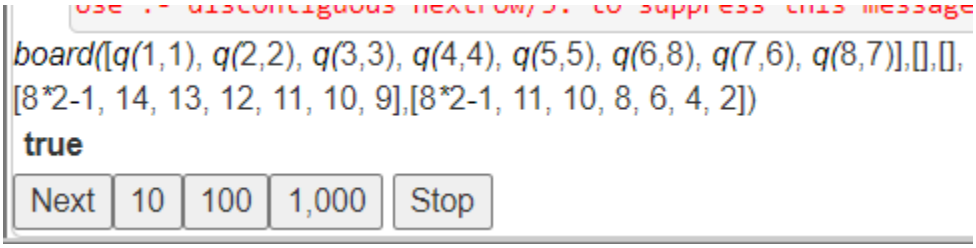
Aim:Write a program to solve N-Queens problems using Prolog.

Program Code:

```
domains
queen = q(integer, integer)
queens = queen*
freelist = integer*
board = board(queens, freelist, freelist, freelist, freelist)
predicates
placeN(integer, board, board)
place_queen(integer, board, board)
nqueens(integer)
makelist(integer, freelist)
find_remove(integer, freelist, freelist)
nextrow(integer, freelist, freelist)
clauses
nqueens(N) :-
makelist(N, L),
Diagonal = N * 2 - 1,
makelist(Diagonal, LL),
placeN(N, board([], L, L, LL, LL), Final),
write(Final).
placeN(0, Board, Board).
placeN(N, Board1, Result) :-
N > 0,
N1 = N - 1,
place_queen(N, Board1, Board2),
placeN(N1, Board2, Result).
place_queen(N, board(Queens, Rows, Columns, Diag1, Diag2),
board([q(N, C) | Queens],NewR, NewC, NewD1, NewD2)):-
nextrow(R, Rows, NewR),
find_remove(C, Columns, NewC),
D1 = N + C - R,
find_remove(D1, Diag1, NewD1),
D2 = R + C - 1,
find_remove(D2, Diag2, NewD2).
find_remove(X, [X | Rest], Rest).
find_remove(X, [Y | Rest], [Y | Tail]) :-
find_remove(X, Rest, Tail).
```

```
makelist(1, [1]).
makelist(N, [N | Ret]) :-

N > 1,
N1 = N - 1,
makelist(N1, Ret).
nextrow(Row, [Row | Rest], Rest).
```

Output Screenshot:



Practical No.14

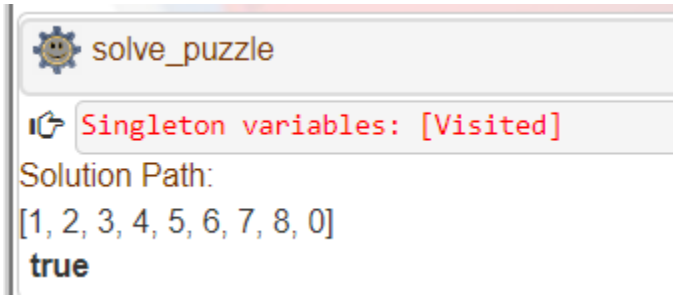
Aim: Write a program to solve 8 puzzle problem using Prolog

Program Code:

```
% Define the initial state of the puzzle
initial_state([1, 2, 3, 4, 5, 6, 7, 8, 0]).

% Define the goal state of the puzzle
goal_state([1, 2, 3, 4, 5, 6, 7, 8, 0]).

% Define the possible moves
move([A, B, C, D, E, F, G, H, 0], [A, B, C, D, E, F, H, 0, G]). % Move empty space up
move([A, B, C, D, E, F, 0, H, G], [A, B, C, D, E, F, G, H, 0]). % Move empty space down
move([A, B, C, 0, E, F, D, H, G], [A, B, C, E, 0, F, D, H, G]). % Move empty space left
move([A, B, C, E, F, 0, D, H, G], [A, B, C, E, F, G, D, H, 0]). % Move empty space right
move([A, B, 0, D, E, F, C, H, G], [A, B, F, D, E, 0, C, H, G]). % Move empty space left (in second row)
move([A, B, F, D, E, 0, C, H, G], [A, B, F, D, E, C, 0, H, G]). % Move empty space right (in second row)
move([0, B, C, D, E, F, A, H, G], [B, 0, C, D, E, F, A, H, G]). % Move empty space down (in first column)
move([B, C, 0, D, E, F, A, H, G], [B, C, F, D, E, 0, A, H, G]). % Move empty space down (in second column)
move([A, 0, C, D, E, F, B, H, G], [A, F, C, D, E, 0, B, H, G]). % Move empty space up (in second row)
move([A, B, C, 0, E, F, D, H, G], [A, B, C, E, 0, F, D, H, G]). % Move empty space down (in second row)

% Solve the puzzle using breadth-first search
solve(State, Path, Visited) :-
    goal_state(State),
    write('Solution Path: '), nl,
    print_path(Path), !.

solve(State, Path, Visited) :-
    findall(Next, (move(State, Next), \+ member(Next, Visited)), NextStates),
    append(Path, NextStates, NewPath),
    member(NewState, NextStates),
    solve(NewState, NewPath, [State | Visited]).

print_path([]).
```

```
print_path([H|T]) :-
    write(H), nl,
    print_path(T).

% To start solving the puzzle
solve_puzzle :-
    initial_state(InitialState),
    solve(InitialState, [InitialState], []).
```

Output Screenshot:



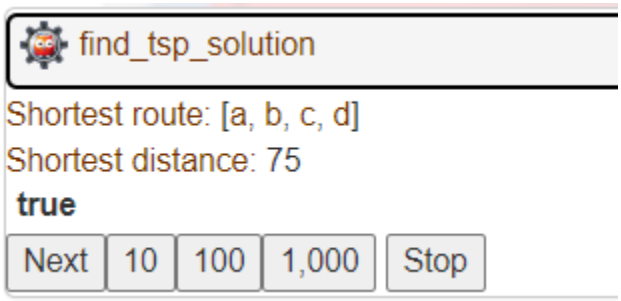
Practical No.15

Aim: Write a program to solve traveling salesman problems using Prolog.

Program Code:

```
% Define the distances between cities
distance(a, b, 10).
distance(a, c, 15).
distance(a, d, 20).
distance(b, c, 35).
distance(b, d, 25).
distance(c, d, 30).

% Calculate the total distance of a route
total_distance([], 0).
total_distance([_], 0).  % No distance when only one city
total_distance([City1, City2 | Rest], Distance) :-
    distance(City1, City2, D1),
    total_distance([City2 | Rest], D2),
    Distance is D1 + D2.

% Generate all permutations of a list
permutation([], []).
permutation(L, [H|P]) :-
    select(H, L, R),
    permutation(R, P).

% Solve the Traveling Salesman Problem
solve_tsp(Cities, ShortestRoute, ShortestDistance) :-
    permutation(Cities, Route),              % Generate all possible routes
    total_distance(Route, Distance),         % Calculate distance for the route
    \+ (total_distance(Route, D), D < Distance), % Ensure it's the shortest distance
    ShortestRoute = Route,
    ShortestDistance = Distance.

% To start solving the TSP
find_tsp_solution :-
    Cities = [a, b, c, d],                   % List of cities
    solve_tsp(Cities, ShortestRoute, ShortestDistance),
    write('Shortest route: '), write(ShortestRoute), nl,
    write('Shortest distance: '), write(ShortestDistance), nl.
```

Output Screenshot:



Practical No.16

Aim: Write a program to implement perceptron for AND gate

Program Code:

```
% Define the perceptron parameters
weights([1, 1]). % Weights for inputs x1 and x2
bias(1.5).       % Bias

% Activation function: Step function
activation(Sum, Output) :-
   (Sum >= 0 -> Output = 1; Output = 0).

% Compute the output of the perceptron
perceptron(X1, X2, Output) :-
   weights([W1, W2]),
   bias(Bias),
   % Calculate the weighted sum
   Sum is W1 * X1 + W2 * X2 - Bias,
   % Apply the activation function
   activation(Sum, Output).

% Test the perceptron with all combinations of inputs
test_perceptron :-
   forall(
      (between(0, 1, X1), between(0, 1, X2)),
      (perceptron(X1, X2, Output),
       format('Input: (~w, ~w), Output: ~w~n', [X1, X2, Output]))
   ).
```

Output Screenshot: