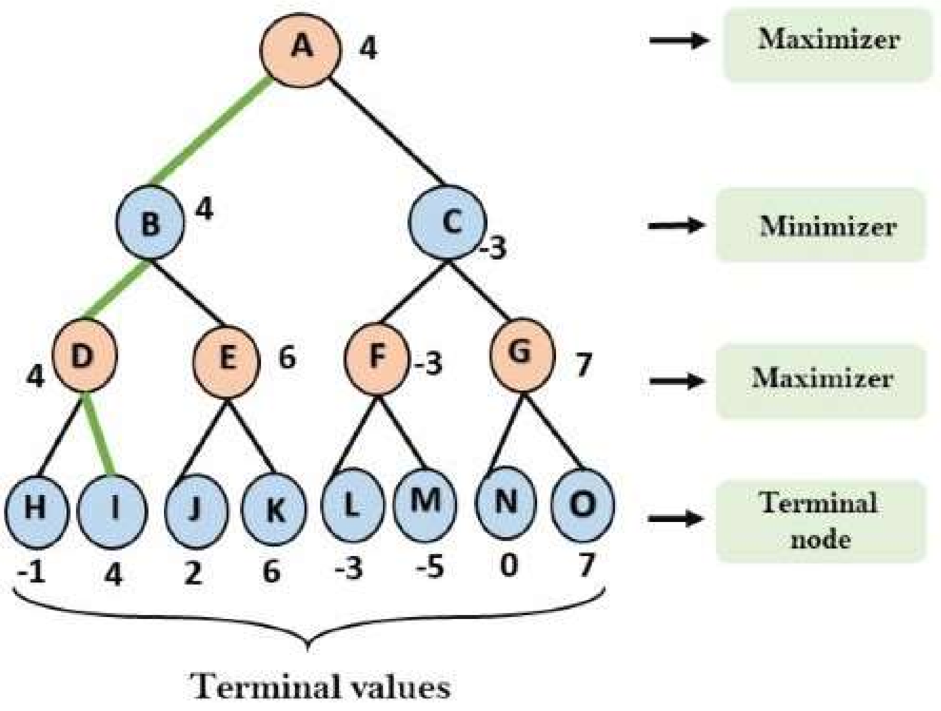
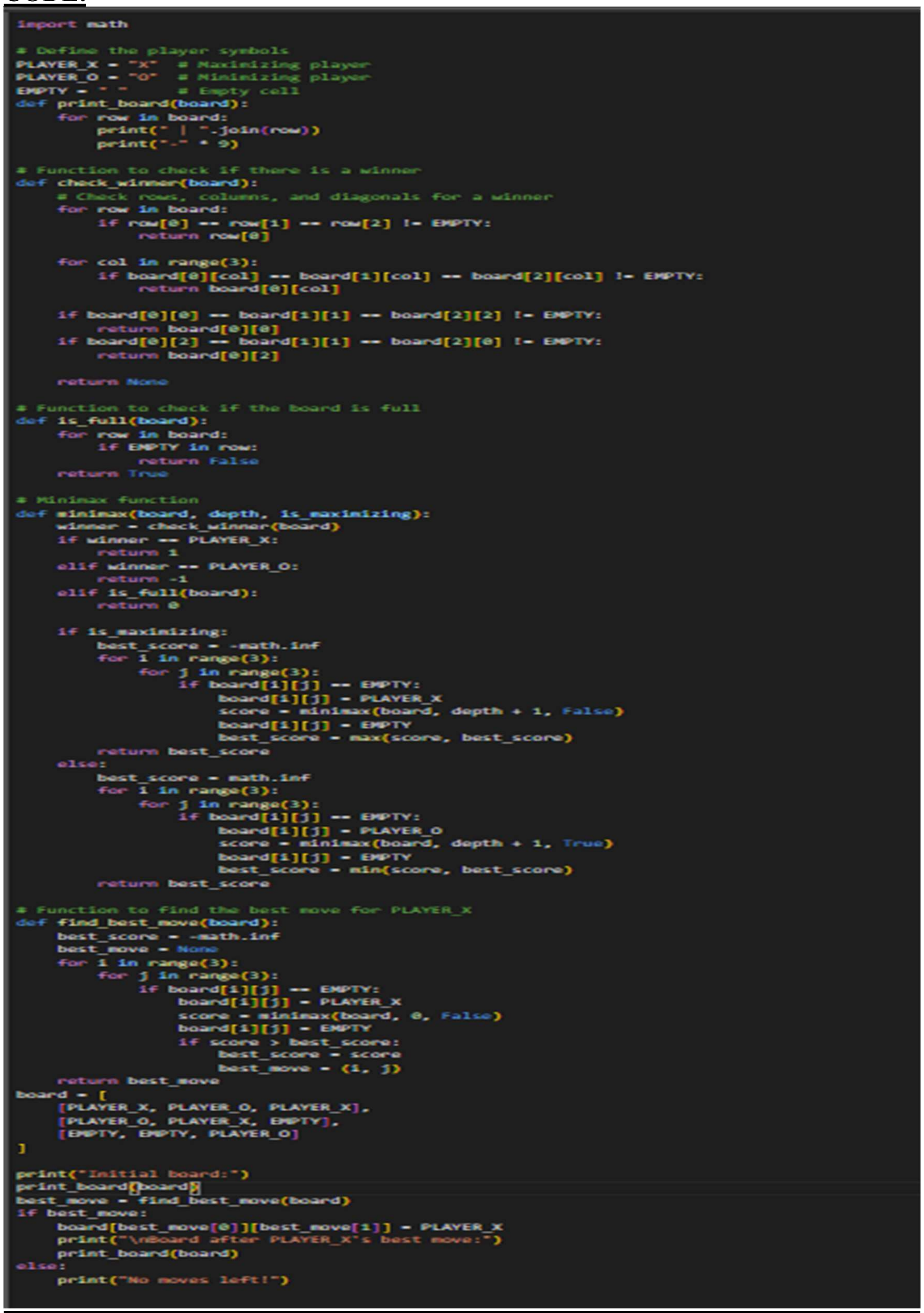
EX.NO: **DATE:**

MINIMAX ALGORITHM

- A simple example can be used to explain how the minimax algorithm works. We've included an example of a game-tree below, which represents a two-player game.
- There are two players in this scenario, one named Maximizer and the other named Minimizer.
- Maximizer will strive for the highest possible score, while Minimizer will strive for the lowest possible score.
- Because this algorithm uses DFS, we must go all the way through the leaves to reach the terminal nodes in this game-tree.
- The terminal values are given at the terminal node, so we'll compare them and retrace the tree till we reach the original state.

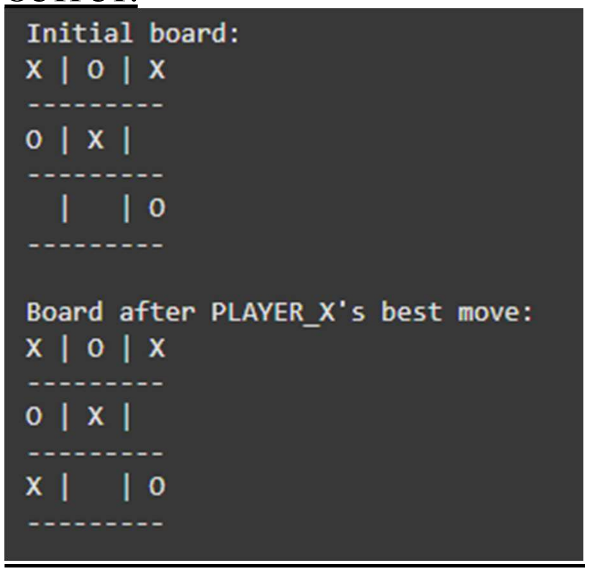


CODE:



```python
import math

# Define the player symbols
PLAYER_X = "X"  # Maximizing player
PLAYER_O = "O"  # Minimizing player
EMPTY = " "     # Empty cell
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

# Function to check if there is a winner
def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for row in board:
        if row[0] == row[1] == row[2] != EMPTY:
            return row[0]

    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != EMPTY:
            return board[0][col]

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

    return None

# Function to check if the board is full
def is_full(board):
    for row in board:
        if EMPTY in row:
            return False
    return True

# Minimax function
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner == PLAYER_X:
        return 1
    elif winner == PLAYER_O:
        return -1
    elif is_full(board):
        return 0

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

# Function to find the best move for PLAYER_X
def find_best_move(board):
    best_score = -math.inf
    best_move = None
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = PLAYER_X
                score = minimax(board, 0, False)
                board[i][j] = EMPTY
                if score > best_score:
                    best_score = score
                    best_move = (i, j)
    return best_move
board = [
    [PLAYER_X, PLAYER_O, PLAYER_X],
    [PLAYER_O, PLAYER_X, EMPTY],
    [EMPTY, EMPTY, PLAYER_O]
]

print("Initial board:")
print_board(board)
best_move = find_best_move(board)
if best_move:
    board[best_move[0]][best_move[1]] = PLAYER_X
    print("\nBoard after PLAYER_X's best move:")
    print_board(board)
else:
    print("No moves left!")
```

OUTPUT:



```
Initial board:
X | O | X
---------
O | X |
---------
  |   | O
---------

Board after PLAYER_X's best move:
X | O | X
---------
O | X |
---------
X |   | O
---------
```

RESULT:

Thus Program is Executed Successfully And Output is Verified.