Code:

```
def is_safe(board, row, col, n):
    for i in range(row):
        if board[i][col] == 1:
            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, -1, -1), range(col, n)):
        if board[i][j] == 1:
            return False
    return True

def print_solution(board):
    for row in board:
        print(" ".join(["Q" if col == 1 else "." for col in row]))
    print()

def solve_n_queens_backtracking(n):
    board = [[0 for _ in range(n)] for _ in range(n)]

    def backtrack(row):
        if row == n:
            print_solution(board)
            return True
        for col in range(n):
            if is_safe(board, row, col, n):
                board[row][col] = 1
                if backtrack(row + 1):
                    return True
                board[row][col] = 0
        return False
```

```
    if not backtrack(0):
        print("No solution exists.")

def solve_n_queens_branch_and_bound(n):
    def heuristic(board, row):
        return sum(board[i][row] for i in range(n))

    def branch_and_bound(row, board):
        if row == n:
            print_solution(board)
            return True
        min_col = None
        min_score = float('inf')
        for col in range(n):
            if is_safe(board, row, col, n):
                score = heuristic(board, col)
                if score < min_score:
                    min_score = score
                    min_col = col
        if min_col is not None:
            board[row][min_col] = 1
            if branch_and_bound(row + 1, board):
                return True
            board[row][min_col] = 0
        return False

    board = [[0 for _ in range(n)] for _ in range(n)]

    if not branch_and_bound(0, board):
        print("No solution exists.")
```

n = 8

print("Solutions using Backtracking:")

solve_n_queens_backtracking(n)

print("Solutions using Branch and Bound:")

solve_n_queens_branch_and_bound(n)

Output:

```
Solutions using Backtracking:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solutions using Branch and Bound:
No solution exists.
```

Time Complexity:

Backtracking:

In the worst case, the backtracking algorithm explores all possible configurations of queens on the board.

The time complexity of the backtracking algorithm is typically exponential, specifically O(N^N), where N is the size of the chessboard (number of rows and columns).

Branch and Bound:

The Branch and Bound method is more efficient than the simple backtracking approach because it uses heuristics to prioritize safe placements.

The time complexity of Branch and Bound can still be exponential in the worst case but is often much better in practice, especially for larger board sizes.

The exact time complexity can vary depending on the heuristic used. In the provided code, a simple heuristic is used to prioritize columns with fewer queens. In practice, good heuristics can significantly reduce the number of explored states, leading to improved performance.