100 LeetCode Hard Problems Study Guide

Contents

1 Median of Two Sorted Arrays

Problem Description

Find the median of two sorted arrays with overall runtime complexity of $O(\log(m+n))$.

Solution Approach

- Use binary search on the smaller array.
- Find partition points in both arrays.
- Ensure elements on left $\leq$ elements on right.
- Calculate median based on total length (odd/even).

Key Algorithms

Binary Search, Partitioning

Edge Cases

Empty arrays, single element arrays, all elements in one array ¡ other array.

```python
from typing import List

def findMedianSortedArrays(nums1: List[int], nums2: List[int]) -> float:
    # Ensure nums1 is the smaller array for optimization
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    m, n = len(nums1), len(nums2)
    low, high = 0, m

    while low <= high:
        # Partition nums1 at i and nums2 at j
        i = (low + high) // 2
        j = (m + n + 1) // 2 - i

        # Handle edge values with infinity
        left1 = float('-inf') if i == 0 else nums1[i - 1]
        right1 = float('inf') if i == m else nums1[i]
        left2 = float('-inf') if j == 0 else nums2[j - 1]
        right2 = float('inf') if j == n else nums2[j]

        # Check if partition is correct
        if left1 <= right2 and left2 <= right1:
            # Found correct partition
            if (m + n) % 2 == 0:
                return (max(left1, left2) + min(right1, right2)) / 2
            else:
                return max(left1, left2)
        elif left1 > right2:
            # Move left in nums1
            high = i - 1
        else:
            # Move right in nums1
            low = i + 1

    return 0.0
```

Complexity: Time $O(\log(\min(m,n)))$, Space $O(1)$

2 Regular Expression Matching

Problem Description

Implement regex matching with '.' (any single char) and '*' (zero or more of preceding element).

Solution Approach

- Use dynamic programming with 2D table.
- `dp[i][j] = true` if `s[0:i]` matches `p[0:j]`.
- Handle '*' by checking zero occurrences or multiple occurrences.
- Build solution bottom-up.

Key Algorithms

Dynamic Programming

Edge Cases

Empty pattern/string, consecutive '*', '.*' matching everything.

```python
def isMatch(s: str, p: str) -> bool:
    m, n = len(s), len(p)
    # dp[i][j] means s[0:i] matches p[0:j]
    dp = [[False] * (n + 1) for _ in range(m + 1)]

    # Empty string matches empty pattern
    dp[0][0] = True

    # Handle patterns like a*, a*b*, etc. that can match empty string
    for j in range(2, n + 1):
        if p[j - 1] == '*':
            dp[0][j] = dp[0][j - 2]

    # Fill the dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if p[j - 1] == '*':
                # Check zero occurrences of preceding element
                dp[i][j] = dp[i][j - 2]
                # Check one or more occurrences
                if p[j - 2] == s[i - 1] or p[j - 2] == '.':
                    dp[i][j] = dp[i][j] or dp[i - 1][j]
            else:
                # Regular character match or '.'
                if p[j - 1] == s[i - 1] or p[j - 1] == '.':
                    dp[i][j] = dp[i - 1][j - 1]

    return dp[m][n]
```

Complexity: Time $O(mn)$, Space $O(mn)$

3 Merge k Sorted Lists

Problem Description

Merge k sorted linked lists into one sorted list.

Solution Approach

- Use min-heap to track smallest elements.
- Push first element of each list to heap.
- Pop minimum, add to result, push next from same list.
- Continue until heap is empty.

Key Algorithms

Min-Heap, Priority Queue

Edge Cases

Empty lists, single list, all ‘None‘ lists.

```
from typing import List, Optional
import heapq

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeKLists(lists: List[Optional[ListNode]]) -> Optional[ListNode]:
    # Min heap to store (value, list_index, node)
    heap = []

    # Add first node of each list to heap
    for i, node in enumerate(lists):
        if node:
            heapq.heappush(heap, (node.val, i, node))

    # Dummy head for result
    dummy = ListNode(0)
    current = dummy

    while heap:
        # Get minimum element
        val, list_idx, node = heapq.heappop(heap)

        # Add to result
        current.next = node
        current = current.next

        # Add next node from same list if exists
        if node.next:
            heapq.heappush(heap, (node.next.val, list_idx, node.next))

    return dummy.next
```

Complexity: Time $O(N \log k)$ where N = total nodes, Space $O(k)$

4 Reverse Nodes in k-Group

Problem Description

Reverse nodes of linked list in groups of k.

Solution Approach

- Count nodes to ensure enough for reversal.
- Reverse k nodes at a time.
- Connect reversed groups properly.
- Handle remaining nodes (don’t reverse if ¡ k).

Key Algorithms

Linked List Manipulation, Reversal

Edge Cases

k = 1, k ¿ list length, exact multiple of k nodes.

```
def reverseKGroup(head: Optional[ListNode], k: int) -> Optional[ListNode]:
    # Count total nodes
    count = 0
    node = head
```

```python
    while node:
        count += 1
        node = node.next

    # Dummy node to simplify edge cases
    dummy = ListNode(0)
    dummy.next = head
    prev_group = dummy

    while count >= k:
        # Current group starts after prev_group
        current = prev_group.next
        next_node = current.next

        # Reverse k nodes
        for _ in range(k - 1):
            current.next = next_node.next
            next_node.next = prev_group.next
            prev_group.next = next_node
            next_node = current.next

        # Move to next group
        prev_group = current
        count -= k

    return dummy.next
```

Complexity: Time $O(n)$, Space $O(1)$

5 Substring with Concatenation of All Words

Problem Description

Find all starting indices of substring(s) that is a concatenation of all words exactly once.

Solution Approach

- Use sliding window with hashmap.
- Window size = `len(words) * len(words[0])`.
- Check each possible starting position.
- Verify word counts match expected.

Key Algorithms

Sliding Window, Hash Map

Edge Cases

Duplicate words, overlapping results, single word.

```python
from typing import List
from collections import defaultdict

def findSubstring(s: str, words: List[str]) -> List[int]:
    if not s or not words:
        return []

    word_len = len(words[0])
    total_len = word_len * len(words)
    word_count = defaultdict(int)

    # Count frequency of each word
    for word in words:
        word_count[word] += 1

    result = []
```

```python

    # Try each possible starting position
    for i in range(len(s) - total_len + 1):
        seen = defaultdict(int)
        j = 0

        # Check if substring starting at i is valid
        while j < len(words):
            word = s[i + j * word_len : i + (j + 1) * word_len]

            if word not in word_count:
                break

            seen[word] += 1

            if seen[word] > word_count[word]:
                break

            j += 1

        # All words matched
        if j == len(words):
            result.append(i)

    return result
```

Complexity: Time $O(n \cdot m \cdot w)$ where n = len(s), m = len(words), w = word length, Space $O(m)$

6 Longest Valid Parentheses

Problem Description

Find length of longest valid parentheses substring.

Solution Approach

- Use stack to track indices.
- Push -1 initially as base.
- For '(': push index.
- For ')': pop and calculate length using current index - top of stack.

Key Algorithms

Stack

Edge Cases

All open/close parentheses, empty string, nested parentheses.

```python
def longestValidParentheses(s: str) -> int:
    max_length = 0
    stack = [-1]  # Base for length calculation

    for i, char in enumerate(s):
        if char == '(':
            stack.append(i)
        else:  # char == ')'
            stack.pop()

            if not stack:
                # No matching '(' for this ')'
                stack.append(i)
            else:
                # Calculate length of valid substring
                max_length = max(max_length, i - stack[-1])

    return max_length
```

Complexity: Time $O(n)$, Space $O(n)$

7 Sudoku Solver

Problem Description

Solve a 9x9 Sudoku puzzle by filling empty cells.

Solution Approach

- Use backtracking.
- For each empty cell, try digits 1-9.
- Check if placement is valid (row, column, 3x3 box).
- Recursively solve remaining board.
- Backtrack if no solution found.

Key Algorithms

Backtracking, Constraint Satisfaction

Edge Cases

Invalid initial board, multiple solutions (return any).

```python
def solveSudoku(board: List[List[str]]) -> None:
    def is_valid(row: int, col: int, num: str) -> bool:
        # Check row
        for j in range(9):
            if board[row][j] == num:
                return False

        # Check column
        for i in range(9):
            if board[i][col] == num:
                return False

        # Check 3x3 box
        box_row, box_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(box_row, box_row + 3):
            for j in range(box_col, box_col + 3):
                if board[i][j] == num:
                    return False

        return True

    def solve() -> bool:
        # Find empty cell
        for i in range(9):
            for j in range(9):
                if board[i][j] == '.':
                    # Try digits 1-9
                    for num in '123456789':
                        if is_valid(i, j, num):
                            board[i][j] = num

                            if solve():
                                return True

                            # Backtrack
                            board[i][j] = '.'

                    return False

        # All cells filled
        return True
```

```
    solve()
```

Complexity: Time $O(9^m)$ where m = empty cells, Space $O(1)$

8 First Missing Positive

Problem Description

Find the smallest missing positive integer in $O(n)$ time and $O(1)$ space.

Solution Approach

- Place each number in its correct position (`nums[i] = i+1`).
- Swap elements to their correct positions.
- Find first position where `nums[i] != i+1`.
- Handle numbers outside range [1, n].

Key Algorithms

Array Manipulation, Cyclic Sort

Edge Cases

All negative, all ¿ n, duplicates, [1,2,3,...,n].

```python
def firstMissingPositive(nums: List[int]) -> int:
    n = len(nums)

    # Place each positive integer at its correct position
    for i in range(n):
        while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]:
            # Swap to correct position
            correct_pos = nums[i] - 1
            nums[i], nums[correct_pos] = nums[correct_pos], nums[i]

    # Find first missing positive
    for i in range(n):
        if nums[i] != i + 1:
            return i + 1

    # All positions filled correctly, answer is n + 1
    return n + 1
```

Complexity: Time $O(n)$, Space $O(1)$

9 Trapping Rain Water

Problem Description

Calculate water trapped after raining given elevation map.

Solution Approach

- Use two pointers from both ends.
- Track max height seen from left and right.
- Water trapped = `min(left_max, right_max) - current height`.
- Move pointer with smaller max inward.

Key Algorithms

Two Pointers

Edge Cases

Monotonic array, no water trapped, single peak.

```python
def trap(height: List[int]) -> int:
    if not height:
        return 0

    left, right = 0, len(height) - 1
    left_max = right_max = 0
    water = 0

    while left < right:
        if height[left] < height[right]:
            # Process left side
            if height[left] >= left_max:
                left_max = height[left]
            else:
                water += left_max - height[left]
            left += 1
        else:
            # Process right side
            if height[right] >= right_max:
                right_max = height[right]
            else:
                water += right_max - height[right]
            right -= 1

    return water
```

Complexity: Time $O(n)$, Space $O(1)$

10 Wildcard Matching

Problem Description

Implement wildcard pattern matching with '¿ (any single char) and '*' (any sequence).

Solution Approach

- Use dynamic programming.
- `dp[i][j] = true` if `s[0:i]` matches `p[0:j]`.
- '*' can match empty or any sequence.
- '¿ matches exactly one character.

Key Algorithms

Dynamic Programming

Edge Cases

Multiple '*', leading/trailing '*', empty pattern/string.

```python
def isMatch(s: str, p: str) -> bool:
    m, n = len(s), len(p)
    dp = [[False] * (n + 1) for _ in range(m + 1)]

    # Empty pattern matches empty string
    dp[0][0] = True

    # Handle patterns with leading '*'
    for j in range(1, n + 1):
```

```python
        if p[j - 1] == '*':
            dp[0][j] = dp[0][j - 1]

    # Fill dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if p[j - 1] == '*':
                # '*' matches empty or any sequence
                dp[i][j] = dp[i - 1][j] or dp[i][j - 1]
            elif p[j - 1] == '?' or p[j - 1] == s[i - 1]:
                # Character match or '?'
                dp[i][j] = dp[i - 1][j - 1]

    return dp[m][n]
```

Complexity: Time $O(mn)$, Space $O(mn)$

11 N-Queens

Problem Description

Place N queens on NxN board so no two queens attack each other.

Solution Approach

- Use backtracking with row-by-row placement.
- Track columns, diagonals, anti-diagonals under attack.
- Try each column in current row.
- Recursively solve for next row.

Key Algorithms

Backtracking

Edge Cases

N = 1, N = 2,3 (no solution).

```python
def solveNQueens(n: int) -> List[List[str]]:
    result = []
    board = [['.'] * n for _ in range(n)]
    cols = set()
    diags = set()  # row - col
    anti_diags = set()  # row + col

    def backtrack(row: int) -> None:
        if row == n:
            # Found valid solution
            result.append([''.join(row) for row in board])
            return

        for col in range(n):
            if col in cols or (row - col) in diags or (row + col) in anti_diags:
                continue

            # Place queen
            board[row][col] = 'Q'
            cols.add(col)
            diags.add(row - col)
            anti_diags.add(row + col)

            # Try next row
            backtrack(row + 1)

            # Remove queen (backtrack)
            board[row][col] = '.'
```

```
            cols.remove(col)
            diags.remove(row - col)
            anti_diags.remove(row + col)

    backtrack(0)
    return result
```

Complexity: Time $O(N!)$, Space $O(N)$

12 N-Queens II

Problem Description

Return number of distinct N-Queens solutions.

Solution Approach

- Similar to N-Queens but count solutions instead of storing boards.
- Use backtracking with pruning.
- Track attacked positions efficiently.

Key Algorithms

Backtracking

Edge Cases

Same as N-Queens.

```
def totalNQueens(n: int) -> int:
    cols = set()
    diags = set()
    anti_diags = set()

    def backtrack(row: int) -> int:
        if row == n:
            return 1

        count = 0
        for col in range(n):
            if col in cols or (row - col) in diags or (row + col) in anti_diags:
                continue

            # Place queen
            cols.add(col)
            diags.add(row - col)
            anti_diags.add(row + col)

            count += backtrack(row + 1)

            # Remove queen
            cols.remove(col)
            diags.remove(row - col)
            anti_diags.remove(row + col)

        return count

    return backtrack(0)
```

Complexity: Time $O(N!)$, Space $O(N)$

13 Permutation Sequence

Problem Description

Return the kth permutation sequence of numbers 1 to n.

Solution Approach

- Use factorial number system.
- Determine which number goes in each position.
- `k-1` divided by `(n-1)!` gives index of first number.
- Update k and repeat for remaining positions.

Key Algorithms

Math, Factorial Number System

Edge Cases

k = 1 (first permutation), k = n! (last permutation).

```python
def getPermutation(n: int, k: int) -> str:
    # Calculate factorials
    factorial = [1] * n
    for i in range(1, n):
        factorial[i] = factorial[i - 1] * i

    # Available numbers
    numbers = list(range(1, n + 1))
    result = []

    # Convert to 0-indexed
    k -= 1

    # Build permutation digit by digit
    for i in range(n, 0, -1):
        # Find which number to use
        index = k // factorial[i - 1]
        result.append(str(numbers[index]))
        numbers.pop(index)

        # Update k for next position
        k %= factorial[i - 1]

    return ''.join(result)
```

Complexity: Time $O(n^2)$, Space $O(n)$

14 Valid Number

Problem Description

Validate if string is a valid decimal number.

Solution Approach

- Use finite state machine or careful parsing.
- Handle signs, digits, decimal point, exponent.
- Check valid transitions between components.
- Ensure required parts are present.

Key Algorithms

String Parsing, State Machine

Edge Cases

Leading/trailing spaces, multiple signs/decimals, 'e' without digits.

```python
def isNumber(s: str) -> bool:
    s = s.strip()
    if not s:
        return False

    # Flags to track what we've seen
    num_seen = dot_seen = e_seen = False
    num_after_e = True

    for i, char in enumerate(s):
        if char.isdigit():
            num_seen = True
            num_after_e = True
        elif char == '.':
            # Can't have dot after 'e' or second dot
            if e_seen or dot_seen:
                return False
            dot_seen = True
        elif char in 'eE':
            # Must have number before 'e' and can't have second 'e'
            if e_seen or not num_seen:
                return False
            e_seen = True
            num_after_e = False
        elif char in '+-':
            # Sign only at start or right after 'e'
            if i > 0 and s[i - 1] not in 'eE':
                return False
        else:
            return False

    # Must have at least one number and number after 'e' if present
    return num_seen and num_after_e
```

Complexity: Time $O(n)$, Space $O(1)$

15 Text Justification

Problem Description

Format text with full justification given max width per line.

Solution Approach

- Pack words into lines greedily.
- Distribute spaces evenly between words.
- Extra spaces go to leftmost gaps.
- Last line is left-justified.

Key Algorithms

Greedy, String Manipulation

Edge Cases

Single word per line, last line, one word exceeds maxWidth.

```python
def fullJustify(words: List[str], maxWidth: int) -> List[str]:
    result = []
    current_line = []
    current_length = 0

```

```python
    for word in words:
        # Check if word fits in current line
        if current_length + len(word) + len(current_line) > maxWidth:
            # Justify current line
            if len(current_line) == 1:
                # Single word - left justify
                result.append(current_line[0] + ' ' * (maxWidth - len(current_line[0])))
            else:
                # Multiple words - full justify
                total_spaces = maxWidth - current_length
                gaps = len(current_line) - 1
                spaces_per_gap = total_spaces // gaps
                extra_spaces = total_spaces % gaps

                line = ''
                for i, w in enumerate(current_line):
                    line += w
                    if i < gaps:
                        line += ' ' * spaces_per_gap
                        if i < extra_spaces:
                            line += ' '

                result.append(line)

            # Start new line
            current_line = [word]
            current_length = len(word)
        else:
            current_line.append(word)
            current_length += len(word)

    # Handle last line (left-justified)
    last_line = ' '.join(current_line)
    result.append(last_line + ' ' * (maxWidth - len(last_line)))

    return result
```

Complexity: Time $O(n)$, Space $O(n)$

16 Minimum Window Substring

Problem Description

Find minimum window in ‘s‘ containing all characters of ‘t‘.

Solution Approach

- Use sliding window with two pointers.
- Expand window until all chars included.
- Contract window while maintaining validity.
- Track minimum window seen.

Key Algorithms

Sliding Window, Hash Map

Edge Cases

No valid window, ‘t‘ longer than ‘s‘, duplicate chars in ‘t‘.

```python
from collections import Counter

def minWindow(s: str, t: str) -> str:
    if not s or not t:
        return ""

```

```python
    # Count characters in t
    dict_t = Counter(t)
    required = len(dict_t)

    # Sliding window
    left = right = 0
    formed = 0  # Number of unique chars in window with desired frequency

    # Count of chars in current window
    window_counts = {}

    # Result
    min_len = float('inf')
    min_left = 0

    while right < len(s):
        # Expand window
        char = s[right]
        window_counts[char] = window_counts.get(char, 0) + 1

        if char in dict_t and window_counts[char] == dict_t[char]:
            formed += 1

        # Contract window
        while left <= right and formed == required:
            # Update result
            if right - left + 1 < min_len:
                min_len = right - left + 1
                min_left = left

            # Remove from left
            char = s[left]
            window_counts[char] -= 1
            if char in dict_t and window_counts[char] < dict_t[char]:
                formed -= 1

            left += 1

        right += 1

    return "" if min_len == float('inf') else s[min_left:min_left + min_len]
```

Complexity: Time $O(|s| + |t|)$, Space $O(|s| + |t|)$

17 Largest Rectangle in Histogram

Problem Description

Find largest rectangle area in histogram.

Solution Approach

- Use stack to track indices of increasing heights.
- When smaller height found, calculate areas.
- Pop from stack until current height fits.
- Calculate area using popped height and width.

Key Algorithms

Stack, Monotonic Stack

Edge Cases

All same height, strictly increasing/decreasing, single bar.

```python
def largestRectangleArea(heights: List[int]) -> int:
    stack = []  # Indices of bars
    max_area = 0

    for i, height in enumerate(heights):
        # Pop bars taller than current
        while stack and heights[stack[-1]] > height:
            h_index = stack.pop()
            h = heights[h_index]
            # Width is from previous bar in stack to current position
            w = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, h * w)

        stack.append(i)

    # Process remaining bars
    while stack:
        h_index = stack.pop()
        h = heights[h_index]
        w = len(heights) if not stack else len(heights) - stack[-1] - 1
        max_area = max(max_area, h * w)

    return max_area
```

Complexity: Time $O(n)$, Space $O(n)$

18 Maximal Rectangle

Problem Description

Find largest rectangle containing only 1s in binary matrix.

Solution Approach

- Transform to histogram problem for each row.
- `Height[j]` = consecutive 1s above in column j.
- Apply largest rectangle in histogram to each row.
- Track maximum across all rows.

Key Algorithms

Dynamic Programming, Stack

Edge Cases

All 0s, all 1s, single row/column.

```python
def maximalRectangle(matrix: List[List[str]]) -> int:
    if not matrix or not matrix[0]:
        return 0

    rows, cols = len(matrix), len(matrix[0])
    heights = [0] * cols
    max_area = 0

    for row in matrix:
        # Update heights for current row
        for j in range(cols):
            if row[j] == '1':
                heights[j] += 1
            else:
                heights[j] = 0

        # Find max rectangle in this histogram
        max_area = max(max_area, largestRectangleArea(heights))

```

```python
    return max_area

def largestRectangleArea(heights: List[int]) -> int:
    stack = []
    max_area = 0

    for i, height in enumerate(heights):
        while stack and heights[stack[-1]] > height:
            h_index = stack.pop()
            h = heights[h_index]
            w = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, h * w)
        stack.append(i)

    while stack:
        h_index = stack.pop()
        h = heights[h_index]
        w = len(heights) if not stack else len(heights) - stack[-1] - 1
        max_area = max(max_area, h * w)

    return max_area
```

Complexity: Time $O(\text{rows} \times \text{cols})$, Space $O(\text{cols})$

19 Scramble String

Problem Description

Check if ‘s2‘ is scrambled string of ‘s1‘ (binary tree scrambling).

Solution Approach

- Use recursion with memoization.
- Try all possible split points.
- Check both non-swapped and swapped cases.
- Base case: strings are equal.

Key Algorithms

Recursion, Memoization

Edge Cases

Same strings, single character, anagrams.

```python
from functools import lru_cache

def isScramble(s1: str, s2: str) -> bool:
    @lru_cache(None)
    def helper(s1: str, s2: str) -> bool:
        # Base cases
        if s1 == s2:
            return True

        if sorted(s1) != sorted(s2):
            return False

        n = len(s1)

        # Try all split points
        for i in range(1, n):
            # Case 1: No swap
            if helper(s1[:i], s2[:i]) and helper(s1[i:], s2[i:]):
                return True

            # Case 2: Swap
```

```
            if helper(s1[:i], s2[-i:]) and helper(s1[i:], s2[:-i]):
                return True

        return False

    return helper(s1, s2)
```

Complexity: Time $O(n^4)$, Space $O(n^3)$

20 Distinct Subsequences

Problem Description

Count distinct subsequences of ‘s‘ that equal ‘t‘.

Solution Approach

- Use dynamic programming.
- `dp[i][j]` = ways to form `t[0:j]` from `s[0:i]`.
- If `s[i-1] == t[j-1]`: can use or skip `s[i-1]`.
- Otherwise: must skip `s[i-1]`.

Key Algorithms

Dynamic Programming

Edge Cases

Empty ‘t‘ (return 1), ‘t‘ longer than ‘s‘ (return 0).

```
def numDistinct(s: str, t: str) -> int:
    m, n = len(s), len(t)

    # dp[i][j] = number of ways to form t[0:j] from s[0:i]
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Empty t can be formed in one way (delete all)
    for i in range(m + 1):
        dp[i][0] = 1

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # Skip s[i-1]
            dp[i][j] = dp[i - 1][j]

            # Use s[i-1] if it matches t[j-1]
            if s[i - 1] == t[j - 1]:
                dp[i][j] += dp[i - 1][j - 1]

    return dp[m][n]
```

Complexity: Time $O(mn)$, Space $O(mn)$

21 Best Time to Buy and Sell Stock III

Problem Description

Maximum profit with at most 2 transactions.

Solution Approach

- Track 4 states: after buy1, sell1, buy2, sell2.
- Update states for each price.
- `buy1` = max profit after first buy.
- `sell2` = max profit after second sell.

Key Algorithms

State Machine, Dynamic Programming

Edge Cases

Prices always decrease, single price, need only 1 transaction.

```python
def maxProfit(prices: List[int]) -> int:
    if not prices:
        return 0

    # State variables
    buy1 = -prices[0]   # Max profit after first buy
    sell1 = 0           # Max profit after first sell
    buy2 = -prices[0]   # Max profit after second buy
    sell2 = 0           # Max profit after second sell

    for price in prices[1:]:
        # Update in reverse order to avoid using updated values
        sell2 = max(sell2, buy2 + price)
        buy2 = max(buy2, sell1 - price)
        sell1 = max(sell1, buy1 + price)
        buy1 = max(buy1, -price)

    return sell2
```

Complexity: Time $O(n)$, Space $O(1)$

22 Binary Tree Maximum Path Sum

Problem Description

Find maximum path sum in binary tree (path can start/end anywhere).

Solution Approach

- Use DFS to explore all paths.
- At each node, calculate max path through node.
- Return max path starting from node to parent.
- Track global maximum.

Key Algorithms

DFS, Tree Traversal

Edge Cases

All negative values, single node, straight line tree.

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def maxPathSum(root: Optional[TreeNode]) -> int:
    max_sum = float('-inf')

    def max_gain(node: Optional[TreeNode]) -> int:
        nonlocal max_sum

        if not node:
            return 0

        # Max sum starting from left/right child
        left_gain = max(max_gain(node.left), 0)
        right_gain = max(max_gain(node.right), 0)

        # Max path through current node
        current_max = node.val + left_gain + right_gain
        max_sum = max(max_sum, current_max)

        # Return max gain if we continue path through parent
        return node.val + max(left_gain, right_gain)

    max_gain(root)
    return max_sum
```

Complexity: Time $O(n)$, Space $O(h)$ where h = height

23 Word Ladder II

Problem Description

Find all shortest transformation sequences from ‘beginWord‘ to ‘endWord‘.

Solution Approach

- BFS to find shortest path length.
- Build adjacency graph during BFS.
- DFS/backtrack to find all paths of shortest length.
- Prune paths that can’t reach end in time.

Key Algorithms

BFS, DFS, Graph

Edge Cases

No path exists, multiple shortest paths, ‘beginWord‘ = ‘endWord‘.

```python
from collections import defaultdict, deque

def findLadders(beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:
    if endWord not in wordList:
        return []

    # Build adjacency list
    neighbors = defaultdict(list)
    wordList.append(beginWord)

```

```python
    for word in wordList:
        for i in range(len(word)):
            pattern = word[:i] + '*' + word[i + 1:]
            neighbors[pattern].append(word)

    # BFS to find shortest path and build graph
    visited = {beginWord}
    queue = deque([beginWord])
    found = False
    adjacency = defaultdict(list)

    while queue and not found:
        next_visited = set()
        for _ in range(len(queue)):
            word = queue.popleft()
            for i in range(len(word)):
                pattern = word[:i] + '*' + word[i + 1:]
                for neighbor in neighbors[pattern]:
                    if neighbor == endWord:
                        found = True
                    if neighbor not in visited:
                        if neighbor not in next_visited:
                            next_visited.add(neighbor)
                            queue.append(neighbor)
                        adjacency[word].append(neighbor)
        visited.update(next_visited)

    # DFS to find all shortest paths
    result = []

    def dfs(word: str, path: List[str]) -> None:
        if word == endWord:
            result.append(path[:])
            return

        for next_word in adjacency[word]:
            path.append(next_word)
            dfs(next_word, path)
            path.pop()

    if found:
        dfs(beginWord, [beginWord])

    return result
```

Complexity: Time $O(N \times L^2)$ where N = words, L = length, Space $O(N \times L)$

24 Word Ladder

Problem Description

Find length of shortest transformation sequence from ‘beginWord‘ to ‘endWord‘.

Solution Approach

- BFS from ‘beginWord‘.
- Generate all possible one-letter transformations.
- Check if transformation is in ‘wordList‘.
- Track visited to avoid cycles.

Key Algorithms

BFS, Hash Set

Edge Cases

No path, ‘beginWord‘ = ‘endWord‘, ‘endWord‘ not in list.

```python
def ladderLength(beginWord: str, endWord: str, wordList: List[str]) -> int:
    if endWord not in wordList:
        return 0

    wordSet = set(wordList)
    queue = deque([(beginWord, 1)])
    visited = {beginWord}

    while queue:
        word, length = queue.popleft()

        if word == endWord:
            return length

        # Try all one-letter transformations
        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                if c == word[i]:
                    continue

                next_word = word[:i] + c + word[i + 1:]

                if next_word in wordSet and next_word not in visited:
                    visited.add(next_word)
                    queue.append((next_word, length + 1))

    return 0
```

Complexity: Time $O(N \times L^2 \times 26)$, Space $O(N \times L)$

25 Palindrome Partitioning II

Problem Description

Minimum cuts needed to partition string into palindromes.

Solution Approach

- Precompute palindrome table.
- `dp[i]` = minimum cuts for `s[0:i]`.
- For each position, try all palindrome endings.
- Update minimum cuts needed.

Key Algorithms

Dynamic Programming

Edge Cases

Already palindrome (0 cuts), single chars, no palindromes ¿ 1.

```python
def minCut(s: str) -> int:
    n = len(s)

    # Precompute palindrome table
    is_palindrome = [[False] * n for _ in range(n)]

    for right in range(n):
        for left in range(right + 1):
            if s[left] == s[right] and (right - left <= 2 or is_palindrome[left + 1][right - 1]):
                is_palindrome[left][right] = True
```

```python

    # dp[i] = minimum cuts for s[0:i]
    dp = [float('inf')] * (n + 1)
    dp[0] = -1  # Empty string needs -1 cuts

    for i in range(1, n + 1):
        for j in range(i):
            if is_palindrome[j][i - 1]:
                dp[i] = min(dp[i], dp[j] + 1)

    return dp[n]
```

Complexity: Time $O(n^2)$, Space $O(n^2)$

26 Candy

Problem Description

Minimum candies to give children where higher rating gets more candy than neighbors.

Solution Approach

- Two passes: left-to-right and right-to-left.
- Left pass: ensure higher rating than left gets more.
- Right pass: ensure higher rating than right gets more.
- Take maximum of both requirements.

Key Algorithms

Greedy, Two Pass

Edge Cases

All same ratings, strictly increasing/decreasing.

```python
def candy(ratings: List[int]) -> int:
    n = len(ratings)
    candies = [1] * n

    # Left to right pass
    for i in range(1, n):
        if ratings[i] > ratings[i - 1]:
            candies[i] = candies[i - 1] + 1

    # Right to left pass
    for i in range(n - 2, -1, -1):
        if ratings[i] > ratings[i + 1]:
            candies[i] = max(candies[i], candies[i + 1] + 1)

    return sum(candies)
```

Complexity: Time $O(n)$, Space $O(n)$

27 Word Break II

Problem Description

Return all possible sentences by breaking 's' using dictionary words.

Solution Approach

- Use backtracking with memoization.
- Try each prefix that's a valid word.
- Recursively break remaining string.
- Combine results.

Key Algorithms

Backtracking, Memoization

Edge Cases

No valid breaks, multiple ways to break, overlapping words.

```python
def wordBreak(s: str, wordDict: List[str]) -> List[str]:
    word_set = set(wordDict)
    memo = {}

    def backtrack(start: int) -> List[str]:
        if start in memo:
            return memo[start]

        if start == len(s):
            return ['']

        sentences = []

        for end in range(start + 1, len(s) + 1):
            word = s[start:end]
            if word in word_set:
                # Recursively break the rest
                for sub_sentence in backtrack(end):
                    if sub_sentence:
                        sentences.append(word + ' ' + sub_sentence)
                    else:
                        sentences.append(word)

        memo[start] = sentences
        return sentences

    return backtrack(0)
```

Complexity: Time $O(n^3)$, Space $O(n^3)$

28 Max Points on a Line

Problem Description

Maximum number of points on the same straight line.

Solution Approach

- For each point as origin, calculate slopes to others.
- Use hash map to count points with same slope.
- Handle vertical lines and same points specially.
- Use GCD to normalize slopes.

Key Algorithms

Hash Map, GCD

Edge Cases

Duplicate points, vertical lines, all points collinear.

```python
from math import gcd
from collections import defaultdict

def maxPoints(points: List[List[int]]) -> int:
    if len(points) <= 2:
        return len(points)

    max_points = 0

    for i in range(len(points)):
        slopes = defaultdict(int)
        same_point = 1
        local_max = 0

        for j in range(i + 1, len(points)):
            dx = points[j][0] - points[i][0]
            dy = points[j][1] - points[i][1]

            if dx == 0 and dy == 0:
                same_point += 1
            else:
                # Normalize slope using GCD
                g = gcd(dx, dy)
                dx, dy = dx // g, dy // g

                # Ensure consistent sign
                if dx < 0:
                    dx, dy = -dx, -dy
                elif dx == 0:
                    dy = abs(dy)

                slopes[(dx, dy)] += 1
                local_max = max(local_max, slopes[(dx, dy)])

        max_points = max(max_points, local_max + same_point)

    return max_points
```

Complexity: Time $O(n^2)$, Space $O(n)$

29 Find Minimum in Rotated Sorted Array II

Problem Description

Find minimum in rotated sorted array with duplicates.

Solution Approach

- Modified binary search.
- Compare mid with right endpoint.
- If equal, can't determine side, reduce search space by 1.
- Otherwise, binary search on correct half.

Key Algorithms

Binary Search

Edge Cases

No rotation, all duplicates, single element.

```python
def findMin(nums: List[int]) -> int:
    left, right = 0, len(nums) - 1

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

        if nums[mid] > nums[right]:
            # Minimum is in right half
            left = mid + 1
        elif nums[mid] < nums[right]:
            # Minimum is in left half (including mid)
            right = mid
        else:
            # nums[mid] == nums[right], can't determine
            # Safely reduce search space
            right -= 1

    return nums[left]
```

Complexity: Time $O(n)$ worst case, $O(\log n)$ average, Space $O(1)$

30 Read N Characters Given read4 II - Call Multiple Times

Problem Description

Implement ‘read‘ method using ‘read4‘ that can be called multiple times.

Solution Approach

- Maintain internal buffer between calls.
- Read from buffer first if available.
- Call ‘read4‘ to refill buffer when needed.
- Handle partial reads correctly.

Key Algorithms

Buffer Management

Edge Cases

Multiple calls, buffer boundary, EOF handling.

```python
class Solution:
    def __init__(self):
        self.buffer = [''] * 4
        self.buffer_ptr = 0
        self.buffer_count = 0

    def read(self, buf: List[str], n: int) -> int:
        total_read = 0

        while total_read < n:
            # Read from buffer first
            if self.buffer_ptr < self.buffer_count:
                buf[total_read] = self.buffer[self.buffer_ptr]
                self.buffer_ptr += 1
                total_read += 1
            else:
                # Buffer empty, read more
                self.buffer_count = read4(self.buffer)
                self.buffer_ptr = 0

                if self.buffer_count == 0:
                    break

        return total_read
```

Complexity: Time $O(n)$, Space $O(1)$

31 Dungeon Game

Problem Description

Minimum initial health to reach bottom-right rescuing princess.

Solution Approach

- Work backwards from destination.
- `dp[i][j]` = minimum health needed at `(i,j)`.
- Need at least 1 health after any cell.
- Calculate based on minimum of right/down paths.

Key Algorithms

Dynamic Programming

Edge Cases

All positive values, large negative values, single cell.

```python
def calculateMinimumHP(dungeon: List[List[int]]) -> int:
    m, n = len(dungeon), len(dungeon[0])

    # dp[i][j] = minimum health needed to reach bottom-right from (i,j)
    dp = [[float('inf')] * (n + 1) for _ in range(m + 1)]
    dp[m][n - 1] = dp[m - 1][n] = 1

    # Work backwards
    for i in range(m - 1, -1, -1):
        for j in range(n - 1, -1, -1):
            # Minimum health needed for next step
            min_health = min(dp[i + 1][j], dp[i][j + 1])

            # Health needed at current cell
            dp[i][j] = max(1, min_health - dungeon[i][j])

    return dp[0][0]
```

Complexity: Time $O(mn)$, Space $O(mn)$

32 Best Time to Buy and Sell Stock IV

Problem Description

Maximum profit with at most ‘k‘ transactions.

Solution Approach

- If ‘k ¿= n/2‘, unlimited transactions.
- `dp[i][j]` = max profit with at most `i` transactions by day `j`.
- Track max profit after buying for each transaction.
- Update based on sell or hold.

Key Algorithms

Dynamic Programming

Edge Cases

‘k = 0‘, ‘k ¿= n/2‘ (unlimited), prices decreasing.

```
def maxProfit(k: int, prices: List[int]) -> int:
    if not prices or k == 0:
        return 0

    n = len(prices)

    # If k >= n/2, unlimited transactions
    if k >= n // 2:
        profit = 0
        for i in range(1, n):
            profit += max(0, prices[i] - prices[i - 1])
        return profit

    # dp[i][0] = max profit after at most i transactions and not holding
    # dp[i][1] = max profit after at most i transactions and holding
    dp = [[0, -prices[0]] for _ in range(k + 1)]

    for price in prices[1:]:
        for i in range(k, 0, -1):
            dp[i][0] = max(dp[i][0], dp[i][1] + price)
            dp[i][1] = max(dp[i][1], dp[i - 1][0] - price)

    return dp[k][0]
```

Complexity: Time $O(nk)$, Space $O(k)$

33 Word Search II

Problem Description

Find all words from dictionary in 2D board.

Solution Approach

- Build Trie from word list.
- DFS from each cell.
- Track current path in Trie.
- Mark found words to avoid duplicates.

Key Algorithms

Trie, DFS, Backtracking

Edge Cases

No words found, overlapping words, word prefix of another.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.word = None

def findWords(board: List[List[str]], words: List[str]) -> List[str]:
    # Build Trie
    root = TrieNode()
    for word in words:
        node = root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.word = word
```

```python

    m, n = len(board), len(board[0])
    result = []

    def dfs(i: int, j: int, node: TrieNode) -> None:
        if i < 0 or i >= m or j < 0 or j >= n:
            return

        char = board[i][j]
        if char not in node.children or char == '#':
            return

        node = node.children[char]

        if node.word:
            result.append(node.word)
            node.word = None  # Avoid duplicates

        # Mark as visited
        board[i][j] = '#'

        # Explore neighbors
        dfs(i + 1, j, node)
        dfs(i - 1, j, node)
        dfs(i, j + 1, node)
        dfs(i, j - 1, node)

        # Restore
        board[i][j] = char

    for i in range(m):
        for j in range(n):
            dfs(i, j, root)

    return result
```

Complexity: Time $O(m \times n \times 4^L)$ where L = max word length, Space $O(\text{total chars})$

34 Shortest Palindrome

Problem Description

Find shortest palindrome by adding characters in front.

Solution Approach

- Find longest palindrome starting from index 0.
- Use KMP algorithm's failure function.
- Create string `s + "#" + reverse(s)`.
- LPS value gives longest palindrome prefix.

Key Algorithms

KMP Algorithm, String Matching

Edge Cases

Already palindrome, single character, no palindrome prefix.

```python
def shortestPalindrome(s: str) -> str:
    if not s:
        return s

    # Create combined string for KMP
    combined = s + '#' + s[::-1]

```

```python
    # Build KMP failure function
    n = len(combined)
    lps = [0] * n

    for i in range(1, n):
        j = lps[i - 1]

        while j > 0 and combined[i] != combined[j]:
            j = lps[j - 1]

        if combined[i] == combined[j]:
            j += 1

        lps[i] = j

    # Length of longest palindrome prefix
    palindrome_len = lps[-1]

    # Add reverse of remaining suffix to front
    return s[palindrome_len:][::-1] + s
```

Complexity: Time $O(n)$, Space $O(n)$

35 The Skyline Problem

Problem Description

Output skyline formed by buildings.

Solution Approach

- Create events for building start/end.
- Sort events by position, then by height.
- Use multiset/heap to track active buildings.
- Add key point when max height changes.

Key Algorithms

Sweep Line, Priority Queue

Edge Cases

Overlapping buildings, same height, touching buildings.

```python
import heapq

def getSkyline(buildings: List[List[int]]) -> List[List[int]]:
    # Create events: (position, is_start, height)
    events = []
    for left, right, height in buildings:
        events.append((left, True, height))
        events.append((right, False, height))

    # Sort events
    events.sort(key=lambda x: (x[0], not x[1], -x[2] if x[1] else x[2]))

    result = []
    heights = [0]  # Min heap (use negative for max heap)

    for pos, is_start, height in events:
        if is_start:
            heapq.heappush(heights, -height)
        else:
            heights.remove(-height)
            heapq.heapify(heights)

```

```
        # Current max height
        max_height = -heights[0]

        # Add key point if height changed
        if not result or result[-1][1] != max_height:
            result.append([pos, max_height])

    return result
```

Complexity: Time $O(n^2 \log n)$, Space $O(n)$

36 Contains Duplicate III

Problem Description

Check if array has two indices ‘i,j‘ where ‘—nums[i]-nums[j]— ¡= valueDiff‘ and ‘—i-j— ¡= indexDiff‘.

Solution Approach

- Use sliding window with ordered set.
- For each element, check range `[num-valueDiff, num+valueDiff]`.
- Maintain window size of `indexDiff`.
- Use buckets for O(n) solution.

Key Algorithms

Sliding Window, Bucket Sort

Edge Cases

‘valueDiff = 0‘, ‘indexDiff = 0‘, negative numbers.

```python
def containsNearbyAlmostDuplicate(nums: List[int], indexDiff: int, valueDiff: int) -> bool:
    if indexDiff < 1 or valueDiff < 0:
        return False

    # Bucket approach
    buckets = {}
    bucket_size = valueDiff + 1

    for i, num in enumerate(nums):
        # Determine bucket
        bucket_id = num // bucket_size

        # Check current bucket
        if bucket_id in buckets:
            return True

        # Check adjacent buckets
        if bucket_id - 1 in buckets and abs(num - buckets[bucket_id - 1]) <= valueDiff:
            return True
        if bucket_id + 1 in buckets and abs(num - buckets[bucket_id + 1]) <= valueDiff:
            return True

        # Add to bucket
        buckets[bucket_id] = num

        # Remove old element
        if i >= indexDiff:
            old_bucket = nums[i - indexDiff] // bucket_size
            del buckets[old_bucket]

    return False
```

Complexity: Time $O(n)$, Space $O(\min(n, \text{indexDiff}))$

37 Basic Calculator

Problem Description

Evaluate expression with '+', '-', '(', ')'.

Solution Approach

- Use stack to handle parentheses.
- Track current number and sign.
- When '(' found, push current result and sign.
- When ')' found, pop and combine with previous.

Key Algorithms

Stack

Edge Cases

Nested parentheses, negative numbers, spaces.

```python
def calculate(s: str) -> int:
    stack = []
    result = 0
    number = 0
    sign = 1

    for char in s:
        if char.isdigit():
            number = number * 10 + int(char)
        elif char == '+':
            result += sign * number
            number = 0
            sign = 1
        elif char == '-':
            result += sign * number
            number = 0
            sign = -1
        elif char == '(':
            # Push current result and sign
            stack.append(result)
            stack.append(sign)
            result = 0
            sign = 1
        elif char == ')':
            result += sign * number
            number = 0
            # Pop sign and previous result
            result = stack.pop() * result + stack.pop()

    # Handle last number
    result += sign * number

    return result
```

Complexity: Time $O(n)$, Space $O(n)$

38 Number of Digit One

Problem Description

Count total occurrences of digit 1 in all numbers from 1 to 'n'.

Solution Approach

- Count digit by digit position.
- For each position, count complete cycles and remainder.
- Handle current digit cases: 0, 1, or ¿1.
- Sum counts for all positions.

Key Algorithms

Math, Digital DP

Edge Cases

'n = 0', single digit numbers, powers of 10.

```python
def countDigitOne(n: int) -> int:
    if n <= 0:
        return 0

    count = 0
    factor = 1

    while factor <= n:
        # Divide number into parts
        lower = n % factor
        current = (n // factor) % 10
        higher = n // (factor * 10)

        if current == 0:
            count += higher * factor
        elif current == 1:
            count += higher * factor + lower + 1
        else:
            count += (higher + 1) * factor

        factor *= 10

    return count
```

Complexity: Time $O(\log n)$, Space $O(1)$

39 Sliding Window Maximum

Problem Description

Maximum value in each sliding window of size 'k'.

Solution Approach

- Use deque to maintain decreasing order.
- Remove elements outside window.
- Remove smaller elements from right.
- Front of deque is always maximum.

Key Algorithms

Deque, Monotonic Queue

Edge Cases

'k = 1', 'k = n', all same values.

```python
from collections import deque

def maxSlidingWindow(nums: List[int], k: int) -> List[int]:
    if not nums or k == 0:
        return []

    # Deque stores indices
    dq = deque()
    result = []

    for i, num in enumerate(nums):
        # Remove indices outside window
        while dq and dq[0] <= i - k:
            dq.popleft()

        # Remove smaller elements from right
        while dq and nums[dq[-1]] <= num:
            dq.pop()

        dq.append(i)

        # Add to result after first window
        if i >= k - 1:
            result.append(nums[dq[0]])

    return result
```

Complexity: Time $O(n)$, Space $O(k)$

40 Strobogrammatic Number III

Problem Description

Count strobogrammatic numbers in range '[low, high]'.

Solution Approach

- Generate all strobogrammatic numbers of each length.
- Use recursion to build from middle outward.
- Count those within range.
- Handle edge cases for leading zeros.

Key Algorithms

Recursion, String Building

Edge Cases

Single digit range, 'low ¿ high', leading zeros.

```python
def strobogrammaticInRange(low: str, high: str) -> int:
    pairs = [('0', '0'), ('1', '1'), ('6', '9'), ('8', '8'), ('9', '6')]
    count = 0

    def dfs(left: str, right: str, remaining: int) -> None:
        nonlocal count

        if remaining == 0:
            num_str = left + right
            # Check if in range (handle leading zeros)
            if (len(num_str) == 1 or num_str[0] != '0') and \
               len(low) <= len(num_str) <= len(high) and \
```

```python
                low <= num_str <= high:
                    count += 1
                return

        for p1, p2 in pairs:
            dfs(left + p1, p2 + right, remaining - 2)

    # Generate numbers of each length
    for length in range(len(low), len(high) + 1):
        if length % 2 == 0:
            dfs('', '', length)
        else:
            # Odd length - middle can be 0, 1, or 8
            for mid in ['0', '1', '8']:
                dfs(mid, '', length - 1)

    return count
```

Complexity: Time $O(5^{(n/2)})$ where n = max length, Space $O(n)$

41 Paint House II

Problem Description

Minimum cost to paint ‘n‘ houses with ‘k‘ colors, no adjacent same color.

Solution Approach

- Track minimum and second minimum cost for previous row.
- For each house, use minimum if different color.
- Use second minimum if same color as minimum.
- Update minimums for next iteration.

Key Algorithms

Dynamic Programming

Edge Cases

‘k = 1‘ (impossible if ‘n ¿ 1‘), ‘n = 1‘, all costs same.

```python
def minCostII(costs: List[List[int]]) -> int:
    if not costs or not costs[0]:
        return 0

    n, k = len(costs), len(costs[0])

    if k == 1:
        return costs[0][0] if n == 1 else -1

    # Track min and second min from previous house
    prev_min = prev_second_min = 0
    prev_min_color = -1

    for house in range(n):
        curr_min = curr_second_min = float('inf')
        curr_min_color = -1

        for color in range(k):
            # Cost for this color
            cost = costs[house][color]
            if color == prev_min_color:
                cost += prev_second_min
            else:
                cost += prev_min

```

```
            # Update current minimums
            if cost < curr_min:
                curr_second_min = curr_min
                curr_min = cost
                curr_min_color = color
            elif cost < curr_second_min:
                curr_second_min = cost

        prev_min = curr_min
        prev_second_min = curr_second_min
        prev_min_color = curr_min_color

    return prev_min
```

Complexity: Time $O(nk)$, Space $O(1)$

42 Alien Dictionary

Problem Description

Derive lexicographic order from sorted alien words.

Solution Approach

- Build graph from adjacent word comparisons.
- Find first differing character pairs.
- Topological sort using DFS.
- Detect cycles (invalid order).

Key Algorithms

Topological Sort, Graph

Edge Cases

Invalid order, prefix relationships, single word.

```
from collections import defaultdict, deque

def alienOrder(words: List[str]) -> str:
    # Build adjacency list
    adj = defaultdict(set)
    in_degree = {c: 0 for word in words for c in word}

    # Compare adjacent words
    for i in range(len(words) - 1):
        w1, w2 = words[i], words[i + 1]
        min_len = min(len(w1), len(w2))

        # Check if w2 is prefix of w1 (invalid)
        if len(w1) > len(w2) and w1[:min_len] == w2:
            return ""

        # Find first different character
        for j in range(min_len):
            if w1[j] != w2[j]:
                if w2[j] not in adj[w1[j]]:
                    adj[w1[j]].add(w2[j])
                    in_degree[w2[j]] += 1
                break

    # Topological sort using BFS
    queue = deque([c for c in in_degree if in_degree[c] == 0])
    result = []

    while queue:
```

```
        char = queue.popleft()
        result.append(char)

        for neighbor in adj[char]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Check if all characters are included (no cycle)
    return ''.join(result) if len(result) == len(in_degree) else ""
```

Complexity: Time $O(\text{total chars})$, Space $O(1)$ since at most 26 letters

43 Closest Binary Search Tree Value II

Problem Description

Find ‘k‘ values in BST closest to ‘target‘.

Solution Approach

- Inorder traversal to get sorted values.
- Use two pointers or binary search to find closest.
- Expand window to get ‘k‘ values.
- Alternative: Use heap during traversal.

Key Algorithms

BST Traversal, Two Pointers

Edge Cases

‘k‘ equals tree size, ‘target‘ outside tree range.

```
def closestKValues(root: Optional[TreeNode], target: float, k: int) -> List[int]:
    # Inorder traversal to get sorted values
    values = []

    def inorder(node: Optional[TreeNode]) -> None:
        if not node:
            return
        inorder(node.left)
        values.append(node.val)
        inorder(node.right)

    inorder(root)

    # Find closest value using binary search
    left = 0
    right = len(values) - 1

    while right - left + 1 > k:
        if abs(values[left] - target) > abs(values[right] - target):
            left += 1
        else:
            right -= 1

    return values[left:right + 1]
```

Complexity: Time $O(n)$, Space $O(n)$

44 Integer to English Words

Problem Description

Convert integer to English words representation.

Solution Approach

- Handle groups of three digits.
- Process billions, millions, thousands, hundreds.
- Special cases for 0-19 and tens.
- Combine parts with proper spacing.

Key Algorithms

String Manipulation

Edge Cases

0, powers of 10, teens, exact thousands/millions.

```python
def numberToWords(num: int) -> str:
    if num == 0:
        return "Zero"

    # Define mappings
    ones = ["", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
            "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen",
            "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"]
    tens = ["", "", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty",
    "Ninety"]
    thousands = ["", "Thousand", "Million", "Billion"]

    def helper(num: int) -> str:
        if num == 0:
            return ""
        elif num < 20:
            return ones[num]
        elif num < 100:
            return tens[num // 10] + (" " + ones[num % 10] if num % 10 else "")
        else:
            return ones[num // 100] + " Hundred" + (" " + helper(num % 100) if num % 100
     else "")

    result = []
    group_index = 0

    while num > 0:
        if num % 1000 != 0:
            group_words = helper(num % 1000)
            if thousands[group_index]:
                group_words += " " + thousands[group_index]
            result.append(group_words)
        num //= 1000
        group_index += 1

    return " ".join(reversed(result))
```

Complexity: Time $O(1)$, Space $O(1)$

45 Expression Add Operators

Problem Description

Add operators '(+, -, *)' to digits to get target value.

Solution Approach

- Backtracking with current expression and value.
- Track previous operand for multiplication.

- Try splitting at each position.
- Handle leading zeros.

Key Algorithms

Backtracking

Edge Cases

Leading zeros, overflow, single digit.

```python
def addOperators(num: str, target: int) -> List[str]:
    result = []

    def backtrack(index: int, path: str, value: int, prev: int) -> None:
        if index == len(num):
            if value == target:
                result.append(path)
            return

        for i in range(index, len(num)):
            # Skip numbers with leading zeros
            if i > index and num[index] == '0':
                break

            curr_str = num[index:i + 1]
            curr_num = int(curr_str)

            if index == 0:
                # First number
                backtrack(i + 1, curr_str, curr_num, curr_num)
            else:
                # Addition
                backtrack(i + 1, path + '+' + curr_str, value + curr_num, curr_num)

                # Subtraction
                backtrack(i + 1, path + '-' + curr_str, value - curr_num, -curr_num)

                # Multiplication
                backtrack(i + 1, path + '*' + curr_str,
                          value - prev + prev * curr_num, prev * curr_num)

    backtrack(0, "", 0, 0)
    return result
```

Complexity: Time $O(4^n)$, Space $O(n)$

46 Find Median from Data Stream

Problem Description

Find median after each number insertion.

Solution Approach

- Use two heaps: max heap for smaller half, min heap for larger.
- Balance heaps to differ by at most 1.
- Median is top of larger heap or average of tops.
- Always insert to max heap first.

Key Algorithms

Two Heaps

Edge Cases

Single element, even/odd count.

```python
import heapq

class MedianFinder:
    def __init__(self):
        self.small = []  # Max heap (negate values)
        self.large = []  # Min heap

    def addNum(self, num: int) -> None:
        # Add to max heap
        heapq.heappush(self.small, -num)

        # Balance: move largest from small to large
        heapq.heappush(self.large, -heapq.heappop(self.small))

        # Ensure size property
        if len(self.large) > len(self.small):
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self) -> float:
        if len(self.small) > len(self.large):
            return -self.small[0]
        else:
            return (-self.small[0] + self.large[0]) / 2
```

Complexity: Time $O(\log n)$ insert, $O(1)$ find, Space $O(n)$

47 Best Meeting Point

Problem Description

Find point minimizing total Manhattan distance for all people.

Solution Approach

- Median minimizes sum of absolute deviations.
- Find median of x-coordinates and y-coordinates separately.
- Meeting point is `(median_x, median_y)`.
- Calculate total distance.

Key Algorithms

Math, Median

Edge Cases

Single person, all in line, grid boundaries.

```python
def minTotalDistance(grid: List[List[int]]) -> int:
    rows, cols = len(grid), len(grid[0])

    # Collect all x and y coordinates
    x_coords = []
    y_coords = []

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
                x_coords.append(i)
                y_coords.append(j)

    # Sort coordinates
    x_coords.sort()
```

```python
    y_coords.sort()

    # Find medians
    median_x = x_coords[len(x_coords) // 2]
    median_y = y_coords[len(y_coords) // 2]

    # Calculate total distance
    distance = 0
    for x in x_coords:
        distance += abs(x - median_x)
    for y in y_coords:
        distance += abs(y - median_y)

    return distance
```

Complexity: Time $O(mn \log mn)$, Space $O(mn)$

48 Serialize and Deserialize Binary Tree

Problem Description

Serialize binary tree to string and deserialize back.

Solution Approach

- Use preorder traversal for serialization.
- Use delimiter between values.
- Use special marker for null nodes.
- Deserialize using queue or recursion.

Key Algorithms

Tree Traversal, String Parsing

Edge Cases

Empty tree, single node, unbalanced tree.

```python
class Codec:
    def serialize(self, root: Optional[TreeNode]) -> str:
        def preorder(node: Optional[TreeNode]) -> List[str]:
            if not node:
                return ['null']
            return [str(node.val)] + preorder(node.left) + preorder(node.right)

        return ','.join(preorder(root))

    def deserialize(self, data: str) -> Optional[TreeNode]:
        values = iter(data.split(','))

        def build() -> Optional[TreeNode]:
            val = next(values)
            if val == 'null':
                return None

            node = TreeNode(int(val))
            node.left = build()
            node.right = build()
            return node

        return build()
```

Complexity: Time $O(n)$, Space $O(n)$

49 Remove Invalid Parentheses

Problem Description

Remove minimum parentheses to make valid.

Solution Approach

- BFS to find minimum removals.
- Try removing each parenthesis.
- Check if valid and add to next level.
- Stop at first valid level.

Key Algorithms

BFS

Edge Cases

Already valid, no valid possible, multiple solutions.

```python
def removeInvalidParentheses(s: str) -> List[str]:
    def is_valid(string: str) -> bool:
        count = 0
        for char in string:
            if char == '(':
                count += 1
            elif char == ')':
                count -= 1
                if count < 0:
                    return False
        return count == 0

    # BFS
    level = {s}

    while level:
        valid = [string for string in level if is_valid(string)]
        if valid:
            return valid

        # Generate next level
        next_level = set()
        for string in level:
            for i in range(len(string)):
                if string[i] in '()':
                    next_level.add(string[:i] + string[i + 1:])

        level = next_level

    return [""]
```

Complexity: Time $O(2^n)$, Space $O(2^n)$

50 Smallest Rectangle Enclosing Black Pixels

Problem Description

Find smallest rectangle containing all black pixels.

Solution Approach

- Use binary search on rows and columns.
- Find topmost, bottommost, leftmost, rightmost black pixels.
- Project to 1D and binary search.
- Rectangle is bounded by these coordinates.

Key Algorithms

Binary Search

Edge Cases

Single pixel, full grid black, scattered pixels.

```python
def minArea(image: List[List[str]], x: int, y: int) -> int:
    m, n = len(image), len(image[0])

    def search_rows(start: int, end: int, check_white: bool) -> int:
        while start < end:
            mid = (start + end) // 2
            if any(image[mid][j] == '1' for j in range(n)) == check_white:
                end = mid
            else:
                start = mid + 1
        return start

    def search_cols(start: int, end: int, check_white: bool) -> int:
        while start < end:
            mid = (start + end) // 2
            if any(image[i][mid] == '1' for i in range(m)) == check_white:
                end = mid
            else:
                start = mid + 1
        return start

    # Find boundaries
    top = search_rows(0, x, True)
    bottom = search_rows(x + 1, m, False)
    left = search_cols(0, y, True)
    right = search_cols(y + 1, n, False)

    return (bottom - top) * (right - left)
```

Complexity: Time $O((m+n)\log mn)$, Space $O(1)$

51 Number of Islands II

Problem Description

Count islands after each land addition operation.

Solution Approach

- Use Union-Find data structure.
- For each land addition, check 4 neighbors.
- Union with neighboring lands.
- Track number of connected components.

Key Algorithms

Union-Find

Edge Cases

Adding same position twice, no lands, all water.

```
class UnionFind:
    def __init__(self):
        self.parent = {}
        self.rank = {}
        self.count = 0

    def add(self, x: int) -> None:
        if x not in self.parent:
            self.parent[x] = x
            self.rank[x] = 0
            self.count += 1

    def find(self, x: int) -> int:
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x: int, y: int) -> None:
        px, py = self.find(x), self.find(y)
        if px == py:
            return

        if self.rank[px] < self.rank[py]:
            self.parent[px] = py
        elif self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[py] = px
            self.rank[px] += 1

        self.count -= 1

def numIslands2(m: int, n: int, positions: List[List[int]]) -> List[int]:
    uf = UnionFind()
    result = []

    for r, c in positions:
        key = r * n + c

        if key in uf.parent:
            result.append(uf.count)
            continue

        uf.add(key)

        # Check 4 neighbors
        for dr, dc in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            nr, nc = r + dr, c + dc
            neighbor_key = nr * n + nc

            if 0 <= nr < m and 0 <= nc < n and neighbor_key in uf.parent:
                uf.union(key, neighbor_key)

        result.append(uf.count)

    return result
```

Complexity: Time $O(k \times \alpha(mn))$ where k = operations, Space $O(k)$

52 Burst Balloons

Problem Description

Maximum coins by bursting balloons in optimal order.

Solution Approach

- Dynamic programming with interval.
- `dp[i][j]` = max coins bursting balloons 'i' to 'j'.
- Try each balloon as last to burst in interval.
- Add virtual balloons with value 1 at ends.

Key Algorithms

Dynamic Programming, Interval DP

Edge Cases

Single balloon, all same values.

```python
def maxCoins(nums: List[int]) -> int:
    # Add virtual balloons
    nums = [1] + nums + [1]
    n = len(nums)

    # dp[i][j] = max coins bursting balloons (i, j) exclusive
    dp = [[0] * n for _ in range(n)]

    # Iterate by interval length
    for length in range(3, n + 1):
        for left in range(n - length + 1):
            right = left + length - 1

            # Try each balloon as last to burst
            for k in range(left + 1, right):
                coins = nums[left] * nums[k] * nums[right]
                coins += dp[left][k] + dp[k][right]
                dp[left][right] = max(dp[left][right], coins)

    return dp[0][n - 1]
```

Complexity: Time $O(n^3)$, Space $O(n^2)$

53 Count of Smaller Numbers After Self

Problem Description

Count smaller elements to the right of each element.

Solution Approach

- Use merge sort with index tracking.
- During merge, count inversions.
- Track original indices through sorting.
- Alternative: Binary Indexed Tree.

Key Algorithms

Merge Sort, Binary Indexed Tree

Edge Cases

Sorted array, reverse sorted, duplicates.

```python
def countSmaller(nums: List[int]) -> List[int]:
    n = len(nums)
    result = [0] * n
    indices = list(range(n))

    def merge_sort(start: int, end: int) -> None:
        if end - start <= 1:
            return

        mid = (start + end) // 2
        merge_sort(start, mid)
        merge_sort(mid, end)

        # Merge and count
        temp = []
        i, j = start, mid

        while i < mid and j < end:
            if nums[indices[j]] < nums[indices[i]]:
                temp.append(indices[j])
                j += 1
            else:
                result[indices[i]] += j - mid
                temp.append(indices[i])
                i += 1

        while i < mid:
            result[indices[i]] += j - mid
            temp.append(indices[i])
            i += 1

        while j < end:
            temp.append(indices[j])
            j += 1

        indices[start:end] = temp

    merge_sort(0, n)
    return result
```

Complexity: Time $O(n \log n)$, Space $O(n)$

54 Shortest Distance from All Buildings

Problem Description

Find best empty land to build minimizing distance to all buildings.

Solution Approach

- BFS from each building.
- Track total distance to each empty land.
- Track reachability count.
- Return minimum distance reachable by all.

Key Algorithms

BFS, Grid Traversal

Edge Cases

No valid location, single building, obstacles blocking.

```
from collections import deque

def shortestDistance(grid: List[List[int]]) -> int:
    if not grid or not grid[0]:
        return -1

    m, n = len(grid), len(grid[0])
    buildings = []

    # Find all buildings
    for i in range(m):
        for j in range(n):
            if grid[i][j] == 1:
                buildings.append((i, j))

    # Distance sum and reachability count for each empty land
    dist_sum = [[0] * n for _ in range(m)]
    reach_count = [[0] * n for _ in range(m)]

    def bfs(start_i: int, start_j: int) -> bool:
        visited = [[False] * n for _ in range(m)]
        queue = deque([(start_i, start_j, 0)])
        visited[start_i][start_j] = True
        reached_buildings = 0

        while queue:
            i, j, dist = queue.popleft()

            for di, dj in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
                ni, nj = i + di, j + dj

                if 0 <= ni < m and 0 <= nj < n and not visited[ni][nj]:
                    visited[ni][nj] = True

                    if grid[ni][nj] == 0:
                        dist_sum[ni][nj] += dist + 1
                        reach_count[ni][nj] += 1
                        queue.append((ni, nj, dist + 1))
                    elif grid[ni][nj] == 1:
                        reached_buildings += 1

        return reached_buildings == len(buildings) - 1

    # BFS from each building
    for i, j in buildings:
        if not bfs(i, j):
            return -1

    # Find minimum distance
    min_dist = float('inf')
    for i in range(m):
        for j in range(n):
            if grid[i][j] == 0 and reach_count[i][j] == len(buildings):
                min_dist = min(min_dist, dist_sum[i][j])

    return min_dist if min_dist != float('inf') else -1
```

Complexity: Time $O(m^2n^2)$, Space $O(mn)$

55 Create Maximum Number

Problem Description

Create maximum number of length 'k' from two arrays preserving order.

Solution Approach

- Try all combinations of taking 'i' from 'nums1', 'k-i' from 'nums2'.
- Find maximum subsequence of given length.
- Merge two subsequences optimally.
- Compare all possibilities.

Key Algorithms

Greedy, Monotonic Stack

Edge Cases

'k = 0', 'k = m + n', one array empty.

```
def maxNumber(nums1: List[int], nums2: List[int], k: int) -> List[int]:
    def max_subsequence(nums: List[int], length: int) -> List[int]:
        drop = len(nums) - length
        stack = []

        for num in nums:
            while drop > 0 and stack and stack[-1] < num:
                stack.pop()
                drop -= 1
            stack.append(num)

        return stack[:length]

    def merge(arr1: List[int], arr2: List[int]) -> List[int]:
        result = []
        i = j = 0

        while i < len(arr1) or j < len(arr2):
            if i < len(arr1) and (j >= len(arr2) or arr1[i:] > arr2[j:]):
                result.append(arr1[i])
                i += 1
            else:
                result.append(arr2[j])
                j += 1

        return result

    max_result = []

    for i in range(max(0, k - len(nums2)), min(k, len(nums1)) + 1):
        sub1 = max_subsequence(nums1, i)
        sub2 = max_subsequence(nums2, k - i)
        merged = merge(sub1, sub2)
        max_result = max(max_result, merged)

    return max_result
```

Complexity: Time $O(k \times (m + n + k))$, Space $O(k)$

56 Count of Range Sum

Problem Description

Count ranges where sum is in '[lower, upper]'.

Solution Approach

- Use merge sort to count during merge.
- Convert to prefix sum problem.

- Count pairs where `lower <= prefix[j] - prefix[i] <= upper`.
- Use merge sort to efficiently count.

Key Algorithms

Merge Sort, Prefix Sum

Edge Cases

Single element, all negative, empty range.

```python
def countRangeSum(nums: List[int], lower: int, upper: int) -> int:
    # Compute prefix sums
    prefix = [0]
    for num in nums:
        prefix.append(prefix[-1] + num)

    def merge_sort(start: int, end: int) -> int:
        if end - start <= 1:
            return 0

        mid = (start + end) // 2
        count = merge_sort(start, mid) + merge_sort(mid, end)

        # Count valid ranges crossing mid
        j = k = mid
        for i in range(start, mid):
            while j < end and prefix[j] - prefix[i] < lower:
                j += 1
            while k < end and prefix[k] - prefix[i] <= upper:
                k += 1
            count += k - j

        # Merge sorted halves
        temp = []
        i, j = start, mid
        while i < mid and j < end:
            if prefix[i] <= prefix[j]:
                temp.append(prefix[i])
                i += 1
            else:
                temp.append(prefix[j])
                j += 1

        temp.extend(prefix[i:mid])
        temp.extend(prefix[j:end])
        prefix[start:end] = temp

        return count

    return merge_sort(0, len(prefix))
```

Complexity: Time $O(n \log n)$, Space $O(n)$

57 Longest Increasing Path in a Matrix

Problem Description

Find longest increasing path in matrix.

Solution Approach

- DFS with memoization from each cell.
- Explore 4 directions with increasing values.
- Cache results to avoid recomputation.
- Return maximum among all starting points.

Key Algorithms

DFS, Memoization

Edge Cases

Single cell, all same values, strictly increasing.

```
def longestIncreasingPath(matrix: List[List[int]]) -> int:
    if not matrix or not matrix[0]:
        return 0

    m, n = len(matrix), len(matrix[0])
    memo = {}

    def dfs(i: int, j: int) -> int:
        if (i, j) in memo:
            return memo[(i, j)]

        max_path = 1

        for di, dj in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            ni, nj = i + di, j + dj

            if 0 <= ni < m and 0 <= nj < n and matrix[ni][nj] > matrix[i][j]:
                max_path = max(max_path, 1 + dfs(ni, nj))

        memo[(i, j)] = max_path
        return max_path

    return max(dfs(i, j) for i in range(m) for j in range(n))
```

Complexity: Time $O(mn)$, Space $O(mn)$

58 Patching Array

Problem Description

Minimum patches to array so sums cover '[1, n]'.

Solution Approach

- Track maximum reachable number.
- If can't reach next number, add it as patch.
- When adding number 'x', extend range by 'x'.
- Continue until reach 'n'.

Key Algorithms

Greedy

Edge Cases

Empty array, 'n = 1', array already covers range.

```
def minPatches(nums: List[int], n: int) -> int:
    patches = 0
    i = 0
    miss = 1  # Smallest number we can't form

    while miss <= n:
        if i < len(nums) and nums[i] <= miss:
            # Can use nums[i] to extend range
            miss += nums[i]
            i += 1
        else:
```

```python
            # Need to patch with 'miss'
            miss += miss
            patches += 1

    return patches
```

Complexity: Time $O(m + \log n)$ where m = len(nums), Space $O(1)$

59 Reconstruct Itinerary

Problem Description

Find itinerary visiting all tickets exactly once, lexicographically smallest.

Solution Approach

- Build adjacency list with sorted destinations.
- Use DFS with backtracking.
- Use tickets in order, mark as used.
- Hierholzer's algorithm for Eulerian path.

Key Algorithms

DFS, Eulerian Path

Edge Cases

Multiple valid paths, cycles, dead ends.

```python
from collections import defaultdict

def findItinerary(tickets: List[List[str]]) -> List[str]:
    # Build graph
    graph = defaultdict(list)
    for src, dst in sorted(tickets, reverse=True):
        graph[src].append(dst)

    result = []

    def dfs(airport: str) -> None:
        while graph[airport]:
            next_airport = graph[airport].pop()
            dfs(next_airport)
        result.append(airport)

    dfs("JFK")
    return result[::-1]
```

Complexity: Time $O(E \log E)$ where E = edges, Space $O(E)$

60 Self Crossing

Problem Description

Check if path crosses itself.

Solution Approach

- Check if current line crosses any of last 3-5 lines.
- Line 'i' can only cross lines 'i-3', 'i-4', or 'i-5'.
- Check intersection conditions for each case.
- Use geometric conditions.

Key Algorithms

Geometry, Line Intersection

Edge Cases

Less than 4 moves, spiral patterns.

```
def isSelfCrossing(distance: List[int]) -> bool:
    n = len(distance)

    for i in range(3, n):
        # Fourth line crosses first line
        if i >= 3:
            if distance[i] >= distance[i-2] and distance[i-1] <= distance[i-3]:
                return True

        # Fifth line crosses second line
        if i >= 4:
            if distance[i-1] == distance[i-3] and \
               distance[i] + distance[i-4] >= distance[i-2]:
                return True

        # Sixth line crosses third line
        if i >= 5:
            if distance[i-2] >= distance[i-4] and \
               distance[i] + distance[i-4] >= distance[i-2] and \
               distance[i-1] + distance[i-5] >= distance[i-3] and \
               distance[i-3] >= distance[i-1]:
                return True

    return False
```

Complexity: Time $O(n)$, Space $O(1)$

61 Palindrome Pairs

Problem Description

Find pairs of words that form palindromes when concatenated.

Solution Approach

- Use Trie or HashMap for efficient lookup.
- For each word, check if reverse exists.
- Check partial matches for different lengths.
- Handle empty strings and self-palindromes.

Key Algorithms

Trie, String Matching

Edge Cases

Empty strings, single character, self-palindromes.

```
def palindromePairs(words: List[str]) -> List[List[int]]:
    word_dict = {word: i for i, word in enumerate(words)}
    result = []

    for i, word in enumerate(words):
        # Check all possible splits
        for j in range(len(word) + 1):
            prefix = word[:j]
            suffix = word[j:]

```

```python
            # If prefix is palindrome, check if reverse suffix exists
            if prefix == prefix[::-1]:
                rev_suffix = suffix[::-1]
                if rev_suffix in word_dict and word_dict[rev_suffix] != i:
                    result.append([word_dict[rev_suffix], i])

            # If suffix is palindrome, check if reverse prefix exists
            if j > 0 and suffix == suffix[::-1]:
                rev_prefix = prefix[::-1]
                if rev_prefix in word_dict and word_dict[rev_prefix] != i:
                    result.append([i, word_dict[rev_prefix]])

    return result
```

Complexity: Time $O(n \times k^2)$ where k = average word length, Space $O(nk)$

62 Data Stream as Disjoint Intervals

Problem Description

Maintain disjoint intervals from data stream.

Solution Approach

- Use TreeMap/SortedList to maintain intervals.
- When adding number, check adjacent intervals.
- Merge if necessary.
- Handle three cases: extend left, extend right, bridge gap.

Key Algorithms

Interval Merging, Binary Search

Edge Cases

Duplicate values, single point intervals.

```python
from sortedcontainers import SortedList

class SummaryRanges:
    def __init__(self):
        self.intervals = SortedList()

    def addNum(self, value: int) -> None:
        # Find position to insert
        idx = self.intervals.bisect_left([value, value])

        # Check if can merge with previous interval
        if idx > 0 and self.intervals[idx - 1][1] >= value - 1:
            self.intervals[idx - 1][1] = max(self.intervals[idx - 1][1], value)
            # Check if can merge with next interval
            if idx < len(self.intervals) and self.intervals[idx][0] <= value + 1:
                self.intervals[idx - 1][1] = max(self.intervals[idx - 1][1],
                                                 self.intervals[idx][1])
                self.intervals.pop(idx)
        # Check if can merge with next interval
        elif idx < len(self.intervals) and self.intervals[idx][0] <= value + 1:
            self.intervals[idx][0] = min(self.intervals[idx][0], value)
        else:
            # Create new interval
            self.intervals.add([value, value])

    def getIntervals(self) -> List[List[int]]:
        return list(self.intervals)
```

Complexity: Time $O(\log n)$ per add, Space $O(n)$

63 Russian Doll Envelopes

Problem Description

Maximum envelopes that can be nested (by width and height).

Solution Approach

- Sort by width ascending, height descending.
- Find LIS on heights.
- Height descending ensures same width won't nest.
- Use binary search for $O(n \log n)$ LIS.

Key Algorithms

LIS, Binary Search

Edge Cases

Same dimensions, single envelope, all same width/height.

```python
from bisect import bisect_left

def maxEnvelopes(envelopes: List[List[int]]) -> int:
    # Sort by width ascending, height descending
    envelopes.sort(key=lambda x: (x[0], -x[1]))

    # Find LIS on heights
    dp = []

    for _, height in envelopes:
        idx = bisect_left(dp, height)
        if idx == len(dp):
            dp.append(height)
        else:
            dp[idx] = height

    return len(dp)
```

Complexity: Time $O(n \log n)$, Space $O(n)$

64 Rearrange String k Distance Apart

Problem Description

Rearrange string so same characters are at least ‘k‘ distance apart.

Solution Approach

- Count character frequencies.
- Use max heap to get most frequent.
- Use queue to track cooling characters.
- Greedily place characters.

Key Algorithms

Heap, Greedy

Edge Cases

'k = 0', 'k ¿' string length, impossible arrangement.

```
from collections import Counter
import heapq

def rearrangeString(s: str, k: int) -> str:
    if k <= 1:
        return s

    # Count frequencies
    freq = Counter(s)

    # Max heap of (-frequency, char)
    heap = [(-count, char) for char, count in freq.items()]
    heapq.heapify(heap)

    result = []
    queue = deque()  # Characters in cooling period

    while heap or queue:
        # Move cooled characters back to heap
        if len(result) >= k and queue:
            count, char = queue.popleft()
            if count < 0:
                heapq.heappush(heap, (count, char))

        if not heap:
            return ""  # No valid arrangement

        # Use most frequent character
        count, char = heapq.heappop(heap)
        result.append(char)

        # Add to cooling queue if more instances remain
        if count < -1:
            queue.append((count + 1, char))

    return ''.join(result)
```

Complexity: Time $O(n \log 26) = O(n)$, Space $O(26) = O(1)$

65 Max Sum of Rectangle No Larger Than K

Problem Description

Find maximum sum rectangle with sum $\leq k$.

Solution Approach

- Fix left and right columns.
- Use Kadane's algorithm variant with constraint.
- Use TreeSet/SortedList to find best prefix.
- Binary search for prefix sum.

Key Algorithms

Kadane's Algorithm, Binary Search

Edge Cases

All negative, 'k ¡' all elements, single element = 'k'.

```python
from sortedcontainers import SortedList

def maxSumSubmatrix(matrix: List[List[int]], k: int) -> int:
    m, n = len(matrix), len(matrix[0])
    max_sum = float('-inf')

    # Try all left boundaries
    for left in range(n):
        # Row sums for current left-right range
        row_sums = [0] * m

        # Try all right boundaries
        for right in range(left, n):
            # Update row sums
            for i in range(m):
                row_sums[i] += matrix[i][right]

            # Find max subarray sum <= k
            sorted_sums = SortedList([0])
            curr_sum = 0

            for row_sum in row_sums:
                curr_sum += row_sum
                # Find smallest prefix sum >= curr_sum - k
                idx = sorted_sums.bisect_left(curr_sum - k)
                if idx < len(sorted_sums):
                    max_sum = max(max_sum, curr_sum - sorted_sums[idx])
                sorted_sums.add(curr_sum)

    return max_sum
```

Complexity: Time $O(n^2 \times m \log m)$, Space $O(m)$

66 Insert Delete GetRandom O(1) - Duplicates allowed

Problem Description

Data structure supporting insert, remove, getRandom with duplicates.

Solution Approach

- Use array for random access.
- HashMap maps value to set of indices.
- For remove, swap with last element.
- Update indices after swap.

Key Algorithms

Hash Map, Array

Edge Cases

Remove non-existent, single element, all duplicates.

```python
import random
from collections import defaultdict

class RandomizedCollection:
    def __init__(self):
        self.nums = []
        self.indices = defaultdict(set)

    def insert(self, val: int) -> bool:
        self.indices[val].add(len(self.nums))
        self.nums.append(val)
        return len(self.indices[val]) == 1
```

```python

    def remove(self, val: int) -> bool:
        if not self.indices[val]:
            return False

        # Get index to remove
        remove_idx = self.indices[val].pop()
        last_val = self.nums[-1]

        # Swap with last element
        self.nums[remove_idx] = last_val

        # Update indices
        if self.indices[last_val]:
            self.indices[last_val].add(remove_idx)
            self.indices[last_val].discard(len(self.nums) - 1)

        self.nums.pop()
        return True

    def getRandom(self) -> int:
        return random.choice(self.nums)
```

Complexity: Time $O(1)$ all operations, Space $O(n)$

67 Perfect Rectangle

Problem Description

Check if rectangles form perfect rectangle without overlap.

Solution Approach

- Track corner points - should appear even times except 4.
- Sum areas should equal bounding rectangle.
- Use set to track corners with odd count.
- Check final corners form rectangle.

Key Algorithms

Hash Set, Geometry

Edge Cases

Single rectangle, gaps, overlaps.

```python
def isRectangleCover(rectangles: List[List[int]]) -> bool:
    corners = set()
    area = 0

    min_x = min_y = float('inf')
    max_x = max_y = float('-inf')

    for x1, y1, x2, y2 in rectangles:
        # Update bounding box
        min_x = min(min_x, x1)
        min_y = min(min_y, y1)
        max_x = max(max_x, x2)
        max_y = max(max_y, y2)

        # Add area
        area += (x2 - x1) * (y2 - y1)

        # Toggle corners
        for x, y in [(x1, y1), (x1, y2), (x2, y1), (x2, y2)]:
            if (x, y) in corners:
```

```python
                corners.remove((x, y))
            else:
                corners.add((x, y))

    # Check area matches
    if area != (max_x - min_x) * (max_y - min_y):
        return False

    # Check exactly 4 corners remain
    expected = {(min_x, min_y), (min_x, max_y), (max_x, min_y), (max_x, max_y)}
    return corners == expected
```

Complexity: Time $O(n)$, Space $O(n)$

68 Frog Jump

Problem Description

Can frog cross river jumping on stones with constraints.

Solution Approach

- Dynamic programming with states (`stone, last_jump`).
- From each stone, try jumps of `k-1, k, k+1`.
- Use memoization to avoid recomputation.
- Check if can reach last stone.

Key Algorithms

Dynamic Programming, Memoization

Edge Cases

No valid path, first jump must be 1, gaps too large.

```python
def canCross(stones: List[int]) -> bool:
    stone_set = set(stones)
    memo = {}

    def dfs(pos: int, jump: int) -> bool:
        if pos == stones[-1]:
            return True

        if (pos, jump) in memo:
            return memo[(pos, jump)]

        # Try jumps of k-1, k, k+1
        for next_jump in [jump - 1, jump, jump + 1]:
            if next_jump > 0 and pos + next_jump in stone_set:
                if dfs(pos + next_jump, next_jump):
                    memo[(pos, jump)] = True
                    return True

        memo[(pos, jump)] = False
        return False

    # First jump must be 1
    return stones[1] == 1 and dfs(1, 1)
```

Complexity: Time $O(n^2)$, Space $O(n^2)$

69 Trapping Rain Water II

Problem Description

Trap rainwater in 2D elevation map.

Solution Approach

- Use min heap starting from boundaries.
- Process cells from lowest to highest.
- Water level determined by minimum boundary.
- Update neighbors and add to heap.

Key Algorithms

Priority Queue, BFS

Edge Cases

No water trapped, bowl shape, single cell.

```
import heapq

def trapRainWater(heightMap: List[List[int]]) -> int:
    if not heightMap or not heightMap[0]:
        return 0

    m, n = len(heightMap), len(heightMap[0])
    visited = [[False] * n for _ in range(m)]
    heap = []

    # Add boundary cells to heap
    for i in range(m):
        for j in range(n):
            if i == 0 or i == m - 1 or j == 0 or j == n - 1:
                heapq.heappush(heap, (heightMap[i][j], i, j))
                visited[i][j] = True

    water = 0
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    while heap:
        height, x, y = heapq.heappop(heap)

        # Check neighbors
        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            if 0 <= nx < m and 0 <= ny < n and not visited[nx][ny]:
                visited[nx][ny] = True
                # Water trapped is difference from current level
                water += max(0, height - heightMap[nx][ny])
                # Add neighbor with updated height
                heapq.heappush(heap, (max(height, heightMap[nx][ny]), nx, ny))

    return water
```

Complexity: Time $O(mn \log(mn))$, Space $O(mn)$

70 Split Array Largest Sum

Problem Description

Split array into 'k' subarrays minimizing largest sum.

Solution Approach

- Binary search on the answer.
- For each candidate sum, check if possible with 'k' splits.
- Greedily assign elements to subarrays.
- Adjust search range based on feasibility.

Key Algorithms

Binary Search, Greedy

Edge Cases

‘k = 1‘, ‘k = n‘, all same values.

```python
def splitArray(nums: List[int], k: int) -> int:
    def can_split(max_sum: int) -> bool:
        count = 1
        current_sum = 0

        for num in nums:
            if current_sum + num > max_sum:
                count += 1
                current_sum = num
                if count > k:
                    return False
            else:
                current_sum += num

        return True

    # Binary search range
    left = max(nums)
    right = sum(nums)

    while left < right:
        mid = (left + right) // 2
        if can_split(mid):
            right = mid
        else:
            left = mid + 1

    return left
```

Complexity: Time $O(n \log(\text{sum}))$, Space $O(1)$

71 Minimum Unique Word Abbreviation

Problem Description

Find shortest abbreviation of ‘target‘ unique among ‘dictionary‘.

Solution Approach

- Generate all possible abbreviations.
- Use bit manipulation for abbreviation patterns.
- Check uniqueness against dictionary.
- Return shortest unique abbreviation.

Key Algorithms

Bit Manipulation, BFS

Edge Cases

No abbreviation needed, single character, all words same length.

```python
def minAbbreviation(target: str, dictionary: List[str]) -> str:
    m = len(target)
    diffs = []

    # Pre-calculate difference masks
    for word in dictionary:
```

```python
        if len(word) == m:
            diff = 0
            for i in range(m):
                if target[i] != word[i]:
                    diff |= 1 << i
            if diff == 0:  # Same word in dictionary
                return ""
            diffs.append(diff)

    if not diffs:
        return str(m)

    # Try abbreviations by length
    for length in range(m + 1):
        for cand in range(1 << m):
            if bin(cand).count('1') != length:
                continue

            # Check if valid abbreviation
            if all(cand & diff for diff in diffs):
                # Build abbreviation string
                result = []
                count = 0
                for i in range(m):
                    if cand & (1 << i):
                        if count:
                            result.append(str(count))
                            count = 0
                        result.append(target[i])
                    else:
                        count += 1
                if count:
                    result.append(str(count))
                return ''.join(result)

    return ""
```

Complexity: Time $O(2^m \times n)$, Space $O(n)$

72 Word Squares

Problem Description

Find all word squares from given words.

Solution Approach

- Build prefix map for quick lookup.
- Use backtracking to build squares.
- For row ‘i‘, prefix is column ‘i‘ of previous rows.
- Ensure symmetric property maintained.

Key Algorithms

Backtracking, Trie/HashMap

Edge Cases

No valid squares, single letter words, all same word.

```python
from collections import defaultdict

def wordSquares(words: List[str]) -> List[List[str]]:
    n = len(words[0])
    prefix_map = defaultdict(list)

```

```python
    # Build prefix map
    for word in words:
        for i in range(n):
            prefix_map[word[:i]].append(word)

    result = []

    def backtrack(square: List[str]) -> None:
        pos = len(square)
        if pos == n:
            result.append(square[:])
            return

        # Get required prefix for next word
        prefix = ''.join(square[i][pos] for i in range(pos))

        # Try all words with this prefix
        for word in prefix_map[prefix]:
            square.append(word)
            backtrack(square)
            square.pop()

    # Try each word as first word
    for word in words:
        backtrack([word])

    return result
```

Complexity: Time $O(N \times 26^L)$ where N = words, L = length, Space $O(NL)$

73 Serialize and Deserialize N-ary Tree

Problem Description

Serialize/deserialize N-ary tree.

Solution Approach

- Use preorder traversal with child count.
- Format: `value,child_count,children...`
- Recursively serialize each node.
- Deserialize by reading count and recursing.

Key Algorithms

Tree Traversal, Recursion

Edge Cases

Empty tree, single node, many children.

```python
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children else []

class Codec:
    def serialize(self, root: Optional[Node]) -> str:
        if not root:
            return ""

        def preorder(node: Node) -> List[str]:
            result = [str(node.val), str(len(node.children))]
            for child in node.children:
                result.extend(preorder(child))
            return result
```

```python

        return ','.join(preorder(root))

    def deserialize(self, data: str) -> Optional[Node]:
        if not data:
            return None

        values = iter(data.split(','))

        def build() -> Node:
            val = int(next(values))
            child_count = int(next(values))

            node = Node(val)
            for _ in range(child_count):
                node.children.append(build())

            return node

        return build()
```

Complexity: Time $O(n)$, Space $O(n)$

74 Encode N-ary Tree to Binary Tree

Problem Description

Convert N-ary tree to binary tree and back.

Solution Approach

- First child becomes left child in binary.
- Siblings become right children chain.
- Decode reverses this process.
- Maintain parent-child relationships.

Key Algorithms

Tree Transformation

Edge Cases

Empty tree, single child, many children.

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Codec:
    def encode(self, root: Optional[Node]) -> Optional[TreeNode]:
        if not root:
            return None

        binary_root = TreeNode(root.val)

        if root.children:
            # First child becomes left child
            binary_root.left = self.encode(root.children[0])

            # Remaining children form right chain
            current = binary_root.left
            for i in range(1, len(root.children)):
                current.right = self.encode(root.children[i])
                current = current.right
```

```python

        return binary_root

    def decode(self, root: Optional[TreeNode]) -> Optional[Node]:
        if not root:
            return None

        n_ary_root = Node(root.val)

        # Traverse right chain to collect children
        current = root.left
        while current:
            n_ary_root.children.append(self.decode(current))
            current = current.right

        return n_ary_root
```

Complexity: Time $O(n)$, Space $O(h)$ where h = height

75 All O' one Data Structure

Problem Description

Data structure with 'inc', 'dec', 'getMaxKey', 'getMinKey' all $O(1)$.

Solution Approach

- Use doubly linked list of buckets (count -¿ keys).
- HashMap: key -¿ bucket node.
- Move keys between buckets on 'inc/dec'.
- Head/tail for quick min/max access.

Key Algorithms

Doubly Linked List, Hash Map

Edge Cases

Empty structure, single key, all same count.

```python
class Bucket:
    def __init__(self, count: int):
        self.count = count
        self.keys = set()
        self.prev = None
        self.next = None

class AllOne:
    def __init__(self):
        self.head = Bucket(float('-inf'))
        self.tail = Bucket(float('inf'))
        self.head.next = self.tail
        self.tail.prev = self.head
        self.key_bucket = {}

    def _add_bucket_after(self, bucket: Bucket, count: int) -> Bucket:
        new_bucket = Bucket(count)
        new_bucket.prev = bucket
        new_bucket.next = bucket.next
        bucket.next.prev = new_bucket
        bucket.next = new_bucket
        return new_bucket

    def _remove_bucket(self, bucket: Bucket) -> None:
        bucket.prev.next = bucket.next
        bucket.next.prev = bucket.prev
```

```python

    def inc(self, key: str) -> None:
        if key in self.key_bucket:
            bucket = self.key_bucket[key]
            bucket.keys.remove(key)

            # Find or create next bucket
            if bucket.next.count == bucket.count + 1:
                next_bucket = bucket.next
            else:
                next_bucket = self._add_bucket_after(bucket, bucket.count + 1)

            next_bucket.keys.add(key)
            self.key_bucket[key] = next_bucket

            # Remove empty bucket
            if not bucket.keys:
                self._remove_bucket(bucket)
        else:
            # New key
            if self.head.next.count == 1:
                bucket = self.head.next
            else:
                bucket = self._add_bucket_after(self.head, 1)

            bucket.keys.add(key)
            self.key_bucket[key] = bucket

    def dec(self, key: str) -> None:
        bucket = self.key_bucket[key]
        bucket.keys.remove(key)

        if bucket.count == 1:
            # Remove key completely
            del self.key_bucket[key]
        else:
            # Find or create previous bucket
            if bucket.prev.count == bucket.count - 1:
                prev_bucket = bucket.prev
            else:
                prev_bucket = self._add_bucket_after(bucket.prev, bucket.count - 1)

            prev_bucket.keys.add(key)
            self.key_bucket[key] = prev_bucket

        # Remove empty bucket
        if not bucket.keys:
            self._remove_bucket(bucket)

    def getMaxKey(self) -> str:
        if self.tail.prev == self.head:
            return ""
        return next(iter(self.tail.prev.keys))

    def getMinKey(self) -> str:
        if self.head.next == self.tail:
            return ""
        return next(iter(self.head.next.keys))
```

Complexity: Time $O(1)$ all operations, Space $O(n)$

76 K-th Smallest in Lexicographical Order

Problem Description

Find ‘k‘th smallest integer in lexicographical order from 1 to ‘n‘.

Solution Approach

- Count numbers with each prefix.

- Navigate tree of numbers lexicographically.
- Skip subtrees when count ¡ ‘k‘.
- Drill down when needed.

Key Algorithms

Tree Navigation, Counting

Edge Cases

‘k = 1‘, ‘k = n‘, large ‘n‘.

```python
def findKthNumber(n: int, k: int) -> int:
    def count_prefix(prefix: int, n: int) -> int:
        current = prefix
        next_prefix = prefix + 1
        count = 0

        while current <= n:
            count += min(n + 1, next_prefix) - current
            current *= 10
            next_prefix *= 10

        return count

    current = 1
    k -= 1  # 1-indexed to 0-indexed

    while k > 0:
        count = count_prefix(current, n)

        if count <= k:
            # Skip this subtree
            k -= count
            current += 1
        else:
            # Go deeper
            current *= 10
            k -= 1

    return current
```

Complexity: Time $O(\log^2 n)$, Space $O(1)$

77 Arithmetic Slices II - Subsequence

Problem Description

Count arithmetic subsequences of length ≥ 3.

Solution Approach

- Dynamic programming with hash maps.
- `dp[i][diff]` = count ending at ‘i‘ with difference ‘diff‘.
- For each pair, extend previous subsequences.
- Sum all counts ≥ 2 length.

Key Algorithms

Dynamic Programming, Hash Map

Edge Cases

All same elements, no valid subsequences, overflow.

```python
from collections import defaultdict

def numberOfArithmeticSlices(nums: List[int]) -> int:
    n = len(nums)
    dp = [defaultdict(int) for _ in range(n)]
    total = 0

    for i in range(1, n):
        for j in range(i):
            diff = nums[i] - nums[j]

            # Subsequences ending at j with difference diff
            count = dp[j][diff]

            # Add to result (these form valid subsequences when extended)
            total += count

            # Update dp[i][diff]
            # +1 for the new 2-element subsequence [j, i]
            dp[i][diff] += count + 1

    return total
```

Complexity: Time $O(n^2)$, Space $O(n^2)$

78 Poor Pigs

Problem Description

Minimum pigs to find poisonous bucket in given time.

Solution Approach

- Each pig can test `(minutesToTest/minutesToDie + 1)` states.
- With ‘d‘ dimensions (pigs), can test `statesd̂` buckets.
- Find minimum pigs where `statesp̂igs >= buckets`.
- Use logarithm to solve.

Key Algorithms

Math, Information Theory

Edge Cases

Only one test round, many buckets, exact power match.

```python
import math

def poorPigs(buckets: int, minutesToDie: int, minutesToTest: int) -> int:
    # Number of test rounds possible
    states = minutesToTest // minutesToDie + 1

    # Find minimum pigs where states^pigs >= buckets
    return math.ceil(math.log(buckets) / math.log(states))
```

Complexity: Time $O(1)$, Space $O(1)$

79 LFU Cache

Problem Description

Implement Least Frequently Used cache with $O(1)$ operations.

Solution Approach

- Use frequency buckets with doubly linked lists.
- Hash map: key -¿ node.
- Hash map: frequency -¿ bucket.
- Track minimum frequency.

Key Algorithms

Doubly Linked List, Hash Map

Edge Cases

Capacity 0, single element, ties in frequency.

```python
class Node:
    def __init__(self, key: int = 0, value: int = 0):
        self.key = key
        self.value = value
        self.freq = 1
        self.prev = None
        self.next = None

class DLinkedList:
    def __init__(self):
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail
        self.tail.prev = self.head
        self.size = 0

    def add_to_head(self, node: Node) -> None:
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node
        self.size += 1

    def remove_node(self, node: Node) -> None:
        node.prev.next = node.next
        node.next.prev = node.prev
        self.size -= 1

    def remove_tail(self) -> Node:
        if self.size == 0:
            return None
        tail_node = self.tail.prev
        self.remove_node(tail_node)
        return tail_node

class LFUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.min_freq = 0
        self.key_to_node = {}
        self.freq_to_list = defaultdict(DLinkedList)

    def get(self, key: int) -> int:
        if key not in self.key_to_node:
            return -1

        node = self.key_to_node[key]
        self._update_freq(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if self.capacity == 0:
            return

```

```
        if key in self.key_to_node:
            node = self.key_to_node[key]
            node.value = value
            self._update_freq(node)
        else:
            if len(self.key_to_node) >= self.capacity:
                # Remove LFU node
                min_freq_list = self.freq_to_list[self.min_freq]
                node_to_remove = min_freq_list.remove_tail()
                del self.key_to_node[node_to_remove.key]

            # Add new node
            new_node = Node(key, value)
            self.key_to_node[key] = new_node
            self.freq_to_list[1].add_to_head(new_node)
            self.min_freq = 1

    def _update_freq(self, node: Node) -> None:
        freq = node.freq
        self.freq_to_list[freq].remove_node(node)

        # Update min_freq if necessary
        if freq == self.min_freq and self.freq_to_list[freq].size == 0:
            self.min_freq += 1

        node.freq += 1
        self.freq_to_list[node.freq].add_to_head(node)
```

Complexity: Time $O(1)$ all operations, Space $O(\text{capacity})$

80 Optimal Account Balancing

Problem Description

Minimum transactions to settle all debts.

Solution Approach

- Calculate net balance for each person.
- Use backtracking to try all settlement orders.
- Skip people with 0 balance.
- Optimize by settling opposite sign balances.

Key Algorithms

Backtracking, Optimization

Edge Cases

No debts, all balanced, circular debts.

```
from collections import defaultdict

def minTransfers(transactions: List[List[int]]) -> int:
    # Calculate net balance
    balance = defaultdict(int)
    for from_person, to_person, amount in transactions:
        balance[from_person] -= amount
        balance[to_person] += amount

    # Get non-zero balances
    debts = [amount for amount in balance.values() if amount != 0]

    def dfs(start: int) -> int:
        # Skip settled debts
        while start < len(debts) and debts[start] == 0:
```

```python
            start += 1

        if start == len(debts):
            return 0

        min_trans = float('inf')

        # Try settling with each person after start
        for i in range(start + 1, len(debts)):
            # Only settle with opposite sign
            if debts[start] * debts[i] < 0:
                # Settle debt
                debts[i] += debts[start]
                min_trans = min(min_trans, 1 + dfs(start + 1))
                # Backtrack
                debts[i] -= debts[start]

        return min_trans

    return dfs(0)
```

Complexity: Time $O(n!)$, Space $O(n)$

81 Count The Repetitions

Problem Description

How many times ‘s2‘ appears as subsequence in ‘n1‘ repetitions of ‘s1‘.

Solution Approach

- Find pattern cycle in matching.
- Track position in ‘s2‘ after each ‘s1‘.
- Detect when pattern repeats.
- Calculate full cycles and remainder.

Key Algorithms

Pattern Detection, Cycle Finding

Edge Cases

No match possible, ‘s2‘ longer than repeated ‘s1‘.

```python
def getMaxRepetitions(s1: str, n1: int, s2: str, n2: int) -> int:
    if n1 == 0:
        return 0

    # Track s2 index and count after each s1
    s1_count = 0
    s2_count = 0
    s2_idx = 0

    # For cycle detection
    seen = {}

    while s1_count < n1:
        # Process one s1
        for char in s1:
            if char == s2[s2_idx]:
                s2_idx += 1
                if s2_idx == len(s2):
                    s2_idx = 0
                    s2_count += 1

        s1_count += 1
```

```python

        # Check for cycle
        if s2_idx in seen:
            # Found cycle
            prev_s1_count, prev_s2_count = seen[s2_idx]

            # Length of cycle
            cycle_s1 = s1_count - prev_s1_count
            cycle_s2 = s2_count - prev_s2_count

            # Complete cycles remaining
            remaining_cycles = (n1 - s1_count) // cycle_s1
            s2_count += remaining_cycles * cycle_s2
            s1_count += remaining_cycles * cycle_s1

            # Process remainder
            for _ in range(n1 - s1_count):
                for char in s1:
                    if char == s2[s2_idx]:
                        s2_idx += 1
                        if s2_idx == len(s2):
                            s2_idx = 0
                            s2_count += 1

            break

        seen[s2_idx] = (s1_count, s2_count)

    return s2_count // n2
```

Complexity: Time $O(\text{len}(s1) \times \text{len}(s2))$, Space $O(\text{len}(s2))$

82 Encode String with Shortest Length

Problem Description

Encode string using 'k[encoded]' format for shortest result.

Solution Approach

- Dynamic programming with substring encoding.
- Try all possible splits.
- Check if substring can be compressed.
- Memoize optimal encodings.

Key Algorithms

Dynamic Programming, String Compression

Edge Cases

No compression possible, nested patterns.

```python
def encode(s: str) -> str:
    n = len(s)
    dp = [[''] * n for _ in range(n)]

    for length in range(1, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            substr = s[i:j + 1]

            # Try no encoding
            dp[i][j] = substr

            # Try encoding as k[pattern]
```

```python
            for k in range(1, length):
                if length % k == 0:
                    pattern = substr[:k]
                    if pattern * (length // k) == substr:
                        encoded = f"{length // k}[{dp[i][i + k - 1]}]"
                        if len(encoded) < len(dp[i][j]):
                            dp[i][j] = encoded

            # Try splitting
            for k in range(i, j):
                split = dp[i][k] + dp[k + 1][j]
                if len(split) < len(dp[i][j]):
                    dp[i][j] = split

    return dp[0][n - 1]
```

Complexity: Time $O(n^3)$, Space $O(n^2)$

83 Concatenated Words

Problem Description

Find words that are concatenation of shorter words in list.

Solution Approach

- Sort words by length.
- Use dynamic programming or DFS.
- Check if word can be formed from shorter words.
- Use Trie or set for efficient lookup.

Key Algorithms

Dynamic Programming, Trie

Edge Cases

Empty strings, single character words, no concatenations.

```python
def findAllConcatenatedWordsInADict(words: List[str]) -> List[str]:
    word_set = set(words)
    result = []

    def can_form(word: str, start: int, count: int) -> bool:
        if start == len(word):
            return count > 1

        for end in range(start + 1, len(word) + 1):
            if word[start:end] in word_set:
                if can_form(word, end, count + 1):
                    return True

        return False

    for word in words:
        if can_form(word, 0, 0):
            result.append(word)

    return result
```

Complexity: Time $O(n \times m^3)$ where m = max word length, Space $O(n)$

84 Largest Palindrome Product

Problem Description

Largest palindrome made from product of two ‘n‘-digit numbers.

Solution Approach

- Start from largest possible palindrome.
- Check if can be factored into two ‘n‘-digit numbers.
- Build palindrome from first half.
- Special case for ‘n = 1‘.

Key Algorithms

Math, Palindrome Construction

Edge Cases

‘n = 1‘, no valid palindrome (impossible here).

```python
def largestPalindrome(n: int) -> int:
    if n == 1:
        return 9

    upper = 10**n - 1
    lower = 10**(n - 1)

    # Try palindromes in decreasing order
    for i in range(upper, lower - 1, -1):
        # Build palindrome
        palindrome = int(str(i) + str(i)[::-1])

        # Check if can be factored
        j = upper
        while j * j >= palindrome:
            if palindrome % j == 0 and palindrome // j <= upper:
                return palindrome % 1337
            j -= 1

    return -1
```

Complexity: Time $O(10^n)$, Space $O(1)$

85 Sliding Window Median

Problem Description

Find median of each window of size ‘k‘.

Solution Approach

- Use two heaps (like median stream).
- Handle window sliding with removal.
- Use multiset or heap with lazy deletion.
- Balance heaps after each operation.

Key Algorithms

Two Heaps, Sliding Window

Edge Cases

'k = 1', even/odd 'k', duplicates.

```python
from sortedcontainers import SortedList

def medianSlidingWindow(nums: List[int], k: int) -> List[float]:
    window = SortedList()
    result = []

    for i, num in enumerate(nums):
        # Add to window
        window.add(num)

        # Remove element outside window
        if i >= k:
            window.remove(nums[i - k])

        # Calculate median when window is full
        if i >= k - 1:
            if k % 2 == 1:
                result.append(float(window[k // 2]))
            else:
                result.append((window[k // 2 - 1] + window[k // 2]) / 2)

    return result
```

Complexity: Time $O(n \log k)$, Space $O(k)$

86 Smallest Good Base

Problem Description

Find smallest base where 'n' is all 1s in that base.

Solution Approach

- For `m` digits of 1s: $n = 1 + k + k^2 + \cdots + k^{m-1}$.
- Try different values of 'm' (digits).
- Binary search for base 'k'.
- Check if forms valid representation.

Key Algorithms

Binary Search, Math

Edge Cases

'n = 3' (base 2), prime numbers.

```python
def smallestGoodBase(n: str) -> str:
    n = int(n)

    # Try different lengths of 1s
    for m in range(64, 1, -1):
        # Binary search for base
        left, right = 2, int(n**(1/(m-1))) + 1

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

            # Calculate sum of geometric series
            sum_val = 0
            for i in range(m):
                sum_val = sum_val * mid + 1
                if sum_val > n:
```

```python
                    break

            if sum_val == n:
                return str(mid)
            elif sum_val < n:
                left = mid + 1
            else:
                right = mid

    return str(n - 1)
```

Complexity: Time $O(\log^2 n)$, Space $O(1)$

87 Zuma Game

Problem Description

Minimum balls to remove all balls by matching 3+ consecutive.

Solution Approach

- Use backtracking with memoization.
- Try inserting each ball at each position.
- Remove consecutive groups of 3+.
- Prune impossible states.

Key Algorithms

Backtracking, String Manipulation

Edge Cases

Already removable, impossible to clear.

```python
from functools import lru_cache
from collections import Counter

def findMinStep(board: str, hand: str) -> int:
    def remove_consecutive(s: str) -> str:
        i = 0
        while i < len(s):
            j = i
            while j < len(s) and s[j] == s[i]:
                j += 1
            if j - i >= 3:
                return remove_consecutive(s[:i] + s[j:])
            i = j
        return s

    @lru_cache(None)
    def dfs(board: str, hand: tuple) -> int:
        board = remove_consecutive(board)
        if not board:
            return 0
        if not hand:
            return float('inf')

        hand_count = Counter(hand)
        min_balls = float('inf')

        for i in range(len(board) + 1):
            for color in hand_count:
                if hand_count[color] > 0:
                    # Try inserting ball at position i
                    new_board = board[:i] + color + board[i:]
                    new_hand = list(hand)
```

```python
                    new_hand.remove(color)

                    result = 1 + dfs(new_board, tuple(new_hand))
                    min_balls = min(min_balls, result)

        return min_balls

    result = dfs(board, tuple(hand))
    return result if result != float('inf') else -1
```

Complexity: Time $O((m+n)^m \times n!)$ where m = board length, n = hand size, Space $O(mn)$

88 Robot Room Cleaner

Problem Description

Clean entire room with robot using only relative movement.

Solution Approach

- DFS with backtracking.
- Track visited cells relative to start.
- Try all 4 directions.
- Return to original position/direction.

Key Algorithms

DFS, Backtracking

Edge Cases

Single cell, complex obstacles.

```python
class Solution:
    def cleanRoom(self, robot):
        visited = set()
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # right, down, left, up

        def go_back():
            robot.turnRight()
            robot.turnRight()
            robot.move()
            robot.turnRight()
            robot.turnRight()

        def dfs(x: int, y: int, direction: int) -> None:
            visited.add((x, y))
            robot.clean()

            # Try all 4 directions
            for i in range(4):
                new_direction = (direction + i) % 4
                dx, dy = directions[new_direction]
                nx, ny = x + dx, y + dy

                if (nx, ny) not in visited and robot.move():
                    dfs(nx, ny, new_direction)
                    go_back()

                robot.turnRight()

        dfs(0, 0, 0)
```

Complexity: Time $O(n)$ where n = accessible cells, Space $O(n)$

89 Reverse Pairs

Problem Description

Count pairs where ‘i ¡ j‘ and ‘nums[i] ¿ 2 * nums[j]‘.

Solution Approach

- Modified merge sort.
- Count during merge process.
- Similar to count inversions.
- Handle overflow with long comparison.

Key Algorithms

Merge Sort, Divide and Conquer

Edge Cases

No reverse pairs, all same values, overflow.

```python
def reversePairs(nums: List[int]) -> int:
    def merge_sort(start: int, end: int) -> int:
        if end - start <= 1:
            return 0

        mid = (start + end) // 2
        count = merge_sort(start, mid) + merge_sort(mid, end)

        # Count reverse pairs
        j = mid
        for i in range(start, mid):
            while j < end and nums[i] > 2 * nums[j]:
                j += 1
            count += j - mid

        # Merge sorted halves
        temp = []
        i, j = start, mid
        while i < mid and j < end:
            if nums[i] <= nums[j]:
                temp.append(nums[i])
                i += 1
            else:
                temp.append(nums[j])
                j += 1

        temp.extend(nums[i:mid])
        temp.extend(nums[j:end])
        nums[start:end] = temp

        return count

    return merge_sort(0, len(nums))
```

Complexity: Time $O(n \log n)$, Space $O(n)$

90 The Maze III

Problem Description

Find path to hole with shortest distance and lexicographically smallest.

Solution Approach

- BFS/Dijkstra with priority queue.
- Priority: distance, then path string.
- Roll ball until wall or hole.
- Track visited with direction.

Key Algorithms

Dijkstra's Algorithm, BFS

Edge Cases

Ball starts at hole, no path, multiple shortest paths.

```python
import heapq

def findShortestWay(maze: List[List[int]], ball: List[int], hole: List[int]) -> str:
    m, n = len(maze), len(maze[0])
    directions = [(1, 0, 'd'), (0, -1, 'l'), (0, 1, 'r'), (-1, 0, 'u')]

    # Priority queue: (distance, path, x, y)
    heap = [(0, '', ball[0], ball[1])]
    visited = set()

    while heap:
        dist, path, x, y = heapq.heappop(heap)

        if (x, y) in visited:
            continue
        visited.add((x, y))

        if [x, y] == hole:
            return path

        for dx, dy, direction in directions:
            nx, ny, steps = x, y, 0

            # Roll until wall or hole
            while 0 <= nx + dx < m and 0 <= ny + dy < n and maze[nx + dx][ny + dy] == 0:
                nx += dx
                ny += dy
                steps += 1
                if [nx, ny] == hole:
                    break

            if (nx, ny) not in visited:
                heapq.heappush(heap, (dist + steps, path + direction, nx, ny))

    return "impossible"
```

Complexity: Time $O(mn \times \max(m, n))$, Space $O(mn)$

91 IPO

Problem Description

Maximize capital by selecting 'k' projects with limited initial capital.

Solution Approach

- Sort projects by capital requirement.
- Use max heap for available projects' profits.
- Greedily select highest profit available.
- Update available projects after each selection.

Key Algorithms

Two Heaps, Greedy

Edge Cases

'k ¿' number of projects, insufficient capital.

```
import heapq

def findMaximizedCapital(k: int, w: int, profits: List[int], capital: List[int]) -> int:
    # Projects sorted by capital requirement
    projects = sorted(zip(capital, profits))

    # Max heap for available projects' profits
    available = []
    i = 0

    for _ in range(k):
        # Add all newly available projects
        while i < len(projects) and projects[i][0] <= w:
            heapq.heappush(available, -projects[i][1])
            i += 1

        if not available:
            break

        # Select project with maximum profit
        w += -heapq.heappop(available)

    return w
```

Complexity: Time $O(n \log n)$, Space $O(n)$

92 Freedom Trail

Problem Description

Minimum steps to spell key by rotating ring.

Solution Approach

- Dynamic programming with states (ring_pos, key_index).
- For each character, try all matching positions.
- Calculate rotation distance (clockwise vs counter).
- Add 1 for button press.

Key Algorithms

Dynamic Programming

Edge Cases

Single character, key longer than ring.

```
from functools import lru_cache

def findRotateSteps(ring: str, key: str) -> int:
    n = len(ring)

    # Precompute character positions
    char_positions = defaultdict(list)
    for i, char in enumerate(ring):
        char_positions[char].append(i)

    @lru_cache(None)
```

```python
    def dp(ring_pos: int, key_idx: int) -> int:
        if key_idx == len(key):
            return 0

        min_steps = float('inf')

        for next_pos in char_positions[key[key_idx]]:
            # Calculate rotation distance
            dist = abs(ring_pos - next_pos)
            dist = min(dist, n - dist)

            # 1 for button press + rotation + remaining
            steps = 1 + dist + dp(next_pos, key_idx + 1)
            min_steps = min(min_steps, steps)

        return min_steps

    return dp(0, 0)
```

Complexity: Time $O(n^2 \times m)$ where m = key length, Space $O(nm)$

93 Super Washing Machines

Problem Description

Minimum moves to redistribute dresses evenly among machines.

Solution Approach

- Check if total divisible by 'n'.
- Calculate required flow through each position.
- Maximum of: max give away, max absolute flow.
- Can't split dress, so some positions bottleneck.

Key Algorithms

Math, Prefix Sum

Edge Cases

Already balanced, impossible to balance.

```python
def findMinMoves(machines: List[int]) -> int:
    total = sum(machines)
    n = len(machines)

    if total % n != 0:
        return -1

    target = total // n
    moves = 0
    balance = 0

    for dresses in machines:
        balance += dresses - target
        # Max between: current imbalance, dresses to give away
        moves = max(moves, abs(balance), dresses - target)

    return moves
```

Complexity: Time $O(n)$, Space $O(1)$

94 Word Abbreviation

Problem Description

Abbreviate words uniquely with shortest abbreviations.

Solution Approach

- Start with shortest abbreviation for each word.
- Resolve conflicts by increasing prefix length.
- Group by abbreviation and resolve duplicates.
- Ensure abbreviation is shorter than original.

Key Algorithms

Hash Map, String Manipulation

Edge Cases

No abbreviation shorter, unique prefixes needed.

```python
def wordsAbbreviation(words: List[str]) -> List[str]:
    def abbrev(word: str, prefix_len: int) -> str:
        if len(word) - prefix_len <= 2:
            return word
        return word[:prefix_len + 1] + str(len(word) - prefix_len - 2) + word[-1]

    n = len(words)
    result = [''] * n
    prefix_len = [0] * n

    for i in range(n):
        result[i] = abbrev(words[i], 0)

    # Resolve conflicts
    duplicates = True
    while duplicates:
        duplicates = False
        groups = defaultdict(list)

        for i in range(n):
            groups[result[i]].append(i)

        for indices in groups.values():
            if len(indices) > 1:
                duplicates = True
                for i in indices:
                    prefix_len[i] += 1
                    result[i] = abbrev(words[i], prefix_len[i])

    return result
```

Complexity: Time $O(n^2 \times L)$ worst case where L = max length, Space $O(n)$

95 Remove Boxes

Problem Description

Maximum points removing continuous boxes of same color.

Solution Approach

- 3D DP: `dp[l][r][k]` = max points removing `boxes[l:r+1]` with `k` same-colored boxes to right of `r`.
- Try removing `boxes[r]` with `k` boxes.
- Try merging with same color in middle.
- Memoization for efficiency.

Key Algorithms

Dynamic Programming, Interval DP

Edge Cases

All same color, alternating colors.

```python
from functools import lru_cache

def removeBoxes(boxes: List[int]) -> int:
    @lru_cache(None)
    def dp(l: int, r: int, k: int) -> int:
        if l > r:
            return 0

        # Optimization: combine consecutive same colors
        while l < r and boxes[r] == boxes[r - 1]:
            r -= 1
            k += 1

        # Option 1: Remove boxes[r] with k boxes
        result = dp(l, r - 1, 0) + (k + 1) ** 2

        # Option 2: Find same color in [l, r-1] and merge
        for i in range(l, r):
            if boxes[i] == boxes[r]:
                result = max(result, dp(l, i, k + 1) + dp(i + 1, r - 1, 0))

        return result

    return dp(0, len(boxes) - 1, 0)
```

Complexity: Time $O(n^4)$, Space $O(n^3)$

96 Split Array with Equal Sum

Problem Description

Find if array can be split at 'i,j,k' where 4 subarrays have equal sum.

Solution Approach

- Fix middle point 'j'.
- Find all valid 'i' on left with equal sums.
- Find all valid 'k' on right with equal sums.
- Check if any sum appears on both sides.

Key Algorithms

Hash Set, Prefix Sum

Edge Cases

Array too short, negative numbers.

```python
def splitArray(nums: List[int]) -> bool:
    n = len(nums)
    if n < 7:
        return False

    # Compute prefix sums
    prefix = [0]
    for num in nums:
        prefix.append(prefix[-1] + num)

    # Try each j (middle split)
    for j in range(3, n - 3):
        left_sums = set()

        # Find valid i values on left
        for i in range(1, j - 1):
            sum1 = prefix[i]
            sum2 = prefix[j] - prefix[i + 1]
            if sum1 == sum2:
                left_sums.add(sum1)

        # Find valid k values on right
        for k in range(j + 2, n - 1):
            sum3 = prefix[k] - prefix[j + 1]
            sum4 = prefix[n] - prefix[k + 1]
            if sum3 == sum4 and sum3 in left_sums:
                return True

    return False
```

Complexity: Time $O(n^2)$, Space $O(n)$

97 Student Attendance Record II

Problem Description

Count valid attendance records with at most 1 'A' and 2 consecutive 'L'.

Solution Approach

- Dynamic programming with states.
- Track: position, number of 'A's, consecutive 'L's.
- Three choices at each position: 'P', 'A', 'L'.
- Use modulo for large numbers.

Key Algorithms

Dynamic Programming

Edge Cases

'n = 1', very large 'n'.

```python
def checkRecord(n: int) -> int:
    MOD = 10**9 + 7

    # dp[i][j][k] = count of valid records of length i with j 'A's and ending with k consecutive 'L's
    # Space optimization: only need previous state
    prev = [[0] * 3 for _ in range(2)]
    prev[0][0] = 1

```

```
    for i in range(n):
        curr = [[0] * 3 for _ in range(2)]

        for j in range(2):  # Number of 'A's
            for k in range(3):  # Consecutive 'L's at end
                # Add 'P'
                curr[j][0] = (curr[j][0] + prev[j][k]) % MOD

                # Add 'A'
                if j < 1:
                    curr[j + 1][0] = (curr[j + 1][0] + prev[j][k]) % MOD

                # Add 'L'
                if k < 2:
                    curr[j][k + 1] = (curr[j][k + 1] + prev[j][k]) % MOD

        prev = curr

    # Sum all valid states
    result = 0
    for j in range(2):
        for k in range(3):
            result = (result + prev[j][k]) % MOD

    return result
```

Complexity: Time $O(n)$, Space $O(1)$

98 Find the Closest Palindrome

Problem Description

Find closest palindrome number (not equal to ‘n‘).

Solution Approach

- Consider special cases: ‘999...9‘, ‘100...01‘.
- Get palindrome by mirroring first half.
- Try increment/decrement middle digit(s).
- Compare all candidates.

Key Algorithms

String Manipulation, Math

Edge Cases

Single digit, all 9s, ‘10...01‘.

```
def nearestPalindromic(n: str) -> str:
    length = len(n)
    num = int(n)

    # Special cases
    candidates = []
    candidates.append(10**length + 1)  # 100...001
    candidates.append(10**(length - 1) - 1)  # 999...999

    # Get prefix (first half)
    is_odd = length % 2
    mid = length // 2
    prefix = int(n[:mid + is_odd])

    # Generate palindromes by changing middle digit(s)
    for delta in [-1, 0, 1]:
        new_prefix = prefix + delta
```

```python
        palin = str(new_prefix)

        if is_odd:
            palin = palin + palin[-2::-1]
        else:
            palin = palin + palin[::-1]

        candidates.append(int(palin))

    # Find closest different from n
    min_diff = float('inf')
    result = 0

    for cand in candidates:
        if cand != num:
            diff = abs(cand - num)
            if diff < min_diff or (diff == min_diff and cand < result):
                min_diff = diff
                result = cand

    return str(result)
```

Complexity: Time $O(1)$, Space $O(1)$

99 Maximum Vacation Days

Problem Description

Maximize vacation days with flight constraints.

Solution Approach

- Dynamic programming: `dp[week][city]` = max days.
- For each week, try all reachable cities.
- Add vacation days for that city/week.
- Handle flight connectivity.

Key Algorithms

Dynamic Programming

Edge Cases

No flights, single city, all flights available.

```python
def maxVacationDays(flights: List[List[int]], days: List[List[int]]) -> int:
    n = len(flights)  # Number of cities
    k = len(days[0])  # Number of weeks

    # dp[city] = max vacation days ending at city
    dp = [-1] * n
    dp[0] = 0  # Start at city 0

    for week in range(k):
        new_dp = [-1] * n

        for dest in range(n):
            for src in range(n):
                # Can stay or fly from src to dest
                if dp[src] != -1 and (src == dest or flights[src][dest]):
                    new_dp[dest] = max(new_dp[dest], dp[src] + days[dest][week])

        dp = new_dp

    return max(dp)
```

Complexity: Time $O(n^2 \times k)$, Space $O(n)$

100 Find Median Given Frequency of Numbers

Problem Description

Find median from numbers table with frequencies.

Solution Approach

- Calculate total count and median position.
- Use cumulative sum with window function.
- Find number where cumulative sum crosses median position.
- Handle even count (average of two middle values).

Key SQL

Window Functions, Cumulative Sum

Edge Cases

Even/odd total count, single number.

```sql
WITH cumulative AS (
    SELECT
        number,
        frequency,
        SUM(frequency) OVER (ORDER BY number) AS cum_sum,
        SUM(frequency) OVER (ORDER BY number) - frequency AS prev_cum_sum
    FROM Numbers
),
total AS (
    SELECT SUM(frequency) AS total_count
    FROM Numbers
)
SELECT
    AVG(number) AS median
FROM cumulative, total
WHERE
    (total_count % 2 = 1 AND prev_cum_sum < (total_count + 1) / 2 AND cum_sum >= (
    total_count + 1) / 2)
    OR
    (total_count % 2 = 0 AND (prev_cum_sum < total_count / 2 AND cum_sum >= total_count
    / 2)
                              OR (prev_cum_sum < total_count / 2 + 1 AND cum_sum >=
    total_count / 2 + 1))
```

Complexity: Time $O(n \log n)$, Space $O(n)$