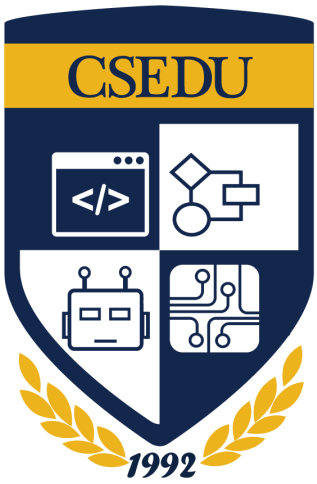


Department of Computer Science & Engineering
University of Dhaka

CSE-3203: Design and Analysis of Algorithms - II

Report 1 : N Queen

Submitted To:

Md. Tanvir Alam

Lecturer, Department of Computer Science and Engineering

Prepared by:

Farzana Tasnim (14)

Submission Date: January 7, 2026

Contents

Contents

1 Problem statement

Given an integer n, place n queens on an n × n chessboard such that no two queens attack each other. A queen can attack another queen if they are placed in the same row, the same column, or on the same diagonal.

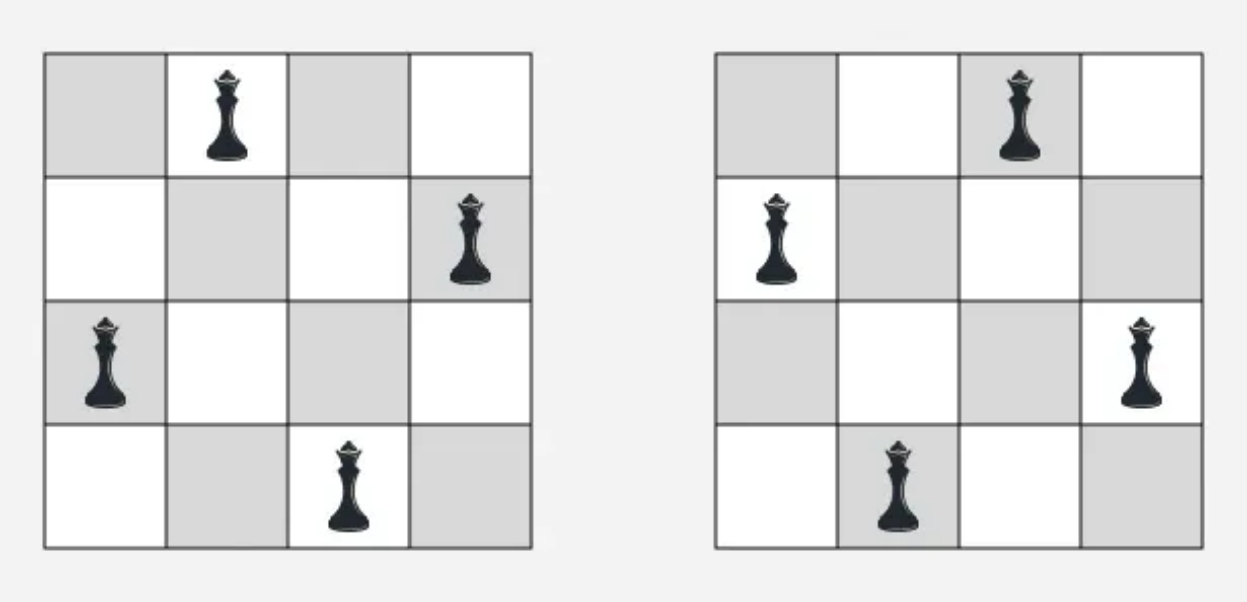


Figure 1: Possible solution for 4*4 chess board

1.1 Identifying a valid move

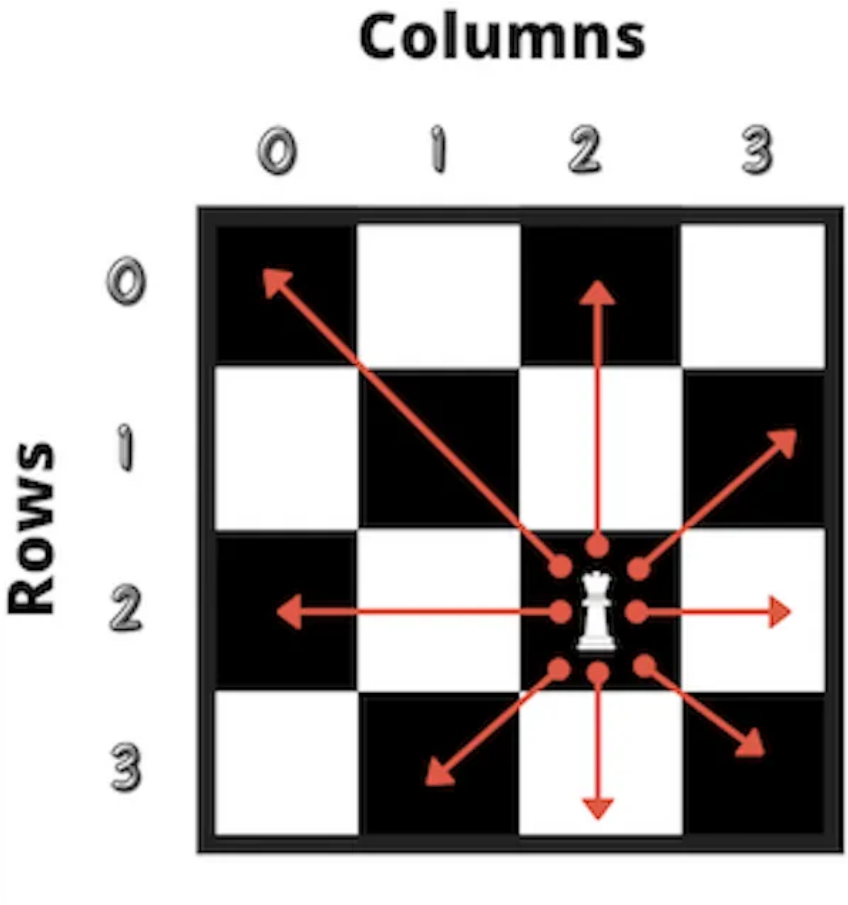


Figure 2: Valid move by queen

The queen can move horizontally, vertically and diagonally across the board.

2 Method 1 (Naive Brute Force)

2.1 Algorithm Description

This approach generates *all* possible board configurations and tests constraints only after complete generation.

Algorithm 1 Naive Generate-and-Test

```
1: procedure F_M1(i)
2:     if i = n^2 then
3:         if IS_VALID(M) then
4:             print M
5:         end if
6:         return
7:     end if
8:     r ← i ÷ n, c ← i mod n
9:     M[r][c] ← 1
10:    F_M1(i + 1)
11:    M[r][c] ← 0
12:    F_M1(i + 1)
13: end procedure
```

2.2 M1 Execution Pattern

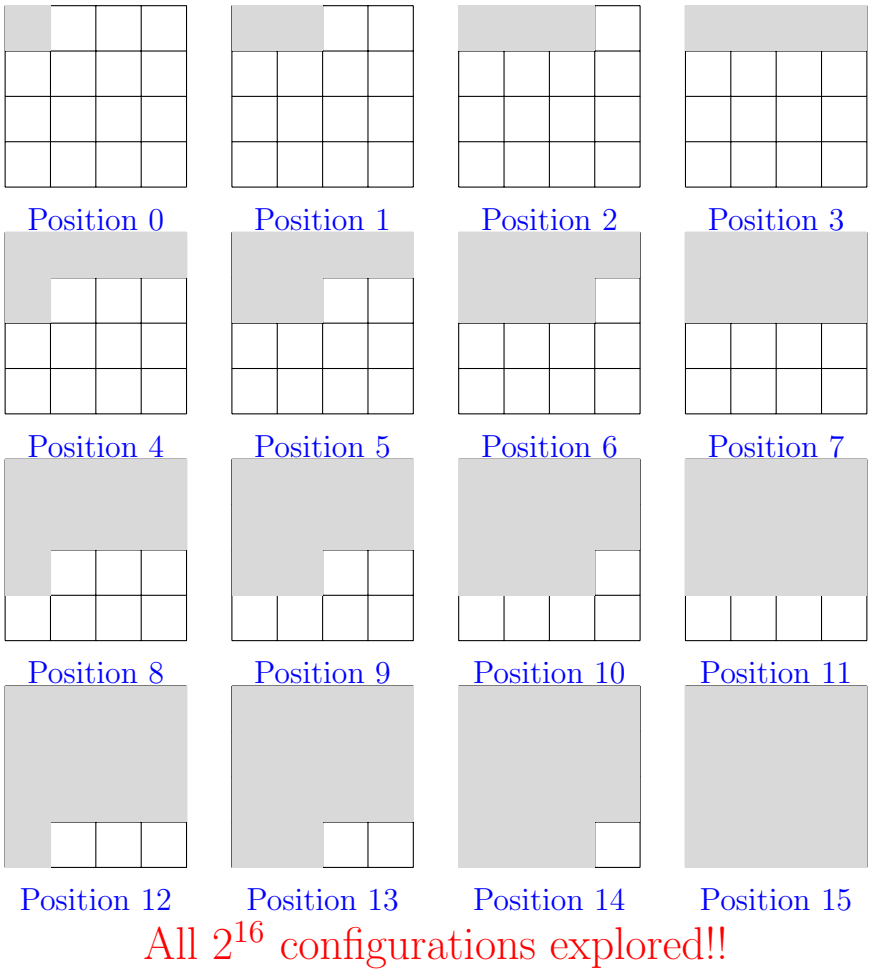


Figure 3: M1 explores in position order (0-15), making binary decisions at each cell.

2.3 Why This Approach Fails: Exponential Explosion

The fundamental flaw lies in the ***deferred constraint checking***. Consider the decision tree:

- At each of n^2 positions, we make a binary choice (place queen or not)
- Total configurations explored: 2^{n^2}
- For $n = 4$: $2^{16} = 65{,}536$ configurations
- For $n = 8$: $2^{64} \approx 1.8 \times 10^{19}$ configurations

Why does this happen? The algorithm lacks ***look-ahead capability***. **It continues exploring branches that are *guaranteed* to fail.**

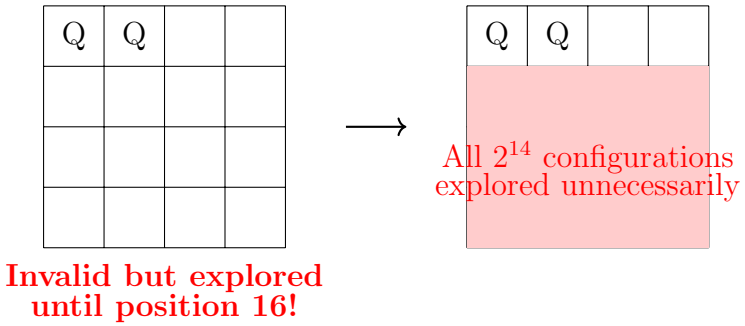


Figure 4: M1: Two queens in same row at positions 0 and 1. The algorithm explores all $2^{14} = 16{,}384$ remaining configurations despite inevitable failure.

Time Complexity: $O(2^{n^2} \cdot n^2)$ where n^2 is the constraint checking cost.

3 Method 2(Constraint-Aware Backtracking)

3.1 Algorithm Description

This approach checks constraints *immediately* after placing each queen.

Algorithm 2 Constraint-Aware Backtracking

```
1: procedure F_M2(i)
2:     if i = n^2 then
3:         if IS_VALID(M) then
4:             print M
5:         end if
6:         return
7:     end if
8:     r ← i ÷ n, c ← i mod n
9:     M[r][c] ← 1
10:    if not SAFE(M) then
11:        return                                  ▷ Prune immediately
12:    end if
13:    F_M2(i + 1)
14:    M[r][c] ← 0
15:    F_M2(i + 1)
16: end procedure
```

3.2 Why This Approach Performs Better: Early Pruning

The key improvement is *fail-fast semantics*. When a constraint violation is detected, the entire subtree is pruned.

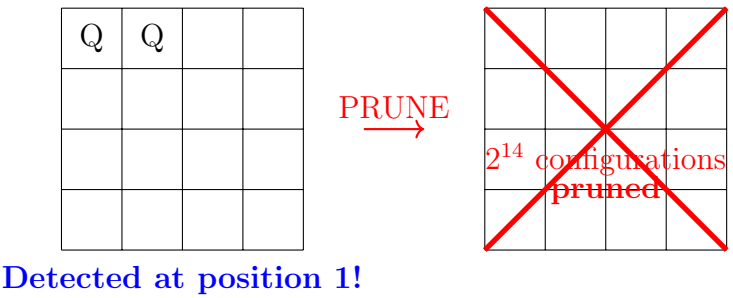


Figure 5: M2: Conflict detected at position 1, pruning 2^{14} configurations immediately.

Pruning Mathematics: When we detect a conflict at depth d, we prune 2^{n^2-d} configurations.

Time Complexity: Still $O(2^{n^2})$ worst case, but with *significantly reduced constant factors* due to pruning.

4 Method 3

4.1 Algorithm Description

This approach leverages the critical insight: ***exactly one queen per row***, combined with immediate validation.

Algorithm 3 Column-Optimized Backtracking with Early Validation

```
1: procedure F_M3(i)
2:     if i = n then
3:         print c                          ▷ Print column array solution
4:         return
5:     end if
6:     for j ← 0 to n − 1 do
7:         c[i] ← j                         ▷ Place queen in row i, column j
8:         if IS_VALID(i) then              ▷ Check if placement is safe
9:             F_M3(i + 1)
10:        end if
11:    end for
12: end procedure
```

Key Feature: The array $c[i]$ stores the column position of the queen in row i. This compact representation inherently prevents row conflicts.

4.2 Why This Approach is Optimal: Dual Optimization Strategy

This algorithm achieves optimality through *two synergistic mechanisms*:

Mechanism 1: Structural Search Space Reduction By using array $c[i]$ to represent "queen in row i at column $c[i]$", the algorithm encodes the constraint "one queen per row" into the data structure itself.

Mechanism 2: Incremental Validation The `is_valid(i)` check examines only the *current row* i against *previously placed queens* (rows 0 to $i-1$).

What does `is_valid(i)` check?

$$\text{Column conflict:} \quad c[i] = c[k] \text{ for any } k < i \tag{1}$$

$$\text{Diagonal conflict:} \quad |c[i] - c[k]| = |i - k| \text{ for any } k < i \tag{2}$$

Detailed step-by-step execution trace:

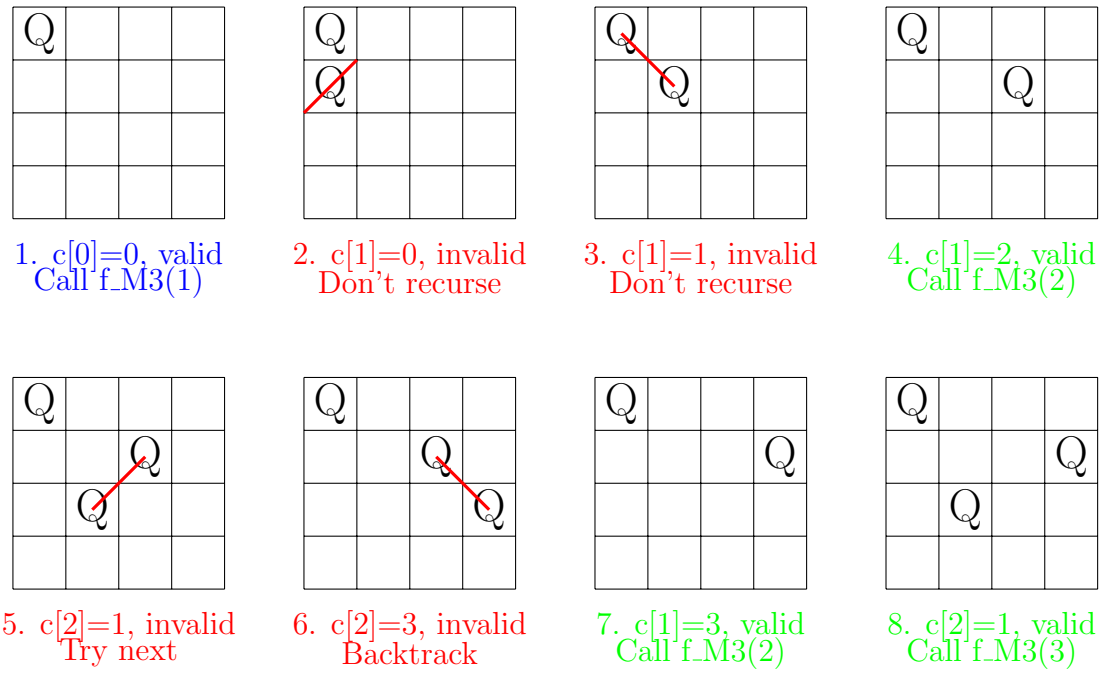
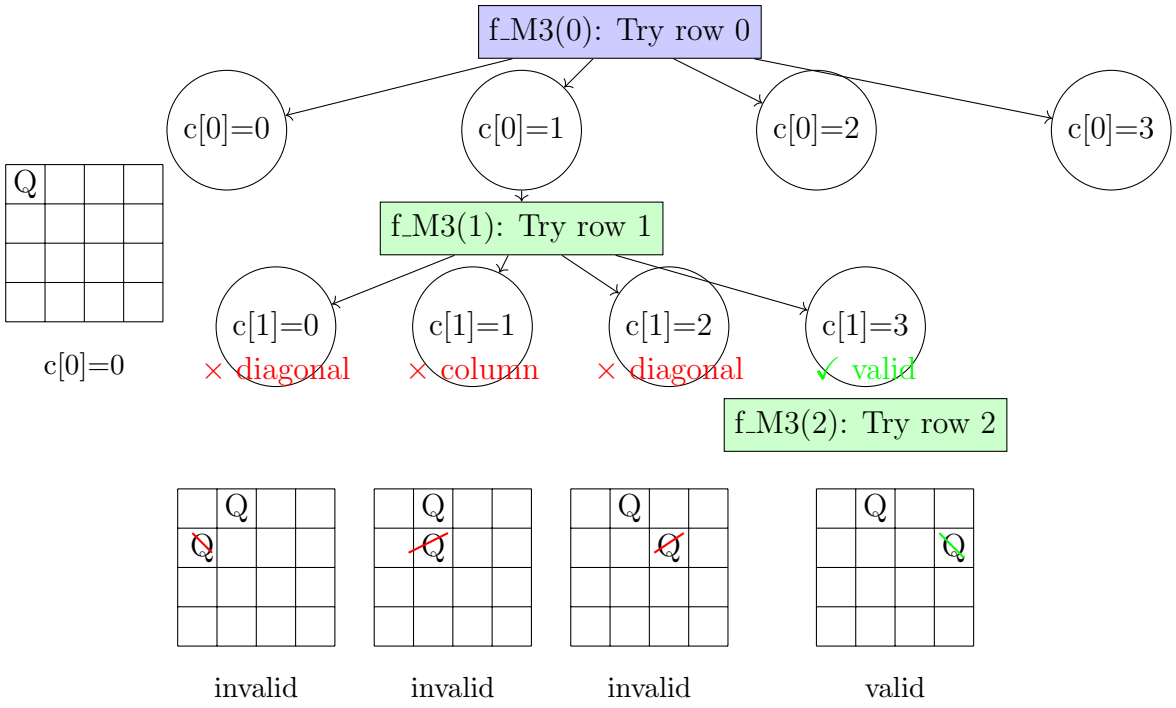


Figure 6: M3 recursive trace: The function tries each column in current row, validates immediately, and only recurses if valid. When all columns in row 1 are exhausted or invalid, it backtracks to row 0 and tries the next column. This is true backtracking: only valid partial solutions spawn deeper recursion.

Recursive Execution Tree (following actual recursion):



5 Concrete Execution Trace: 4×4 Example

To understand *why* these algorithms behave differently, we must trace their actual execution paths.

5.1 M1 Execution Pattern

M1 processes the board in position order (0, 1, 2, ..., 15), making binary decisions at each position. After placing queens at positions 0 and 1 (row conflict), it still explores all 2^{14} remaining configurations before checking validity.

5.2 M2 Execution Pattern: Early Pruning in Action

M2 uses the same position-by-position approach as M1, but checks safety immediately after each placement. When safe() returns False at position 1 (after queen at 0), it immediately backtracks, pruning 2^{14} configurations. However, the algorithm structure doesn't inherently prevent trying to place queens in the same row.

5.3 M3 Execution Pattern: Recursive Row-by-Row Placement

It contains- one queen per row. Significantly reducing unnecessary attempt and time.
Critical observation: Unlike M2 which explores positions 0,1,2,... sequentially, M3 works

depth-first by row:

- Places queen in row 0, column $j \rightarrow$ if valid $\rightarrow$ recurse to row 1
- Row 1 tries all columns $\rightarrow$ if valid $\rightarrow$ recurse to row 2
- Invalid placements don't recurse (immediate pruning)
- Returns only when row is complete or all columns tried

Why M3's execution is fundamentally different from M2:

Aspect	**M2**	**M3**
Iteration order	Position 0,1,2,...,15	Row 0,1,2,3
Decision at each step	Place queen or not (binary)	Which column (n-way)
Can place 2 queens in same row?	Yes (checked later)	No (impossible)

Table 1: Fundamental differences in execution between M2 and M3

6 Fundamental Principles Demonstrated

6.1 The Cost of Late Validation

M1 demonstrates that *deferring constraint checking until completion* leads to exhaustive enumeration.

6.2 The Power of Early Pruning

M2 shows that *incremental constraint checking* enables pruning. However, the algorithm still explores a space fundamentally too large.

6.3 The Importance of Problem Encoding

M3 reveals that one queen per row and then check which column should we place the queen- significantly improves time.

7 Conclusion

This analysis demonstrates four fundamental algorithmic principles:

1. **Fail-fast semantics:** Early detection prevents wasted computation
2. **Constraint encoding:** Building constraints into data structures eliminates entire problem classes

3. **Validation granularity:** Checking only relevant state reduces overhead

For the 4×4 board, M1 examines all configurations, M2 prunes significantly but still works in exponential space, and M3 examines minimum configurations that eliminates unnecessary attempts with early pruning bringing the effective count far lower.

The N-Queens problem thus serves as a paradigmatic example: algorithmic efficiency emerges not from faster computation, but from *avoiding unnecessary computation* through intelligent problem structuring, efficient data representation, and strategic validation placement.