Backtracking / Recursion

Pattern: Backtracking (DFS with State Restoration)

Also known as "Recursive DFS with Pruning"

How to Recognize

- Problem asks for **all possible combinations, permutations, or valid configurations**.
- You need to **build solutions step-by-step**, and at each step, try different choices.
- Solutions must satisfy **constraints** (e.g., well-formed parentheses, unique numbers).
- The search space is **exponential**, but pruning helps reduce it.
- Often involves **generating subsets, permutations, or solving puzzles like Sudoku/Queens**.

Step-by-Step Thinking Process (The Recipe)

1. **Define the base case**: When do we have a complete solution?
2. **Define the recursive structure**: What are the choices at each step?
3. **Make a choice** → add to current path
4. **Recurse** on remaining options
5. **Undo the choice** (backtrack) → remove from path
6. Use **pruning** to skip invalid paths early (e.g., too many open brackets)

   This is essentially **DFS with state restoration** — you explore one path, then backtrack to try others.

Common Pitfalls & Edge Cases

- Forgetting to **undo the choice** (i.e., pop from path), leading to incorrect results.
- Not handling **base cases correctly** (e.g., empty input).
- Generating duplicates when not allowed (use `set` or ordering constraints).
- Infinite recursion due to missing termination condition.

---

1. Permutations

Problem Summary

Given an array `nums` of distinct integers, return all possible permutations.

Pattern

- **Backtracking**
- Generate all full-length permutations using DFS + state restoration.

```python
def permute(nums):
    result = []

    def backtrack(path):
        # Base case: if path has same length as nums,
        # we have a complete permutation
        if len(path) == len(nums):
            result.append(path[:])  # Make a copy to avoid reference issues
            return

        # Try each number not yet used in the current path
        for num in nums:
            if num not in path:  # Avoid duplicates (though nums are distinct)
                path.append(num)  # Choose
                backtrack(path)   # Explore
                path.pop()        # Unchoose (backtrack)

    backtrack([])
    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: nums = [1,2,3]
    # Expected Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

    output = permute([1, 2, 3])

    print("Output:", output)
```

Walkthrough: `nums = [1,2,3]`

- Start with `path = []`
- Pick 1 → `path = [1]`, recurse
    - Pick 2 → `path = [1,2]`, recurse
        - Pick 3 → `path = [1,2,3]` → base case → append `[1,2,3]`

        * Pop `3`, now back to `[1,2]`
    - Pick `3` → `path = [1,3]`, recurse
        * Pick `2` → `path = [1,3,2]` → append → pop
    - Pop `2`, pop `3`

- Then pick `2` as first element, etc.
- All 6 permutations generated.

Complexity

- **Time**: O(N! × N) — N! permutations, each takes O(N) to copy.
- **Space**: O(N) — recursion depth + path storage.

---

2. Subsets

Problem Summary

Given an integer array `nums`, return all possible subsets (the power set).

Pattern

- **Backtracking + Power Set Generation**
- At each element, decide whether to include or exclude it.

```python
def subsets(nums):
    result = []

    def backtrack(start_idx, path):
        # Base case: every path is a valid subset
        result.append(path[:])  # Add current subset

        # Explore further elements starting from `start_idx`
        for i in range(start_idx, len(nums)):
            path.append(nums[i])      # Include nums[i]
            backtrack(i + 1, path)   # Recurse with next index
            path.pop()               # Backtrack: exclude nums[i]

    backtrack(0, [])
    return result
```

```python
# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: nums = [1,2,3]
    # Expected Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

    output = subsets([1, 2, 3])

    print("Output:", output)
```

Walkthrough: `nums = [1,2,3]`

- Start: `path = []`, `start_idx = 0`
- Add `[]` to result
- Include `1`: `path = [1]`, recurse with `start_idx=1`
  - Add `[1]`
  - Include `2`: `path=[1,2]`, recurse `start_idx=2`
    - Add `[1,2]`
    - Include `3`: `path=[1,2,3]`, add → pop
    - Backtrack → pop `2`
  - Backtrack → pop `1`
- Include `2` at root level → build `[2]`, `[2,3]`
- Include `3` → build `[3]`
- All subsets collected.

Complexity

- **Time**: O(2^N × N) — 2^N subsets, each up to N elements to copy.
- **Space**: O(2^N × N) — storing all subsets (output size), plus O(N) recursion stack.

---

3. Letter Combinations of a Phone Number

Problem Summary

Given a string of digits (2–9), return all possible letter combinations that the number could represent.

Pattern

- **Backtracking with Mapping**
- Each digit maps to letters; generate all combos by choosing one letter per digit.

```python
def letterCombinations(digits):
    if not digits:
        return []

    # Mapping from digit to letters
    phone_map = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    }

    result = []

    def backtrack(index, path):
        # Base case: reached end of digits
        if index == len(digits):
            result.append(''.join(path))  # Convert list to string
            return

        # Get current digit and its corresponding letters
        current_digit = digits[index]
        letters = phone_map[current_digit]

        # Try each letter
        for letter in letters:
            path.append(letter)           # Choose
            backtrack(index + 1, path)   # Explore next digit
            path.pop()                   # Backtrack

    backtrack(0, [])
    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: digits = "23"
    # Expected Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]

    output = letterCombinations("23")
```

```
    print("Output:", output)
```

Walkthrough: `digits = "23"`

- `phone_map['2'] = 'abc'`, `phone_map['3'] = 'def'`
- Start: `path = []`, `index = 0`
- Try `'a'` → `path = ['a']`, go to index 1
  - Try `'d'` → `path = ['a','d']` → add `"ad"` → pop
  - Try `'e'` → `"ae"` → pop
  - Try `'f'` → `"af"` → pop
- Backtrack → try `'b'` → `"bd", "be", "bf"`
- Then `'c'` → `"cd", "ce", "cf"`

Complexity

- **Time**: O(3^N × 4^M × K) where N = digits with 3 letters, M = digits with 4 letters, K = average combo length.
- In worst case: O(4^N × N)
- **Space**: O(4^N × N) — output size, plus O(N) recursion depth.

---

4. Generate Parentheses

Problem Summary

Given `n`, generate all combinations of well-formed parentheses.

Pattern

- **Backtracking with Constraints**
- Track open and close counts; only add `)` if `close < open`.

```python
def generateParenthesis(n):
    result = []

    def backtrack(current, open_count, close_count):
        # Base case: if current length is 2*n, we have a valid combo
        if len(current) == 2 * n:
            result.append(current)
            return

        # Add '(' if we haven't used n yet
        if open_count < n:
            backtrack(current + '(', open_count + 1, close_count)

        # Add ')' only if there are unmatched '('
        if close_count < open_count:
            backtrack(current + ')', open_count, close_count + 1)

    backtrack("", 0, 0)
    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: n = 3
    # Expected Output: ["((()))","(()())","(())()","()(())","()()()"]

    output = generateParenthesis(3)

    print("Output:", output)
```

Walkthrough: `n = 3`

- Start: `current = ""`, `open=0`, `close=0`
- Can add `'('` → `"( "` → `open=1`
  - Add `'('` → `"(("` → `open=2`
    - Add `'('` → `"((("` → `open=3`
      - Now can only add `)` until balanced
      - `((()))` → done
    - Then add `)` → `"((()"` → `close=1`
      - Continue...

- Eventually builds all 5 valid sequences.

Complexity

- **Time**: O(4^n / √n) — Catalan number growth
- **Space**: O(4^n / √n) — output size, plus O(n) recursion depth.

---

5. N-Queens

Problem Summary

Place `n` queens on an `n×n` chessboard so no two queens attack each other.

Pattern

- **Backtracking with Pruning**
- Use sets to track occupied columns, diagonals.

```
def solveNQueens(n):
    result = []

    # Track which columns and diagonals are occupied
    cols = set()
    diag1 = set()  # r - c
    diag2 = set()  # r + c

    def backtrack(row, board):
        # Base case: placed all queens
        if row == n:
            result.append(board[:])  # Append copy of board
            return

        for col in range(n):
            # Check if placing queen at (row, col) is safe
            if col in cols or (row - col) in diag1 or (row + col) in diag2:
                continue

            # Place queen
            cols.add(col)
```

```python
            diag1.add(row - col)
            diag2.add(row + col)
            board.append("." * col + "Q" + "." * (n - col - 1))

            # Recurse to next row
            backtrack(row + 1, board)

            # Backtrack: remove queen
            cols.remove(col)
            diag1.remove(row - col)
            diag2.remove(row + col)
            board.pop()

    backtrack(0, [])
    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: n = 4
    # Expected Output: [[".Q..","...Q","Q...","..Q."],
    # ["..Q.","Q...","...Q",".Q.."]]

    output = solveNQueens(4)

    for sol in output:
        print(sol)
```

Walkthrough: `n = 4`

- Row 0: try col 1 → place Q → mark col 1, diag1 (0-1=-1), diag2 (0+1=1)
- Row 1: can't use col 1, -1, or 1 → try col 3
- Row 2: try col 0 → check conflict? (2-0=2, 2+0=2) → not in sets → OK
- Row 3: try col 2 → check: col 2 ok, 3-2=1 → not in diag1, 3+2=5 → not in diag2 → OK
- Board: `["..Q.", "Q...", "...Q", ".Q.."]` → valid

Complexity

- **Time**: O(N!) — worst-case exponential, but pruning reduces it significantly.
- **Space**: O(N^2) — for storing board and sets.

6. Sudoku Solver

Problem Summary

Solve a partially filled 9×9 Sudoku puzzle.

Pattern

- **Backtracking with Pruning**
- Use sets to track used values in rows, cols, and boxes.

```
def solveSudoku(board):
    # Track used values in rows, cols, and 3x3 boxes
    rows = [set() for _ in range(9)]
    cols = [set() for _ in range(9)]
    boxes = [set() for _ in range(9)]

    # Initialize the sets with existing numbers
    for r in range(9):
        for c in range(9):
            if board[r][c] != '.':
                val = board[r][c]
                rows[r].add(val)
                cols[c].add(val)
                box_idx = (r // 3) * 3 + (c // 3)
                boxes[box_idx].add(val)

    def get_box_idx(r, c):
        return (r // 3) * 3 + (c // 3)

    def backtrack(r, c):
        # If we've filled the whole board
        if r == 9:
            return True

        # Move to next row
        if c == 9:
            return backtrack(r + 1, 0)

        # Skip pre-filled cells
```

```python
        if board[r][c] != '.':
            return backtrack(r, c + 1)

        # Try digits 1-9
        for digit in map(str, range(1, 10)):
            box_idx = get_box_idx(r, c)

            # Check if digit is valid
            if (
                digit not in rows[r]
                and digit not in cols[c]
                and digit not in boxes[box_idx]
            ):
                # Place digit
                board[r][c] = digit
                rows[r].add(digit)
                cols[c].add(digit)
                boxes[box_idx].add(digit)

                # Recurse
                if backtrack(r, c + 1):
                    return True

                # Backtrack: remove digit
                board[r][c] = '.'
                rows[r].remove(digit)
                cols[c].remove(digit)
                boxes[box_idx].remove(digit)

        return False  # No valid digit found

    backtrack(0, 0)


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: board = [
    #   ["5","3",".",".","7",".",".",".","."],
    #   ["6",".",".","1","9","5",".",".","."],
    #   [".","9","8",".",".",".",".","6","."],
    #   ["8",".",".",".","6",".",".",".","3"],
    #   ["4",".",".","8",".","3",".",".","1"],
```

```python
#   ["7",".",".",".","2",".",".",".","6"],
#   [".","6",".",".",".",".","2","8","."],
#   [".",".",".","4","1","9",".",".","5"],
#   [".",".",".",".","8",".",".","7","9"]
# ]
# Output: Solved board (same format)

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"]
]

solveSudoku(board)

# Print solved board
for row in board:
    print(row)
```

Walkthrough: Small example

- Find first empty cell (e.g., top-left corner).
- Try digits 1–9, check row/col/box.
- Place valid digit, recurse.
- If stuck later, backtrack and try another digit.

Complexity

- **Time**: O(9^k) where k = number of empty cells (but heavily pruned).
- **Space**: O(1) extra (only 9 sets of size 9), plus O(81) board storage.

Array / Math Manipulation

Pattern: Array Manipulation + Greedy Strategy

Also known as "In-Place Transformation with Index Tracking"

How to Recognize

- Problem involves rearranging elements in an array **in-place**.
- You need to **find the next lexicographically greater permutation** or **transform matrix structure**.
- Often uses **greedy swaps** after identifying a pivot point where order breaks.
- Common in permutations, rotations, and sequence reordering.

Step-by-Step Thinking Process (The Recipe)

1. **Identify the pivot**: Find the first index `i` from right where `nums[i] < nums[i+1]`.
2. **Find successor**: Find smallest element to the right of `i` that is larger than `nums[i]`.
3. **Swap**: Exchange `nums[i]` with that successor.
4. **Reverse suffix**: Reverse the subarray after `i` to get the smallest possible arrangement.
5. Use **two pointers** for efficient reversal.

   This is essentially **lexicographic next permutation generation** via greedy optimization.

Common Pitfalls & Edge Cases

- Forgetting to reverse the suffix → results in non-minimal permutation.
- Not handling edge case: descending array (already largest) → return sorted (smallest).
- Off-by-one errors when finding pivot or successor.
- Not using `swap` correctly — must be value-based, not index-based.

---

7. Next Permutation

Problem Summary

Given an array of integers, rearrange it into the **next lexicographically greater permutation**. If no such permutation exists, rearrange it into the **lowest possible order** (sorted ascending).

Pattern

- **Array Manipulation + Greedy Swap Strategy**

```
def nextPermutation(nums):
    # Step 1: Find the pivot - first index i from right such that
    # nums[i] < nums[i+1]
    pivot = -1
    n = len(nums)

    for i in range(n - 2, -1, -1):
        if nums[i] < nums[i + 1]:
            pivot = i
            break

    # If no pivot found, array is in descending order →reverse to get smallest
    if pivot == -1:
        nums.reverse()
        return

    # Step 2: Find the smallest element to the
    # right of pivot that is > nums[pivot]
    # Since right side is decreasing, scan from
    #right to find first such element
    successor = -1
    for i in range(n - 1, pivot, -1):
        if nums[i] > nums[pivot]:
            successor = i
            break

    # Step 3: Swap pivot and successor
    nums[pivot], nums[successor] = nums[successor], nums[pivot]

    # Step 4: Reverse the suffix (from pivot+1 to end) to make it minimal
    left = pivot + 1
    right = n - 1
    while left < right:
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1


# ---- Official LeetCode Example ----
```

```python
if __name__ == "__main__":
    # Example Input: nums = [1,2,3]
    # Expected Output: [1,3,2]

    nums = [1, 2, 3]
    nextPermutation(nums)
    print("Output:", nums)  # Output: [1, 3, 2]

    # Another example: nums = [3,2,1]
    # Expected Output: [1,2,3]
    nums = [3, 2, 1]
    nextPermutation(nums)
    print("Output:", nums)  # Output: [1, 2, 3]
```

Walkthrough: `nums = [1,2,3]`

- Scan from right: `2 < 3` → pivot = 1 (`index=1`)
- Right of index 1: `[3]`, first element > 2 → successor = 2 (`index=2`)
- Swap `nums[1]` and `nums[2]`: `[1,3,2]`
- Reverse suffix starting at index 2 → only one element → unchanged
- Result: `[1,3,2]`

Complexity

- **Time**: O(N) — single pass forward and backward.
- **Space**: O(1) — in-place modification.

---

8. Spiral Matrix

Problem Summary

Given an `m x n` matrix, return all elements in spiral order.

Pattern

- **Simulation / Matrix Traversal**
- Use boundaries (`top`, `bottom`, `left`, `right`) to simulate spiral movement.

```python
def spiralOrder(matrix):
    if not matrix or not matrix[0]:
        return []

    result = []
    top, bottom = 0, len(matrix) - 1
    left, right = 0, len(matrix[0]) - 1

    while top <= bottom and left <= right:
        # Traverse from left to right along top row
        for col in range(left, right + 1):
            result.append(matrix[top][col])
        top += 1  # shrink top boundary

        # Traverse from top to bottom along right column
        for row in range(top, bottom + 1):
            result.append(matrix[row][right])
        right -= 1  # shrink right boundary

        # Check if we still have rows to traverse
        if top <= bottom:
            # Traverse from right to left along bottom row
            for col in range(right, left - 1, -1):
                result.append(matrix[bottom][col])
            bottom -= 1  # shrink bottom boundary

        # Check if we still have columns to traverse
        if left <= right:
            # Traverse from bottom to top along left column
            for row in range(bottom, top - 1, -1):
                result.append(matrix[row][left])
            left += 1  # shrink left boundary

    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
```

```python
# Example Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
# Expected Output: [1,2,3,6,9,8,7,4,5]

matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

output = spiralOrder(matrix)
print("Output:", output)
```

Walkthrough: `matrix = [[1,2,3],[4,5,6],[7,8,9]]`

- **Round 1**:
  - Left→Right: `[1,2,3]` → top becomes 1
  - Top→Bottom: `[6,9]` → right becomes 1
  - Bottom→Left: `[8,7]` → bottom becomes 0
  - Bottom→Top: `[4]` → left becomes 1
- **Round 2**:
  - Now `top=1`, `bottom=0` → condition fails → exit
- Final: `[1,2,3,6,9,8,7,4,5]`

Complexity

- **Time**: O(m × n) — each cell visited once.
- **Space**: O(1) extra (excluding output).

---

9. Rotate Image

Problem Summary

Given an `n x n` matrix, rotate it **90 degrees clockwise** in-place.

Pattern

- **Matrix Transformation (Transpose + Reverse)**
- Rotate = Transpose + Flip vertically

```
def rotate(matrix):
    n = len(matrix)

    # Step 1: Transpose the matrix (swap matrix[i][j] with matrix[j][i])
    for i in range(n):
        for j in range(i + 1, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

    # Step 2: Reverse each row (flip horizontally)
    for i in range(n):
        left, right = 0, n - 1
        while left < right:
            matrix[i][left],matrix[i][right]=matrix[i][right], matrix[i][left]
            left += 1
            right -= 1


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
    # Expected Output: [[7,4,1],[8,5,2],[9,6,3]]

    matrix = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]

    rotate(matrix)

    for row in matrix:
        print(row)
```

Walkthrough: `matrix = [[1,2,3],[4,5,6],[7,8,9]]`

- **Transpose**:

```
[[1,4,7],
 [2,5,8],
 [3,6,9]]
```

- **Reverse each row**:

```
[[7,4,1],
 [8,5,2],
 [9,6,3]]
```

Complexity

- **Time**: $O(N^2)$ — two passes over matrix.
- **Space**: $O(1)$ — in-place.

---

10. Set Matrix Zeroes

Problem Summary

Given an `m x n` matrix, if an element is `0`, set its entire row and column to `0`.

Pattern

- **In-Place Matrix Update with Markers**
- Use first row/column as markers to avoid extra space.

```python
def setZeroes(matrix):
    m, n = len(matrix), len(matrix[0])

    # Check if first row or column has zero
    first_row_has_zero = any(matrix[0][j] == 0 for j in range(n))
    first_col_has_zero = any(matrix[i][0] == 0 for i in range(m))

    # Use first row and column as storage
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][j] == 0:
                matrix[i][0] = 0  # mark row i
                matrix[0][j] = 0  # mark col j
```

```python
    # Set zeros based on markers
    for i in range(1, m):
        if matrix[i][0] == 0:
            for j in range(n):
                matrix[i][j] = 0

    for j in range(1, n):
        if matrix[0][j] == 0:
            for i in range(m):
                matrix[i][j] = 0

    # Handle first row
    if first_row_has_zero:
        for j in range(n):
            matrix[0][j] = 0

    # Handle first column
    if first_col_has_zero:
        for i in range(m):
            matrix[i][0] = 0


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
    # Expected Output: [[1,0,1],[0,0,0],[1,0,1]]

    matrix = [
        [1, 1, 1],
        [1, 0, 1],
        [1, 1, 1]
    ]

    setZeroes(matrix)

    for row in matrix:
        print(row)
```

Walkthrough: `matrix = [[1,1,1],[1,0,1],[1,1,1]]`

- First row has no zero → `first_row_has_zero = False`
- First col has no zero → `first_col_has_zero = False`

- Scan inner matrix: `matrix[1][1] == 0` → set `matrix[1][0] = 0`, `matrix[0][1] = 0`
- Then:
  - Row 1: `matrix[1][0] == 0` → set entire row 1 to 0
  - Col 1: `matrix[0][1] == 0` → set entire col 1 to 0
- Final: row 1 and col 1 are zeroed.

Complexity

- **Time**: O(m × n)
- **Space**: O(1) — only using first row/col as flags.

Bit Manipulation

Pattern: Bitwise Operations & Binary Arithmetic

Also known as "Low-Level Binary Manipulation"

How to Recognize

- Problems involve **binary representation**, **bit flipping**, **XOR tricks**, or **masking**.
- Often require understanding of **two's complement**, **bit shifts**, and **bit-level logic**.
- Common in: counting set bits, finding unique numbers, simulating binary addition, reversing bits.
- Look for clues like "no extra space", "O(1) time", or "find single number".

Step-by-Step Thinking Process (The Recipe)

1. **Understand bit patterns**: Know how numbers are stored (binary).
2. **Use bitwise operators**:
   - `&` (AND): check if bit is set
   - `|` (OR): set a bit
   - `^` (XOR): toggle bits, find differences
   - `~` (NOT): invert bits
   - `<<`, `>>`: shift left/right (multiply/divide by 2)
3. **Apply common tricks**:
   - `n & (n-1)` → clears the lowest set bit

- `n & (-n)` → isolates the lowest set bit
- XOR all elements → cancel out pairs

4. **Simulate operations manually** when needed (e.g., binary addition).

Common Pitfalls & Edge Cases

- Forgetting that Python integers are **unbounded** — no overflow, but can affect logic.
- Misusing `~` without masking (e.g., `~x` gives negative in two's complement).
- Not handling negative numbers correctly in bit shifts.
- Using `>>` on signed integers — may sign-extend in some languages (not an issue in Python, but important to know).

---

11. Add Binary

Problem Summary

Given two binary strings `a` and `b`, return their sum as a binary string.

Pattern

- **Binary Addition Simulation**
- Simulate manual binary addition with carry.

```python
def addBinary(a, b):
    result = []
    carry = 0
    i, j = len(a) - 1, len(b) - 1

    # Process digits from right to left
    while i >= 0 or j >= 0 or carry:
        # Get current bits (0 if index out of range)
        bit_a = int(a[i]) if i >= 0 else 0
        bit_b = int(b[j]) if j >= 0 else 0

        # Sum of bits + carry
        total = bit_a + bit_b + carry

        # Append the least significant bit
        result.append(str(total % 2))
```

```python
        # Update carry
        carry = total // 2

        # Move pointers left
        i -= 1
        j -= 1

    # Reverse since we built result backwards
    return ''.join(reversed(result))


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: a = "11", b = "1"
    # Expected Output: "100"

    output = addBinary("11", "1")
    print("Output:", output)  # Output: "100"
```

Walkthrough: `a = "11"`, `b = "1"`

- Start: `i=1`, `j=0`, `carry=0`
- Iteration 1: `bit_a=1`, `bit_b=1`, `total=2` → append `0`, `carry=1`
- Iteration 2: `bit_a=1`, `bit_b=0`, `total=1+0+1=2` → append `0`, `carry=1`
- Iteration 3: `i=-1`, `j=-1`, but `carry=1` → append `1`, `carry=0`
- Result: `['0','0','1']` → reversed → `"100"`

Complexity

- **Time**: O(max(M, N)) — length of longer string.
- **Space**: O(max(M, N)) — result string.

---

12. Counting Bits

Problem Summary

For every number `i` from `0` to `n`, count the number of `1`s in its binary representation.

Pattern

- **Dynamic Programming + Bit Manipulation**
- Use recurrence: `dp[i] = dp[i >> 1] + (i & 1)`

```python
def countBits(n):
    dp = [0] * (n + 1)

    for i in range(1, n + 1):
        # Number of 1s in i = number of 1s in i//2 (i >> 1)
        # + whether last bit is 1
        dp[i] = dp[i >> 1] + (i & 1)

    return dp


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: n = 2
    # Expected Output: [0,1,1]

    output = countBits(2)
    print("Output:", output)  # Output: [0, 1, 1]

    # Another example: n = 5 → [0,1,1,2,1,2]
    output = countBits(5)
    print("Output:", output)  # Output: [0, 1, 1, 2, 1, 2]
```

Walkthrough: `n = 5`

- `dp[0] = 0`
- `dp[1] = dp[0] + 1 = 1`
- `dp[2] = dp[1] + 0 = 1`
- `dp[3] = dp[1] + 1 = 2`
- `dp[4] = dp[2] + 0 = 1`
- `dp[5] = dp[2] + 1 = 2`

Insight: `i >> 1` is equivalent to `i // 2`. The lower bits remain same except shifted.

Complexity

- **Time**: O(N)
- **Space**: O(N) — output array

---

13. Number of 1 Bits

Problem Summary

Given a 32-bit unsigned integer, return the number of `1` bits it has (Hamming weight).

Pattern

- **Brian Kernighan's Algorithm (Efficient Bit Counting)**

```
def hammingWeight(n):
    count = 0
    while n:
        # This clears the lowest set bit: n & (n-1)
        n &= n - 1
        count += 1
    return count


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: n = 11 (binary: 1011)
    # Expected Output: 3

    output = hammingWeight(11)
    print("Output:", output)  # Output: 3
```

Walkthrough: `n = 11` → `1011`

- `n = 1011`, `n-1 = 1010`
- `n & (n-1) = 1010` → count = 1
- `n = 1010`, `n-1 = 1001` → `n & (n-1) = 1000` → count = 2
- `n = 1000`, `n-1 = 0111` → `n & (n-1) = 0000` → count = 3
- Exit loop → return 3

  Why? Each iteration removes one `1` bit.

Complexity

- **Time**: O(k), where k = number of `1` bits (much better than O(32)).
- **Space**: O(1)

---

14. Single Number

Problem Summary

Given an array where every element appears twice except one, find the single number.

Pattern

- **XOR Trick**
- `a ^ a = 0`, `a ^ 0 = a` → duplicates cancel out.

```
def singleNumber(nums):
    result = 0
    for num in nums:
        result ^= num
    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: nums = [2,2,1]
    # Expected Output: 1

    output = singleNumber([2, 2, 1])
    print("Output:", output)  # Output: 1
```

Walkthrough: `nums = [2,2,1]`

- `result = 0`
- `0 ^ 2 = 2`
- `2 ^ 2 = 0`
- `0 ^ 1 = 1`
- Return `1`

  All duplicates cancel; only the unique number remains.

Complexity

- **Time**: O(N)
- **Space**: O(1)

---

15. Missing Number

Problem Summary

Given an array of `n` distinct numbers from `0` to `n`, find the missing one.

Pattern

- **XOR Trick / Summation Formula**
- Two approaches: XOR or arithmetic sum.

```python
def missingNumber(nums):
    # Option 1: XOR trick
    n = len(nums)
    result = n  # Start with n (the missing number could be n)

    for i in range(n):
        result ^= i ^ nums[i]

    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: nums = [3,0,1]
    # Expected Output: 2

    output = missingNumber([3, 0, 1])
    print("Output:", output)  # Output: 2
```

Walkthrough: `nums = [3,0,1], n=3`

- Start: `result = 3`
- Loop:
  - `i=0`: `result ^= 0 ^ 3 = 3 ^ 3 = 0`
  - `i=1`: `result ^= 1 ^ 0 = 0 ^ 1 = 1`
  - `i=2`: `result ^= 2 ^ 1 = 1 ^ 2 = 3`
- Return `3`? Wait — this doesn't match!

Oops — let's fix the logic.

Actually, here's the correct version using XOR:

```
def missingNumber(nums):
    result = 0
    n = len(nums)

    # XOR all indices (0 to n) and all nums
    for i in range(n):
        result ^= i ^ nums[i]

    # Final XOR with n (since we're missing one from 0..n)
    result ^= n
    return result
```

Or simpler: **Sum approach**

```
def missingNumber(nums):
    n = len(nums)
    expected_sum = n * (n + 1) // 2
    actual_sum = sum(nums)
    return expected_sum - actual_sum
```

Better version:

```
def missingNumber(nums):
    n = len(nums)
    expected_sum = n * (n + 1) // 2
    return expected_sum - sum(nums)


# ---- Official LeetCode Example ----
```

```python
if __name__ == "__main__":
    # Example Input: nums = [3,0,1]
    # Expected Output: 2

    output = missingNumber([3, 0, 1])
    print("Output:", output)  # Output: 2
```

Walkthrough: `nums = [3,0,1]`

- `n = 3`
- `expected_sum = 3*4//2 = 6`
- `actual_sum = 3+0+1 = 4`
- `6 - 4 = 2`

Complexity

- **Time**: O(N)
- **Space**: O(1)

---

16. Reverse Bits

Problem Summary

Reverse the bits of a 32-bit unsigned integer.

Pattern

- **Bit Manipulation (Extract & Build)**

```python
def reverseBits(n):
    result = 0
    for _ in range(32):
        # Shift result left to make room
        result <<= 1
        # Add the least significant bit of n
        result |= n & 1
        # Shift n right to get next bit
        n >>= 1
```

```python
    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: n = 43261596 (binary: 00000010100101000001111010011100)
    # Expected Output: 964176192 (binary: 00111001011110000010100101000000)

    output = reverseBits(43261596)
    print("Output:", output)  # Output: 964176192
```

Walkthrough: `n = 43261596`

- Extract bits one by one from right → build result from left.
- After 32 iterations, you've reversed all bits.

Complexity

- **Time**: O(32) = O(1)
- **Space**: O(1)

Binary Search / Two Pointers

Pattern: Binary Search on Answer (or Value)

Also known as "Search Space Optimization"

How to Recognize

- Problem asks for **minimum/maximum value** satisfying a condition.
- You can **check feasibility** of a candidate answer in O(log N) or O(N).
- The search space is **monotonic**: if `x` works, then all values x might work (or vice versa).
- Common in: finding minimum time, maximum capacity, kth element, duplicate detection.

Step-by-Step Thinking Process (The Recipe)

1. **Define search space**: Low = min possible, High = max possible.
2. **Define feasibility function**: Can we achieve this value?
3. **Binary search loop**:
   - Mid = (low + high) // 2
   - If feasible → try smaller (`high = mid`)
   - Else → try larger (`low = mid + 1`)
4. Return `low` (or `high`) — the first valid value.

   This pattern avoids brute force by turning a linear search into logarithmic.

Common Pitfalls & Edge Cases

- Off-by-one errors in boundaries (`low < high` vs `low <= high`).
- Not handling edge cases like empty input or single element.
- Feasibility function not properly defined (e.g., missing constraints).
- Integer overflow in `mid = (low + high) // 2` → use `mid = low + (high - low) // 2`.

---

17. Find the Duplicate Number

Problem Summary

Given an array of `n + 1` integers where each integer is between `1` and `n`, find the **one duplicate number**. Do not modify the array and use only O(1) extra space.

Pattern

- **Binary Search on Value** OR **Floyd's Cycle Detection**
- We'll focus on **Binary Search on Value** here.

```
def findDuplicate(nums):
    # Binary search on the value range [1, n]
    left, right = 1, len(nums) - 1

    while left < right:
        mid = (left + right) // 2
```

```
        # Count how many numbers are <= mid
        count = sum(1 for num in nums if num <= mid)

        # If count > mid, then there are more numbers than available slots
        # → duplicate must be in [left, mid]
        if count > mid:
            right = mid
        else:
            left = mid + 1

    return left


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: nums = [1,3,4,2,2]
    # Expected Output: 2

    output = findDuplicate([1, 3, 4, 2, 2])
    print("Output:", output)  # Output: 2
```

Walkthrough: `nums = [1,3,4,2,2]`

- `left=1`, `right=4`
- `mid=2`: count of numbers 2 → `[1,2,2]` → count=3 > 2 → duplicate in `[1,2]`
- `right=2`
- `left=1`, `right=2`, `mid=1`: count 1 → `[1]` → count=1 → not >1 → `left=2`
- Now `left==right=2` → return 2

  Insight: If more than `k` numbers `k`, then at least one duplicate exists among them.

Complexity

- **Time**: O(N log N) — binary search over N values, each check takes O(N)
- **Space**: O(1)

Absolutely! You're right — I **did** cover `Random Pick with Weight` and `Valid Sudoku` in the final chunk, but let me now **re-verify and fully deliver both** using the **exact same structured format** you requested.

We'll treat them as a **final mini-chunk (2 problems)**, with:

- Problem name hyperlinked
- Summary
- Pattern(s)
- Editorial-style Python solution with **inline comments**
- Official LeetCode sample input/output in `if __name__ == "__main__":` block
- Step-by-step walkthrough
- Time & space complexity

Probability / Prefix Sum

Pattern: Prefix Sum + Binary Search

Also known as "Weighted Random Selection"

How to Recognize

- Problem asks for **random selection based on weights**.
- Each index has a probability proportional to its weight.
- You need to **avoid O(n) per query** → use **prefix sum + binary search**.
- Often used in simulation, game mechanics, or sampling.

Step-by-Step Thinking Process (The Recipe)

1. **Build prefix sum array**: cumulative weights.
2. **Generate random number** between 0 and total weight.
3. **Binary search** for the first index where prefix sum random number.
4. Return that index.

   This turns O(n) per query into O(log n) after O(n) preprocessing.

Common Pitfalls & Edge Cases

- Not including `0` in prefix sum → off-by-one errors.
- Using `random.randint(1, total)` instead of `[0, total)` → wrong bounds.
- Not handling zero weights properly.
- Binary search logic error (e.g., `left <= right` vs `left < right`).

---

18. Random Pick with Weight

Problem Summary

Given an array of weights, return a random index such that the probability of picking index `i` is proportional to `weights[i]`.

Pattern

- **Prefix Sum + Binary Search**

```
import random
from bisect import bisect_left

class Solution:
    def __init__(self, w):
        # Step 1: Build prefix sum array
        self.prefix_sum = []
        total = 0
        for weight in w:
            total += weight
            self.prefix_sum.append(total)

        # Total weight for random range
        self.total_weight = total

    def pickIndex(self):
        # Step 2: Generate random number in [0, total_weight)
        rand_num = random.randint(0, self.total_weight - 1)

        # Step 3: Binary search for first index where prefix_sum >= rand_num
        # bisect_left returns leftmost position to insert rand_num
        return bisect_left(self.prefix_sum, rand_num)
```

```python
# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input:
    # obj = Solution([1, 3])
    # obj.pickIndex() # Returns 0 or 1 with probabilities 0.25 and 0.75
    # obj.pickIndex()
    # obj.pickIndex()
    # obj.pickIndex()
    # obj.pickIndex()

    obj = Solution([1, 3])

    print("Picking 5 times:")
    for _ in range(5):
        idx = obj.pickIndex()
        print(f"Index: {idx}")
```

Walkthrough: `w = [1, 3]`

- `prefix_sum = [1, 4]`, `total_weight = 4`
- Random number from `[0, 3]` → say `rand_num = 2`
- `bisect_left([1,4], 2)` → finds first index where value 2 → index `1`
- Return `1` → correct (probability 3/4)

  If `rand_num = 0` or `1` → return `0`; if `2` or `3` → return `1`.

Complexity

- **Constructor**: O(N) — build prefix sum
- **pickIndex()**: O(log N) — binary search
- **Space**: O(N) — prefix sum array

Hashing / Validation

Pattern: Hash Set Validation (Row, Col, Box)

Also known as "Grid Constraint Checking"

How to Recognize

- Problem involves validating a grid (e.g., Sudoku).
- Must check that no row, column, or subgrid contains duplicate numbers.
- Use sets to track seen values per group.
- Often uses `set()` + coordinate mapping.

Step-by-Step Thinking Process (The Recipe)

1. Loop through each cell `(r, c)`.
2. Skip if empty (`.`).
3. Check if value already exists in:
   - `row_set[r]`
   - `col_set[c]`
   - `box_set[box_idx]`
4. If yes → invalid.
5. Else → add to all three sets.
6. Return `True` if all valid.

   Key trick: `box_idx = r // 3 * 3 + c // 3`

Common Pitfalls & Edge Cases

- Forgetting to skip `'.'` cells.
- Miscomputing `box_idx` (e.g., using `r//3 + c//3` instead of `r//3*3 + c//3`).
- Not reusing sets across rows/columns → incorrect validation.
- Not handling empty board correctly.

---

19. Valid Sudoku

Problem Summary

Determine if a partially filled 9×9 Sudoku board is valid.

Pattern

- **Hash Set Validation of Rows, Cols, Boxes**

```python
def isValidSudoku(board):
    # Initialize sets for rows, cols, and boxes
    rows = [set() for _ in range(9)]
    cols = [set() for _ in range(9)]
    boxes = [set() for _ in range(9)]

    for r in range(9):
        for c in range(9):
            cell = board[r][c]

            # Skip empty cells
            if cell == '.':
                continue

            # Compute box index: (r//3)*3 + (c//3)
            box_idx = (r // 3) * 3 + (c // 3)

            # Check if already seen in row, col, or box
            if cell in rows[r] or cell in cols[c] or cell in boxes[box_idx]:
                return False

            # Add to all three sets
            rows[r].add(cell)
            cols[c].add(cell)
            boxes[box_idx].add(cell)

    return True


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # 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"]
    # ]
```

```python
# Expected Output: true

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 = isValidSudoku(board)
print("Output:", output)  # Output: True
```

Walkthrough: First few cells

- `board[0][0] = '5'`: add to `rows[0]`, `cols[0]`, `boxes[0]`
- `board[0][1] = '3'`: add to `rows[0]`, `cols[1]`, `boxes[0]`
- `board[1][0] = '6'`: add to `rows[1]`, `cols[0]`, `boxes[0]`
- `board[1][3] = '1'`: add to `rows[1]`, `cols[3]`, `boxes[1]`
- All unique → continue
- No duplicates found → return `True`

Complexity

- **Time**: O(1) — fixed 81 cells
- **Space**: O(1) — at most $9 \times 9 = 81$ elements stored

---

Math / Implementation

Pattern: Mathematical Transformation & Digit Manipulation

Also known as "Digit-by-Digit Processing"

How to Recognize

- Problem involves **reversing digits**, **converting between bases**, or **parsing symbolic representations** (like Roman numerals).
- Requires careful handling of **edge cases**: negative numbers, overflow, leading zeros.
- Often uses **modular arithmetic** (`%`, `//`) and **digit extraction** via loops.

Step-by-Step Thinking Process (The Recipe)

1. **Extract digits** using `% 10` and `// 10`.
2. **Build result** by multiplying accumulator by 10 and adding digit.
3. **Handle sign** separately (especially for reverse).
4. **Check overflow** — especially in languages with fixed integers (Python handles it, but still important conceptually).
5. For parsing (e.g., Roman), use **mapping + conditional logic** based on order.

Common Pitfalls & Edge Cases

- Not handling **negative numbers correctly** (e.g., `-123` reversed → `-321`, not `321`).
- Overflow: e.g., reversing `1534236469` → exceeds 32-bit int → return 0.
- Leading zeros in reverse (not an issue if using math).
- Incorrect parsing logic for symbols that depend on context (e.g., `IV = 4`, not `I+V=6`).

---

20. Roman to Integer

Problem Summary

Convert a Roman numeral string to an integer.

Pattern

- **Parsing with Mapping + Contextual Logic**

```python
def romanToInt(s):
    # Map Roman symbols to values
    roman_map = {
        'I': 1, 'V': 5, 'X': 10, 'L': 50,
        'C': 100, 'D': 500, 'M': 1000
    }
```

```python
    total = 0
    i = 0

    while i < len(s):
        # Check if current symbol is smaller than next one → subtract
        if i + 1 < len(s) and roman_map[s[i]] < roman_map[s[i + 1]]:
            total += roman_map[s[i + 1]] - roman_map[s[i]]
            i += 2  # Skip both symbols
        else:
            total += roman_map[s[i]]
            i += 1

    return total


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: s = "III"
    # Expected Output: 3

    output = romanToInt("III")
    print("Output:", output)  # Output: 3

    # Another example: s = "IV" → 4
    output = romanToInt("IV")
    print("Output:", output)  # Output: 4

    # Example: s = "IX" → 9
    output = romanToInt("IX")
    print("Output:", output)  # Output: 9

    # Example: s = "MCMXC" → 1990
    output = romanToInt("MCMXC")
    print("Output:", output)  # Output: 1990
```

Walkthrough: `s = "MCMXC"`

- `M`: 1000 → add 1000 → total=1000
- `C < M` → `CM` = 900 → add 900 → total=1900
- `X < C` → `XC` = 90 → add 90 → total=1990

Key insight: When a smaller symbol precedes a larger one, it's subtractive.

Complexity

- **Time**: O(N)
- **Space**: O(1)

---

21. Reverse Integer

Problem Summary

Reverse the digits of a 32-bit signed integer. Return 0 if overflow occurs.

Pattern

- **Math Manipulation + Overflow Handling**

```python
def reverse(x):
    # Handle sign
    sign = 1 if x >= 0 else -1
    x = abs(x)

    reversed_num = 0
    while x != 0:
        reversed_num = reversed_num * 10 + x % 10
        x //= 10

    # Apply sign
    result = sign * reversed_num

    # Check 32-bit signed integer range
    if result < -2**31 or result > 2**31 - 1:
        return 0

    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
```

```python
# Example Input: x = 123
# Expected Output: 321

output = reverse(123)
print("Output:", output)  # Output: 321

# Example: x = -123 → -321
output = reverse(-123)
print("Output:", output)  # Output: -321

# Example: x = 120 → 21
output = reverse(120)
print("Output:", output)  # Output: 21

# Example: x = 1534236469 → overflows → 0
output = reverse(1534236469)
print("Output:", output)  # Output: 0
```

Walkthrough: `x = 123`

- `sign = 1`, `x = 123`
- Loop:
  - `reversed_num = 0*10 + 3 = 3`
  - `x = 12`
  - `reversed_num = 3*10 + 2 = 32`
  - `x = 1`
  - `reversed_num = 32*10 + 1 = 321`
- Result: `321`

Complexity

- **Time**: O(log N) — number of digits
- **Space**: O(1)

---

22. Palindrome Number

Problem Summary

Check if an integer reads the same forward and backward (ignoring sign? No — negative numbers are not palindromes).

Pattern

- **Math (Reverse Half of Digits)**

```python
def isPalindrome(x):
    # Negative numbers are not palindromes
    if x < 0:
        return False

    # Single digit is palindrome
    if x < 10:
        return True

    # Reverse half of the number
    reversed_half = 0
    while x > reversed_half:
        reversed_half = reversed_half * 10 + x % 10
        x //= 10

    # If even digits: x == reversed_half
    # If odd digits: x == reversed_half // 10 (ignore middle digit)
    return x == reversed_half or x == reversed_half // 10


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: x = 121
    # Expected Output: true

    output = isPalindrome(121)
    print("Output:", output)  # Output: True

    # Example: x = -121 → false
    output = isPalindrome(-121)
    print("Output:", output)  # Output: False
```

```
    # Example: x = 10 → false
    output = isPalindrome(10)
    print("Output:", output)  # Output: False
```

Walkthrough: `x = 121`

- `x > 0`, so continue
- Loop:
  - `reversed_half = 0*10 + 1 = 1`, `x = 12`
  - `reversed_half = 1*10 + 2 = 12`, `x = 1`
- Now `x = 1`, `reversed_half = 12` → `x < reversed_half` → exit loop
- Check: `x == reversed_half // 10` → `1 == 12 // 10 = 1` → True

Palindrome!

Complexity

- **Time**: O(log N)
- **Space**: O(1)

---

23. Pow(x, n)

Problem Summary

Compute `x^n` efficiently (with negative exponents allowed).

Pattern

- **Binary Exponentiation (Fast Power)**

```
def myPow(x, n):
    # Handle negative exponent
    if n < 0:
        x = 1 / x
        n = -n

    result = 1
```

```python
    while n > 0:
        # If n is odd, multiply result by x
        if n % 2 == 1:
            result *= x

        # Square x and halve n
        x *= x
        n //= 2

    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: x = 2.00000, n = 10
    # Expected Output: 1024.00000

    output = myPow(2.0, 10)
    print("Output:", output)  # Output: 1024.0

    # Example: x = 2.1, n = 3 → 9.261
    output = myPow(2.1, 3)
    print("Output:", output)  # Output: 9.261

    # Example: x = 2.0, n = -2 → 0.25
    output = myPow(2.0, -2)
    print("Output:", output)  # Output: 0.25
```

Walkthrough: `x=2, n=10`

- `n=10` → even → square `x=4`, `n=5`
- `n=5` → odd → `result=1*4=4`, square `x=16`, `n=2`
- `n=2` → even → square `x=256`, `n=1`
- `n=1` → odd → `result=4*256=1024`, square `x=...`, `n=0`
- Return `1024`

  Uses `x^n = (x²)^(n/2)` recursively.

Complexity

- **Time**: O(log N)
- **Space**: O(1)