Topic Introduction

Welcome back, PrepLetter reader! Today, we're diving deep into **backtracking for constraint satisfaction**. If you've ever tried to solve a puzzle where each move affects all future possibilities, you've encountered backtracking—even if you didn't realize it.

What is Backtracking?

Backtracking is a powerful algorithmic technique for solving problems incrementally, one step at a time, and removing ("backtracking") those solutions that fail to satisfy constraints. Imagine navigating a maze: you try a path; if you hit a dead end, you step back and try another. Backtracking is essentially a depth-first search (DFS) with constraint checks.

How does it work?

- Build the solution one piece at a time.
- At each step, check if the current path is valid.
- If it's not, stop and "backtrack" to try a different path.
- If it is, keep going.
- Repeat until you find a solution (or all solutions).

When is backtracking useful?

- When you need to generate all valid arrangements, combinations, or permutations under constraints.
- When brute force would be too slow, but you can prune invalid partial solutions early.

Example (not from our main problems):

Suppose you want to generate all subsets of `{1,2,3}`. You can use backtracking by deciding at each step whether to include the next number. If you only want subsets with even sums, you can prune as soon as the sum is odd.

Today's Problems:

- N-Queens
- N-Queens II
- Sudoku Solver

Why these? All three are *classic backtracking* problems that involve constraint satisfaction. N-Queens asks you to place queens on a chessboard so they don't attack each other. N-Queens II simplifies this to just counting solutions. Sudoku Solver fills a 9x9 grid, making sure every row, column, and box contains each digit exactly once.

Let's get to it!

Problem 1: N-Queens ([LeetCode link](https://leetcode.com/problems/n-queens/))

Rephrased Problem Statement:

Given an integer `n`, place `n` queens on an `n x n` chessboard so that no two queens attack each other. Return all distinct board arrangements. Each arrangement should be represented as an array of strings, with `Q` for a queen and `.` for an empty space.

Example:

Input: `n = 4`

Output:

```
[
 [".Q..",
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",
  "Q...",
  "...Q",
  ".Q.."]
]
```

How do we solve this?

This is a classic backtracking challenge. We need to:

- Place one queen per row (or column), moving row by row.
- For each row, try each column and see if placing a queen there is safe:
    - No queen in the same column.
    - No queen on the same diagonal (two types of diagonals).

If it's safe, move to the next row. If not, try the next column. If we reach row `n`, we have a valid solution.

Try this one by hand first!

Try `n = 4` with pen and paper. Can you find both solutions?

Additional test case:

Input: `n = 1`

Output: `[["Q"]]`

Brute-force Approach:

Try every way to place `n` queens on the board, then check if any pair attacks each other. This is O(n^n) time—way too slow.

Optimal Approach: Backtracking with Pruning

- At each row, try each column.
- Use sets to track used columns and diagonals.
- If a queen can be placed, recurse to the next row.
- If we reach the end, record the solution.

Python Solution

```
def solveNQueens(n):
    def backtrack(row, cols, diag1, diag2, board):
        if row == n:
            # Found a valid board; copy and save it.
            result.append([''.join(r) for r in board])
            return
        for col in range(n):
            if col in cols or (row - col) in diag1 or (row + col) in diag2:
                continue  # Conflict, skip
            # Place queen
            board[row][col] = 'Q'
            cols.add(col)
            diag1.add(row - col)
            diag2.add(row + col)
            backtrack(row + 1, cols, diag1, diag2, board)
            # Remove queen (backtrack)
            board[row][col] = '.'
            cols.remove(col)
            diag1.remove(row - col)
            diag2.remove(row + col)

    result = []
    empty_board = [['.'] * n for _ in range(n)]
    backtrack(0, set(), set(), set(), empty_board)
    return result
```

Time and Space Complexity:

- **Time:** O(n!)—since the search space is pruned heavily, but in the worst case, it’s still factorial.
- **Space:** O(n^2 * # of solutions) for storing all boards, plus O(n) extra for the sets and board per recursion stack.

Code Breakdown

- `backtrack(row, cols, diag1, diag2, board)`: Recursive function for each row.
  - `cols`: Columns with queens.
  - `diag1`: "Main" diagonals with queens (row - col).
  - `diag2`: "Anti" diagonals with queens (row + col).
  - `board`: Current board.
- For each column, check if it's safe.
- If safe, place queen, recurse to next row.

- If row == n, we've placed all queens—save the board.
- After recursion, remove the queen (backtrack).

Trace Example (`n=4`):

- Place queen at (0,0). Next row.
- Row 1: Can't place at (1,0) (same column), or (1,1) (diagonal). Try (1,2): safe.
- Row 2: Try (2,0), (2,1), (2,2): blocked, (2,3): safe.
- Row 3: Only (3,1) is safe.
- All queens placed! Save solution.

Try tracing `n=4` yourself to see the steps.

Extra Test Case:

Input: `n = 2`

Output: `[]` (No solution)

Take a moment to solve this on your own before jumping into the code!

Problem 2: N-Queens II ([LeetCode link](https://leetcode.com/problems/n-queens-ii/))

How is this different?

Instead of returning all solutions, we just need to **count** how many valid arrangements exist for a given `n`.

Example:

Input: `n = 4`

Output: `2` (There are two valid arrangements)

Another test case:

Input: `n = 5`

Output: `10`

Approach:

- The constraints and logic are identical to Problem 1.
- Instead of saving each board, we just increment a counter every time we reach `row == n`.

Brute-force:

Same as before—try everything, but prune with constraints.

Optimal Approach with Backtracking:

- Use sets or bitmasks to track attacks (for extra optimization).
- At each row, try each column. If safe, recurse.
- If all rows are filled, increment the count.

Pseudocode

```
function totalNQueens(n):
    count = 0
    backtrack(row=0, cols=set(), diag1=set(), diag2=set())
    return count

function backtrack(row, cols, diag1, diag2):
    if row == n:
        count += 1
        return
    for col in 0..n-1:
        if col in cols or (row-col) in diag1 or (row+col) in diag2:
            continue
        cols.add(col)
        diag1.add(row-col)
        diag2.add(row+col)
        backtrack(row+1, cols, diag1, diag2)
        cols.remove(col)
        diag1.remove(row-col)
        diag2.remove(row+col)
```

Step-by-step Example (n=4):

- Place queen on row 0, col 1. Proceed.
- Row 1: Try each column, skipping conflicts.
- If all rows are placed, increment count.
- Backtrack and try different placements.

Space and Time Complexity:

- **Time:** O(n!) still, for same reasons.
- **Space:** O(n) for recursion and sets.

Dry-run test case:

Try `n = 3` on your own. (Should return `0`.)

Trace:

With `n = 1`, the function should return `1` (one way to place a queen on a 1x1 board).

Problem 3: Sudoku Solver ([LeetCode link](https://leetcode.com/problems/sudoku-solver/))

What's more challenging here?

Instead of placing queens, we’re filling a 9x9 Sudoku board with digits 1-9 so that each row, column, and 3x3 box contains no duplicates. The input is a partially filled grid.

Example:

Input:

```
board = [
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
```

Output: The board is modified in-place to a valid solution.

Another test case:
Try a board with only one empty cell.

Approach:

- For each empty cell, try digits 1-9.
- Check if placing the digit violates row, column, or box constraints.
- If valid, recurse to the next empty cell.
- If no digit works, backtrack.

Pseudocode

```
function solveSudoku(board):
    find all empty cells
    backtrack(idx=0)

function backtrack(idx):
    if idx == len(empty_cells):
        return True  // all filled!
    (row, col) = empty_cells[idx]
    for digit in '1'..'9':
        if digit is valid at (row, col):
            place digit
            if backtrack(idx + 1):
                return True
            remove digit
    return False
```

Key constraints to check:

- Row: no duplicate digit in the row.
- Column: no duplicate in the column.
- Box: no duplicate in the 3x3 box.

Time and Space Complexity:

- **Time:** In the worst case, O(9^(n)), where n is the number of empty cells. But in practice, the pruning is powerful.
- **Space:** O(1) extra (the board is modified in-place), plus recursion stack of depth up to number of empty cells.

Dry-run test case:
Try a board with only one cell empty, say at (0,2), and see if you can fill it uniquely.

Reflect:
Consider: Could you speed this up with better constraint tracking (like bitmasks or keeping track of available digits per row/col/box)?

Summary and Next Steps

These three problems are grouped together because they beautifully illustrate **backtracking with constraint satisfaction**: trying possibilities, pruning early, and ensuring global constraints are never violated.

Key patterns:

- Place/fill one item at a time.
- For each choice, check if constraints are satisfied before going deeper.
- Backtrack if you hit an invalid state.
- Use sets or arrays to efficiently track constraints.

Common mistakes:

- Forgetting to backtrack (remove your choice before moving up recursion).
- Not checking *all* relevant constraints.
- Inefficient constraint checking—always look for ways to prune early!

Action List

- Solve all three problems on your own, including coding up the full solution for Problem 3.
- For N-Queens II, try using bitwise operations to track columns and diagonals for extra speed.
- For Sudoku Solver, experiment with constraint propagation (e.g., forward checking).
- Compare your code to solutions on LeetCode—focus on style, efficiency, and edge cases.
- If you get stuck, review the step-by-step logic above, and don't hesitate to step through with pen and paper.
- Reflect on how backtracking applies to other constraint satisfaction puzzles!

Keep practicing—each attempt sharpens your intuition and problem-solving skills. Happy coding!