**100 LeetCode Hard Problems Study Guide**

**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**:

1. Use binary search on the smaller array
2. Find partition points in both arrays
3. Ensure elements on left ≤ elements on right
4. 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

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**:

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

**Key Algorithms**: Dynamic Programming

**Edge Cases**: Empty pattern/string, consecutive '*', '.*' matching everything

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**:

1. Use min-heap to track smallest elements
2. Push first element of each list to heap
3. Pop minimum, add to result, push next from same list
4. 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**:

1. Count nodes to ensure enough for reversal
2. Reverse k nodes at a time
3. Connect reversed groups properly
4. 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

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**:

1. Use sliding window with hashmap
2. Window size = len(words) \* len(words[0])
3. Check each possible starting position
4. Verify word counts match expected

**Key Algorithms**: Sliding Window, Hash Map

**Edge Cases**: Duplicate words, overlapping results, single word

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 = []

# 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 \* m \* 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**:

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

**Key Algorithms**: Stack

**Edge Cases**: All open/close parentheses, empty string, nested parentheses

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**:

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

**Key Algorithms**: Backtracking, Constraint Satisfaction

**Edge Cases**: Invalid initial board, multiple solutions (return any)

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**:

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

**Key Algorithms**: Array Manipulation, Cyclic Sort

**Edge Cases**: All negative, all > n, duplicates, [1,2,3,...,n]

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**:

1. Use two pointers from both ends
2. Track max height seen from left and right
3. Water trapped = min(left\_max, right\_max) - current height
4. Move pointer with smaller max inward

**Key Algorithms**: Two Pointers

**Edge Cases**: Monotonic array, no water trapped, single peak

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**:

1. Use dynamic programming
2. dp[i][j] = true if s[0:i] matches p[0:j]
3. '\*' can match empty or any sequence
4. '?' matches exactly one character

**Key Algorithms**: Dynamic Programming

**Edge Cases**: Multiple '*', leading/trailing '*', empty pattern/string

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):

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**:

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

**Key Algorithms**: Backtracking

**Edge Cases**: N = 1, N = 2,3 (no solution)

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**:

1. Similar to N-Queens but count solutions instead of storing boards
2. Use backtracking with pruning
3. 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**:

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

**Key Algorithms**: Math, Factorial Number System

**Edge Cases**: k = 1 (first permutation), k = n! (last permutation)

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(n²), Space O(n)

**14. Valid Number**

**Problem Description**: Validate if string is a valid decimal number.

**Solution Approach**:

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

**Key Algorithms**: String Parsing, State Machine

**Edge Cases**: Leading/trailing spaces, multiple signs/decimals, 'e' without digits

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**:

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

**Key Algorithms**: Greedy, String Manipulation

**Edge Cases**: Single word per line, last line, one word exceeds maxWidth

def fullJustify(words: List[str], maxWidth: int) -> List[str]:

result = []

current\_line = []

current\_length = 0

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**:

1. Use sliding window with two pointers
2. Expand window until all chars included
3. Contract window while maintaining validity
4. Track minimum window seen

**Key Algorithms**: Sliding Window, Hash Map

**Edge Cases**: No valid window, t longer than s, duplicate chars in t

from collections import Counter

def minWindow(s: str, t: str) -> str:

if not s or not t:

return ""

# 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**:

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

**Key Algorithms**: Stack, Monotonic Stack

**Edge Cases**: All same height, strictly increasing/decreasing, single bar

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**:

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

**Key Algorithms**: Dynamic Programming, Stack

**Edge Cases**: All 0s, all 1s, single row/column

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))

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(rows × cols), Space O(cols)

**19. Scramble String**

**Problem Description**: Check if s2 is scrambled string of s1 (binary tree scrambling).

**Solution Approach**:

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

**Key Algorithms**: Recursion, Memoization

**Edge Cases**: Same strings, single character, anagrams

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(n⁴), Space O(n³)

**20. Distinct Subsequences**

**Problem Description**: Count distinct subsequences of s that equal t.

**Solution Approach**:

1. Use dynamic programming
2. dp[i][j] = ways to form t[0:j] from s[0:i]
3. If s[i-1] == t[j-1]: can use or skip s[i-1]
4. 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**:

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

**Key Algorithms**: State Machine, Dynamic Programming

**Edge Cases**: Prices always decrease, single price, need only 1 transaction

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**:

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

**Key Algorithms**: DFS, Tree Traversal

**Edge Cases**: All negative values, single node, straight line tree

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**:

1. BFS to find shortest path length
2. Build adjacency graph during BFS
3. DFS/backtrack to find all paths of shortest length
4. Prune paths that can't reach end in time

**Key Algorithms**: BFS, DFS, Graph

**Edge Cases**: No path exists, multiple shortest paths, beginWord = endWord

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)

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×L²) where N = words, L = length, Space O(N×L)

**24. Word Ladder**

**Problem Description**: Find length of shortest transformation sequence from beginWord to endWord.

**Solution Approach**:

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

**Key Algorithms**: BFS, Hash Set

**Edge Cases**: No path, beginWord = endWord, endWord not in list

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×L²×26), Space O(N×L)

**25. Palindrome Partitioning II**

**Problem Description**: Minimum cuts needed to partition string into palindromes.

**Solution Approach**:

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

**Key Algorithms**: Dynamic Programming

**Edge Cases**: Already palindrome (0 cuts), single chars, no palindromes > 1

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

# 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(n²), Space O(n²)

**26. Candy**

**Problem Description**: Minimum candies to give children where higher rating gets more candy than neighbors.

**Solution Approach**:

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

**Key Algorithms**: Greedy, Two Pass

**Edge Cases**: All same ratings, strictly increasing/decreasing

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**:

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

**Key Algorithms**: Backtracking, Memoization

**Edge Cases**: No valid breaks, multiple ways to break, overlapping words

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(n³), Space O(n³)

**28. Max Points on a Line**

**Problem Description**: Maximum number of points on the same straight line.

**Solution Approach**:

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

**Key Algorithms**: Hash Map, GCD

**Edge Cases**: Duplicate points, vertical lines, all points collinear

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(n²), Space O(n)

**29. Find Minimum in Rotated Sorted Array II**

**Problem Description**: Find minimum in rotated sorted array with duplicates.

**Solution Approach**:

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

**Key Algorithms**: Binary Search

**Edge Cases**: No rotation, all duplicates, single element

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**:

1. Maintain internal buffer between calls
2. Read from buffer first if available
3. Call read4 to refill buffer when needed
4. Handle partial reads correctly

**Key Algorithms**: Buffer Management

**Edge Cases**: Multiple calls, buffer boundary, EOF handling

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**:

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

**Key Algorithms**: Dynamic Programming

**Edge Cases**: All positive values, large negative values, single cell

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**:

1. If k >= n/2, unlimited transactions
2. dp[i][j] = max profit with at most i transactions by day j
3. Track max profit after buying for each transaction
4. 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**:

1. Build Trie from word list
2. DFS from each cell
3. Track current path in Trie
4. 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

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×n×4^L) where L = max word length, Space O(total chars)

**34. Shortest Palindrome**

**Problem Description**: Find shortest palindrome by adding characters in front.

**Solution Approach**:

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

**Key Algorithms**: KMP Algorithm, String Matching

**Edge Cases**: Already palindrome, single character, no palindrome prefix

def shortestPalindrome(s: str) -> str:

if not s:

return s

# Create combined string for KMP

combined = s + '#' + s[::-1]

# 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**:

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

**Key Algorithms**: Sweep Line, Priority Queue

**Edge Cases**: Overlapping buildings, same height, touching buildings

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(n² 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**:

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

**Key Algorithms**: Sliding Window, Bucket Sort

**Edge Cases**: valueDiff = 0, indexDiff = 0, negative numbers

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, indexDiff))

**37. Basic Calculator**

**Problem Description**: Evaluate expression with +, -, (, ).

**Solution Approach**:

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

**Key Algorithms**: Stack

**Edge Cases**: Nested parentheses, negative numbers, spaces

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**:

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

**Key Algorithms**: Math, Digital DP

**Edge Cases**: n = 0, single digit numbers, powers of 10

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**:

1. Use deque to maintain decreasing order
2. Remove elements outside window
3. Remove smaller elements from right
4. Front of deque is always maximum

**Key Algorithms**: Deque, Monotonic Queue

**Edge Cases**: k = 1, k = n, all same values

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**:

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

**Key Algorithms**: Recursion, String Building

**Edge Cases**: Single digit range, low > high, leading zeros

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 \

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**:

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

**Key Algorithms**: Dynamic Programming

**Edge Cases**: k = 1 (impossible if n > 1), n = 1, all costs same

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**:

1. Build graph from adjacent word comparisons
2. Find first differing character pairs
3. Topological sort using DFS
4. 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(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**:

1. Inorder traversal to get sorted values
2. Use two pointers or binary search to find closest
3. Expand window to get k values
4. 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**:

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

**Key Algorithms**: String Manipulation

**Edge Cases**: 0, powers of 10, teens, exact thousands/millions

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**:

1. Backtracking with current expression and value
2. Track previous operand for multiplication
3. Try splitting at each position
4. Handle leading zeros

**Key Algorithms**: Backtracking

**Edge Cases**: Leading zeros, overflow, single digit

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**:

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

**Key Algorithms**: Two Heaps

**Edge Cases**: Single element, even/odd count

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**:

1. Median minimizes sum of absolute deviations
2. Find median of x-coordinates and y-coordinates separately
3. Meeting point is (median\_x, median\_y)
4. Calculate total distance

**Key Algorithms**: Math, Median

**Edge Cases**: Single person, all in line, grid boundaries

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()

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**:

1. Use preorder traversal for serialization
2. Use delimiter between values
3. Use special marker for null nodes
4. Deserialize using queue or recursion

**Key Algorithms**: Tree Traversal, String Parsing

**Edge Cases**: Empty tree, single node, unbalanced tree

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**:

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

**Key Algorithms**: BFS

**Edge Cases**: Already valid, no valid possible, multiple solutions

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**:

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

**Key Algorithms**: Binary Search

**Edge Cases**: Single pixel, full grid black, scattered pixels

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**:

1. Use Union-Find data structure
2. For each land addition, check 4 neighbors
3. Union with neighboring lands
4. 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 × α(mn)) where k = operations, Space O(k)

**52. Burst Balloons**

**Problem Description**: Maximum coins by bursting balloons in optimal order.

**Solution Approach**:

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

**Key Algorithms**: Dynamic Programming, Interval DP

**Edge Cases**: Single balloon, all same values

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(n³), Space O(n²)

**53. Count of Smaller Numbers After Self**

**Problem Description**: Count smaller elements to the right of each element.

**Solution Approach**:

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

**Key Algorithms**: Merge Sort, Binary Indexed Tree

**Edge Cases**: Sorted array, reverse sorted, duplicates

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**:

1. BFS from each building
2. Track total distance to each empty land
3. Track reachability count
4. 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(m²n²), Space O(mn)

**55. Create Maximum Number**

**Problem Description**: Create maximum number of length k from two arrays preserving order.

**Solution Approach**:

1. Try all combinations of taking i from nums1, k-i from nums2
2. Find maximum subsequence of given length
3. Merge two subsequences optimally
4. 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 × (m + n + k)), Space O(k)

**56. Count of Range Sum**

**Problem Description**: Count ranges where sum is in [lower, upper].

**Solution Approach**:

1. Use merge sort to count during merge
2. Convert to prefix sum problem
3. Count pairs where lower <= prefix[j] - prefix[i] <= upper
4. Use merge sort to efficiently count

**Key Algorithms**: Merge Sort, Prefix Sum

**Edge Cases**: Single element, all negative, empty range

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**:

1. DFS with memoization from each cell
2. Explore 4 directions with increasing values
3. Cache results to avoid recomputation
4. 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**:

1. Track maximum reachable number
2. If can't reach next number, add it as patch
3. When adding number x, extend range by x
4. 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:

# 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**:

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

**Key Algorithms**: DFS, Eulerian Path

**Edge Cases**: Multiple valid paths, cycles, dead ends

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**:

1. Check if current line crosses any of last 3-5 lines
2. Line i can only cross lines i-3, i-4, or i-5
3. Check intersection conditions for each case
4. 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**:

1. Use Trie or HashMap for efficient lookup
2. For each word, check if reverse exists
3. Check partial matches for different lengths
4. 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:]

# 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 × k²) where k = average word length, Space O(nk)

**62. Data Stream as Disjoint Intervals**

**Problem Description**: Maintain disjoint intervals from data stream.

**Solution Approach**:

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

**Key Algorithms**: Interval Merging, Binary Search

**Edge Cases**: Duplicate values, single point intervals

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**:

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

**Key Algorithms**: LIS, Binary Search

**Edge Cases**: Same dimensions, single envelope, all same width/height

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**:

1. Count character frequencies
2. Use max heap to get most frequent
3. Use queue to track cooling characters
4. 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 ≤ k.

**Solution Approach**:

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

**Key Algorithms**: Kadane's Algorithm, Binary Search

**Edge Cases**: All negative, k < all elements, single element = k

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(n² × 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**:

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

**Key Algorithms**: Hash Map, Array

**Edge Cases**: Remove non-existent, single element, all duplicates

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

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**:

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

**Key Algorithms**: Hash Set, Geometry

**Edge Cases**: Single rectangle, gaps, overlaps

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:

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**:

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

**Key Algorithms**: Dynamic Programming, Memoization

**Edge Cases**: No valid path, first jump must be 1, gaps too large

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(n²), Space O(n²)

**69. Trapping Rain Water II**

**Problem Description**: Trap rainwater in 2D elevation map.

**Solution Approach**:

1. Use min heap starting from boundaries
2. Process cells from lowest to highest
3. Water level determined by minimum boundary
4. 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**:

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

**Key Algorithms**: Binary Search, Greedy

**Edge Cases**: k = 1, k = n, all same values

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(sum)), Space O(1)

**71. Minimum Unique Word Abbreviation**

**Problem Description**: Find shortest abbreviation of target unique among dictionary.

**Solution Approach**:

1. Generate all possible abbreviations
2. Use bit manipulation for abbreviation patterns
3. Check uniqueness against dictionary
4. Return shortest unique abbreviation

**Key Algorithms**: Bit Manipulation, BFS

**Edge Cases**: No abbreviation needed, single character, all words same length

def minAbbreviation(target: str, dictionary: List[str]) -> str:

m = len(target)

diffs = []

# Pre-calculate difference masks

for word in dictionary:

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 × n), Space O(n)

**72. Word Squares**

**Problem Description**: Find all word squares from given words.

**Solution Approach**:

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

**Key Algorithms**: Backtracking, Trie/HashMap

**Edge Cases**: No valid squares, single letter words, all same word

from collections import defaultdict

def wordSquares(words: List[str]) -> List[List[str]]:

n = len(words[0])

prefix\_map = defaultdict(list)

# 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 × 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**:

1. Use preorder traversal with child count
2. Format: value,child\_count,children...
3. Recursively serialize each node
4. Deserialize by reading count and recursing

**Key Algorithms**: Tree Traversal, Recursion

**Edge Cases**: Empty tree, single node, many children

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

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**:

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

**Key Algorithms**: Tree Transformation

**Edge Cases**: Empty tree, single child, many children

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

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**:

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

**Key Algorithms**: Doubly Linked List, Hash Map

**Edge Cases**: Empty structure, single key, all same count

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

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 kth smallest integer in lexicographical order from 1 to n.

**Solution Approach**:

1. Count numbers with each prefix
2. Navigate tree of numbers lexicographically
3. Skip subtrees when count < k
4. Drill down when needed

**Key Algorithms**: Tree Navigation, Counting

**Edge Cases**: k = 1, k = n, large n

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(log² n), Space O(1)

**77. Arithmetic Slices II - Subsequence**

**Problem Description**: Count arithmetic subsequences of length ≥ 3.

**Solution Approach**:

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

**Key Algorithms**: Dynamic Programming, Hash Map

**Edge Cases**: All same elements, no valid subsequences, overflow

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(n²), Space O(n²)

**78. Poor Pigs**

**Problem Description**: Minimum pigs to find poisonous bucket in given time.

**Solution Approach**:

1. Each pig can test (minutesToTest/minutesToDie + 1) states
2. With d dimensions (pigs), can test states^d buckets
3. Find minimum pigs where states^pigs >= buckets
4. Use logarithm to solve

**Key Algorithms**: Math, Information Theory

**Edge Cases**: Only one test round, many buckets, exact power match

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**:

1. Use frequency buckets with doubly linked lists
2. Hash map: key -> node
3. Hash map: frequency -> bucket
4. Track minimum frequency

**Key Algorithms**: Doubly Linked List, Hash Map

**Edge Cases**: Capacity 0, single element, ties in frequency

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(capacity)

**80. Optimal Account Balancing**

**Problem Description**: Minimum transactions to settle all debts.

**Solution Approach**:

1. Calculate net balance for each person
2. Use backtracking to try all settlement orders
3. Skip people with 0 balance
4. 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:

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**:

1. Find pattern cycle in matching
2. Track position in s2 after each s1
3. Detect when pattern repeats
4. Calculate full cycles and remainder

**Key Algorithms**: Pattern Detection, Cycle Finding

**Edge Cases**: No match possible, s2 longer than repeated s1

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

# 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(len(s1) × len(s2)), Space O(len(s2))

**82. Encode String with Shortest Length**

**Problem Description**: Encode string using k[encoded] format for shortest result.

**Solution Approach**:

1. Dynamic programming with substring encoding
2. Try all possible splits
3. Check if substring can be compressed
4. Memoize optimal encodings

**Key Algorithms**: Dynamic Programming, String Compression

**Edge Cases**: No compression possible, nested patterns

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]

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(n³), Space O(n²)

**83. Concatenated Words**

**Problem Description**: Find words that are concatenation of shorter words in list.

**Solution Approach**:

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

**Key Algorithms**: Dynamic Programming, Trie

**Edge Cases**: Empty strings, single character words, no concatenations

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 × m³) 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**:

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

**Key Algorithms**: Math, Palindrome Construction

**Edge Cases**: n = 1, no valid palindrome (impossible here)

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**:

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

**Key Algorithms**: Two Heaps, Sliding Window

**Edge Cases**: k = 1, even/odd k, duplicates

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**:

1. For m digits of 1s: n = 1 + k + k² + ... + k^(m-1)
2. Try different values of m (digits)
3. Binary search for base k
4. Check if forms valid representation

**Key Algorithms**: Binary Search, Math

**Edge Cases**: n = 3 (base 2), prime numbers

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:

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(log² n), Space O(1)

**87. Zuma Game**

**Problem Description**: Minimum balls to remove all balls by matching 3+ consecutive.

**Solution Approach**:

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

**Key Algorithms**: Backtracking, String Manipulation

**Edge Cases**: Already removable, impossible to clear

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)

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 × 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**:

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

**Key Algorithms**: DFS, Backtracking

**Edge Cases**: Single cell, complex obstacles

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**:

1. Modified merge sort
2. Count during merge process
3. Similar to count inversions
4. Handle overflow with long comparison

**Key Algorithms**: Merge Sort, Divide and Conquer

**Edge Cases**: No reverse pairs, all same values, overflow

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**:

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

**Key Algorithms**: Dijkstra's Algorithm, BFS

**Edge Cases**: Ball starts at hole, no path, multiple shortest paths

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 × max(m,n)), Space O(mn)

**91. IPO**

**Problem Description**: Maximize capital by selecting k projects with limited initial capital.

**Solution Approach**:

1. Sort projects by capital requirement
2. Use max heap for available projects' profits
3. Greedily select highest profit available
4. 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**:

1. Dynamic programming with states (ring\_pos, key\_index)
2. For each character, try all matching positions
3. Calculate rotation distance (clockwise vs counter)
4. 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)

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(n² × m) where m = key length, Space O(nm)

**93. Super Washing Machines**

**Problem Description**: Minimum moves to redistribute dresses evenly among machines.

**Solution Approach**:

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

**Key Algorithms**: Math, Prefix Sum

**Edge Cases**: Already balanced, impossible to balance

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**:

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

**Key Algorithms**: Hash Map, String Manipulation

**Edge Cases**: No abbreviation shorter, unique prefixes needed

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(n² × 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**:

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

**Key Algorithms**: Dynamic Programming, Interval DP

**Edge Cases**: All same color, alternating colors

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(n⁴), Space O(n³)

**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**:

1. Fix middle point j
2. Find all valid i on left with equal sums
3. Find all valid k on right with equal sums
4. Check if any sum appears on both sides

**Key Algorithms**: Hash Set, Prefix Sum

**Edge Cases**: Array too short, negative numbers

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(n²), 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**:

1. Dynamic programming with states
2. Track: position, number of 'A's, consecutive 'L's
3. Three choices at each position: 'P', 'A', 'L'
4. Use modulo for large numbers

**Key Algorithms**: Dynamic Programming

**Edge Cases**: n = 1, very large n

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**:

1. Consider special cases: 999...9, 100...01
2. Get palindrome by mirroring first half
3. Try increment/decrement middle digit(s)
4. 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

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**:

1. Dynamic programming: dp[week][city] = max days
2. For each week, try all reachable cities
3. Add vacation days for that city/week
4. Handle flight connectivity

**Key Algorithms**: Dynamic Programming

**Edge Cases**: No flights, single city, all flights available

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(n² × k), Space O(n)

**100. Find Median Given Frequency of Numbers**

**Problem Description**: Find median from numbers table with frequencies.

**Solution Approach**:

1. Calculate total count and median position
2. Use cumulative sum with window function
3. Find number where cumulative sum crosses median position
4. Handle even count (average of two middle values)

**Key SQL**: Window Functions, Cumulative Sum

**Edge Cases**: Even/odd total count, single number

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)