

# Solutions to DSA Questions 142-170 (Backtracking, Design, Misc) For 1-2 Years

Experience Roles at EPAM Compiled on September 27, 2025

## Introduction

This document provides detailed solutions for 29 Data Structures and Algorithms (DSA) problems (questions 142 to 170) from the Backtracking, Design, and Miscellaneous (Sliding Window, Intervals, Matrix) categories, tailored for candidates with 1-2 years of experience preparing for roles at EPAM Systems. Each problem includes a problem statement, dry run with test cases, algorithm, and a Python solution, formatted for clarity. Backtracking problems include recursive solutions with pruning where applicable.

## Contents

<b>1</b>	<b>Permutations</b>	<b>4</b>
1.1	Problem Statement . . . . .	5
1.2	Dry Run on Test Cases . . . . .	5
1.3	Algorithm . . . . .	5
1.4	Python Solution . . . . .	5
<b>2</b>	<b>Permutations II</b>	<b>5</b>
2.1	Problem Statement . . . . .	5
2.2	Dry Run on Test Cases . . . . .	5
2.3	Algorithm . . . . .	6
2.4	Python Solution . . . . .	6
<b>3</b>	<b>Subsets</b>	<b>6</b>
3.1	Problem Statement . . . . .	6
3.2	Dry Run on Test Cases . . . . .	6
3.3	Algorithm . . . . .	7
3.4	Python Solution . . . . .	7
<b>4</b>	<b>Subsets II</b>	<b>7</b>
4.1	Problem Statement . . . . .	7
4.2	Dry Run on Test Cases . . . . .	7
4.3	Algorithm . . . . .	7
4.4	Python Solution . . . . .	8
<b>5</b>	<b>Combination Sum</b>	<b>8</b>
5.1	Problem Statement . . . . .	8
5.2	Dry Run on Test Cases . . . . .	8
5.3	Algorithm . . . . .	8
5.4	Python Solution . . . . .	8

<b>6</b>	<b>Combination Sum II</b>	<b>9</b>
6.1	Problem Statement . . . . .	9
6.2	Dry Run on Test Cases . . . . .	9
6.3	Algorithm . . . . .	9
6.4	Python Solution . . . . .	9
<b>7</b>	<b>N-Queens</b>	<b>10</b>
7.1	Problem Statement . . . . .	10
7.2	Dry Run on Test Cases . . . . .	10
7.3	Algorithm . . . . .	10
7.4	Python Solution . . . . .	10
<b>8</b>	<b>N-Queens II</b>	<b>11</b>
8.1	Problem Statement . . . . .	11
8.2	Dry Run on Test Cases . . . . .	11
8.3	Algorithm . . . . .	11
8.4	Python Solution . . . . .	11
<b>9</b>	<b>Generate Parentheses</b>	<b>12</b>
9.1	Problem Statement . . . . .	12
9.2	Dry Run on Test Cases . . . . .	12
9.3	Algorithm . . . . .	12
9.4	Python Solution . . . . .	12
<b>10</b>	<b>Letter Combinations of a Phone Number</b>	<b>13</b>
10.1	Problem Statement . . . . .	13
10.2	Dry Run on Test Cases . . . . .	13
10.3	Algorithm . . . . .	13
10.4	Python Solution . . . . .	13
<b>11</b>	<b>Word Search</b>	<b>14</b>
11.1	Problem Statement . . . . .	14
11.2	Dry Run on Test Cases . . . . .	14
11.3	Algorithm . . . . .	14
11.4	Python Solution . . . . .	14
<b>12</b>	<b>Word Search II</b>	<b>15</b>
12.1	Problem Statement . . . . .	15
12.2	Dry Run on Test Cases . . . . .	15
12.3	Algorithm . . . . .	15
12.4	Python Solution . . . . .	15
<b>13</b>	<b>Design Linked List</b>	<b>16</b>
13.1	Problem Statement . . . . .	16
13.2	Dry Run on Test Cases . . . . .	16
13.3	Algorithm . . . . .	17
13.4	Python Solution . . . . .	17
<b>14</b>	<b>Design HashMap</b>	<b>18</b>
14.1	Problem Statement . . . . .	18

14.2 Dry Run on Test Cases . . . . .	18
14.3 Algorithm . . . . .	18
14.4 Python Solution . . . . .	19
<b>15 Design Stack Using Queues</b>	<b>20</b>
15.1 Problem Statement . . . . .	20
15.2 Dry Run on Test Cases . . . . .	20
15.3 Algorithm . . . . .	20
15.4 Python Solution . . . . .	20
<b>16 Design Queue Using Stacks</b>	<b>21</b>
16.1 Problem Statement . . . . .	21
16.2 Dry Run on Test Cases . . . . .	21
16.3 Algorithm . . . . .	21
16.4 Python Solution . . . . .	21
<b>17 Design Circular Queue</b>	<b>22</b>
17.1 Problem Statement . . . . .	22
17.2 Dry Run on Test Cases . . . . .	22
17.3 Algorithm . . . . .	22
17.4 Python Solution . . . . .	23
<b>18 LRU Cache</b>	<b>23</b>
18.1 Problem Statement . . . . .	23
18.2 Dry Run on Test Cases . . . . .	24
18.3 Algorithm . . . . .	24
18.4 Python Solution . . . . .	24
<b>19 Min Stack</b>	<b>25</b>
19.1 Problem Statement . . . . .	25
19.2 Dry Run on Test Cases . . . . .	25
19.3 Algorithm . . . . .	25
19.4 Python Solution . . . . .	26
<b>20 Longest Substring Without Repeating Characters</b>	<b>26</b>
20.1 Problem Statement . . . . .	26
20.2 Dry Run on Test Cases . . . . .	26
20.3 Algorithm . . . . .	26
20.4 Python Solution . . . . .	27
<b>21 Longest Repeating Character Replacement</b>	<b>27</b>
21.1 Problem Statement . . . . .	27
21.2 Dry Run on Test Cases . . . . .	27
21.3 Algorithm . . . . .	27
21.4 Python Solution . . . . .	27
<b>22 Minimum Window Substring</b>	<b>28</b>
22.1 Problem Statement . . . . .	28
22.2 Dry Run on Test Cases . . . . .	28
22.3 Algorithm . . . . .	28

22.4 Python Solution . . . . .	28
<b>23 Sliding Window Maximum</b>	<b>29</b>
23.1 Problem Statement . . . . .	29
23.2 Dry Run on Test Cases . . . . .	29
23.3 Algorithm . . . . .	29
23.4 Python Solution . . . . .	30
<b>24 Merge Intervals</b>	<b>30</b>
24.1 Problem Statement . . . . .	30
24.2 Dry Run on Test Cases . . . . .	30
24.3 Algorithm . . . . .	30
24.4 Python Solution . . . . .	31
<b>25 Non-overlapping Intervals</b>	<b>31</b>
25.1 Problem Statement . . . . .	31
25.2 Dry Run on Test Cases . . . . .	31
25.3 Algorithm . . . . .	31
25.4 Python Solution . . . . .	31
<b>26 Insert Interval</b>	<b>32</b>
26.1 Problem Statement . . . . .	32
26.2 Dry Run on Test Cases . . . . .	32
26.3 Algorithm . . . . .	32
26.4 Python Solution . . . . .	32
<b>27 Meeting Rooms</b>	<b>33</b>
27.1 Problem Statement . . . . .	33
27.2 Dry Run on Test Cases . . . . .	33
27.3 Algorithm . . . . .	33
27.4 Python Solution . . . . .	33
<b>28 Meeting Rooms II</b>	<b>34</b>
28.1 Problem Statement . . . . .	34
28.2 Dry Run on Test Cases . . . . .	34
28.3 Algorithm . . . . .	34
28.4 Python Solution . . . . .	34
<b>29 Spiral Matrix</b>	<b>34</b>
29.1 Problem Statement . . . . .	35
29.2 Dry Run on Test Cases . . . . .	35
29.3 Algorithm . . . . .	35
29.4 Python Solution . . . . .	35

## 1 Permutations

## 1.1 Problem Statement

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

## 1.2 Dry Run on Test Cases

- **Test Case 1:**  $\text{nums} = [1,2,3] \rightarrow \text{Output: } [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]$
- **Test Case 2:**  $\text{nums} = [0,1] \rightarrow \text{Output: } [[0,1],[1,0]]$
- **Test Case 3:**  $\text{nums} = [1] \rightarrow \text{Output: } [[1]]$
- **Test Case 4:**  $\text{nums} = [] \rightarrow \text{Output: } [[]]$

## 1.3 Algorithm

1. Use backtracking to generate permutations.
2. For each number, include it if not used, recurse, and backtrack.
3. Add permutation when length equals input size.

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

## 1.4 Python Solution

```
1 def permute(nums):
2     result = []
3     def backtrack(curr, used):
4         if len(curr) == len(nums):
5             result.append(curr[:])
6             return
7         for i in range(len(nums)):
8             if not used[i]:
9                 used[i] = True
10                curr.append(nums[i])
11                backtrack(curr, used)
12                curr.pop()
13                used[i] = False
14        backtrack([], [False] * len(nums))
15    return result
```

# 2 Permutations II

## 2.1 Problem Statement

Given an array with possible duplicates, return all unique permutations.

## 2.2 Dry Run on Test Cases

- **Test Case 1:**  $\text{nums} = [1,1,2] \rightarrow \text{Output: } [[1,1,2],[1,2,1],[2,1,1]]$

- **Test Case 2:**  $\text{nums} = [1,2,3] \rightarrow \text{Output: } [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]$
- **Test Case 3:**  $\text{nums} = [1] \rightarrow \text{Output: } [[1]]$
- **Test Case 4:**  $\text{nums} = [] \rightarrow \text{Output: } [[]]$

## 2.3 Algorithm

1. Sort array to handle duplicates.
2. Use backtracking, skip duplicates at same level.
3. Add permutation when length equals input size.

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

## 2.4 Python Solution

```

1 def permute_unique(nums):
2     nums.sort()
3     result = []
4     def backtrack(curr, used):
5         if len(curr) == len(nums):
6             result.append(curr[:])
7             return
8         for i in range(len(nums)):
9             if used[i] or (i > 0 and nums[i] == nums[i-1] and not
10                used[i-1]):
11                 continue
12             used[i] = True
13             curr.append(nums[i])
14             backtrack(curr, used)
15             curr.pop()
16             used[i] = False
17     backtrack([], [False] * len(nums))
18     return result

```

# 3 Subsets

## 3.1 Problem Statement

Given an array of distinct integers, return all possible subsets.

## 3.2 Dry Run on Test Cases

- **Test Case 1:**  $\text{nums} = [1,2,3] \rightarrow \text{Output: } [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]$
- **Test Case 2:**  $\text{nums} = [0] \rightarrow \text{Output: } [[],[0]]$
- **Test Case 3:**  $\text{nums} = [] \rightarrow \text{Output: } [[]]$

- **Test Case 4:**  $\text{nums} = [1,2] \rightarrow \text{Output: } [[],[1],[2],[1,2]]$

### 3.3 Algorithm

1. Use backtracking to include/exclude each number.
2. Add current subset at each step.
3. Recurse for next index.

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

### 3.4 Python Solution

```

1 def subsets(nums):
2     result = []
3     def backtrack(curr, i):
4         result.append(curr[:])
5         for j in range(i, len(nums)):
6             curr.append(nums[j])
7             backtrack(curr, j + 1)
8             curr.pop()
9     backtrack([], 0)
10    return result

```

## 4 Subsets II

### 4.1 Problem Statement

Given an array with possible duplicates, return all unique subsets.

### 4.2 Dry Run on Test Cases

- **Test Case 1:**  $\text{nums} = [1,2,2] \rightarrow \text{Output: } [[],[1],[1,2],[1,2,2],[2],[2,2]]$
- **Test Case 2:**  $\text{nums} = [0] \rightarrow \text{Output: } [[],[0]]$
- **Test Case 3:**  $\text{nums} = [] \rightarrow \text{Output: } [[]]$
- **Test Case 4:**  $\text{nums} = [1,1] \rightarrow \text{Output: } [[],[1],[1,1]]$

### 4.3 Algorithm

1. Sort array to handle duplicates.
2. Use backtracking, skip duplicates at same level.
3. Add subset at each step.

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

## 4.4 Python Solution

```
1 def subsets_with_dup(nums):
2     nums.sort()
3     result = []
4     def backtrack(curr, i):
5         result.append(curr[:])
6         for j in range(i, len(nums)):
7             if j > i and nums[j] == nums[j-1]:
8                 continue
9             curr.append(nums[j])
10            backtrack(curr, j + 1)
11            curr.pop()
12    backtrack([], 0)
13    return result
```

## 5 Combination Sum

### 5.1 Problem Statement

Given an array of distinct integers and target, return all combinations summing to target (unlimited use).

### 5.2 Dry Run on Test Cases

- \* **Test Case 1:** candidates = [2,3,6,7], target = 7 → Output: [[2,2,3],[7]]
- \* **Test Case 2:** candidates = [2,3,5], target = 8 → Output: [[2,2,2,2],[2,3,3],[3,5]]
- \* **Test Case 3:** candidates = [2], target = 1 → Output: []
- \* **Test Case 4:** candidates = [], target = 1 → Output: []

### 5.3 Algorithm

1. Use backtracking to try each candidate.
2. If sum equals target, add to result.
3. If sum exceeds target, backtrack.

**Time Complexity:**  $O(2^{target/min(candidates)})$     **Space Complexity:**  $O(target)$

### 5.4 Python Solution

```
1 def combination_sum(candidates, target):
2     result = []
3     def backtrack(curr, i, total):
4         if total == target:
5             result.append(curr[:])
6         return
```

```

7         if total > target:
8             return
9         for j in range(i, len(candidates)):
10             curr.append(candidates[j])
11             backtrack(curr, j, total + candidates[j])
12             curr.pop()
13     backtrack([], 0, 0)
14     return result

```

## 6 Combination Sum II

### 6.1 Problem Statement

Given an array with possible duplicates and target, return all unique combinations summing to target (each number used once).

### 6.2 Dry Run on Test Cases

- **Test Case 1:** candidates = [10,1,2,7,6,1,5], target = 8 → Output: [[1,1,6],[1,2,5],[1,7],[2,6]]
- **Test Case 2:** candidates = [2,5,2,1,2], target = 5 → Output: [[1,2,2],[5]]
- **Test Case 3:** candidates = [2], target = 1 → Output: []
- **Test Case 4:** candidates = [], target = 1 → Output: []

### 6.3 Algorithm

1. Sort candidates to handle duplicates.
2. Use backtracking, skip duplicates at same level.
3. If sum equals target, add to result.

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

### 6.4 Python Solution

```

1 def combination_sum2(candidates, target):
2     candidates.sort()
3     result = []
4     def backtrack(curr, i, total):
5         if total == target:
6             result.append(curr[:])
7             return
8         if total > target:
9             return
10        for j in range(i, len(candidates)):

```

```

11         if j > i and candidates[j] ==
           candidates[j-1]:
12             continue
13         curr.append(candidates[j])
14         backtrack(curr, j + 1, total +
           candidates[j])
15         curr.pop()
16     backtrack([], 0, 0)
17     return result

```

## 7 N-Queens

### 7.1 Problem Statement

Given an  $n \times n$  chessboard, return all distinct solutions to the N-Queens problem.

### 7.2 Dry Run on Test Cases

- **Test Case 1:**  $n = 4 \rightarrow$  Output: `[[".Q..","...Q","Q...", "..Q."],["..Q.", "Q...", "...Q", ".Q.."]]`
- **Test Case 2:**  $n = 1 \rightarrow$  Output: `[["Q"]]`
- **Test Case 3:**  $n = 2 \rightarrow$  Output: `[]`
- **Test Case 4:**  $n = 3 \rightarrow$  Output: `[]`

### 7.3 Algorithm

1. Use backtracking to place queens row by row.
2. Check if position is safe (no conflicts in column, diagonals).
3. Add valid board configuration to result.

**Time Complexity:**  $O(n!)$     **Space Complexity:**  $O(n^2)$

### 7.4 Python Solution

```

1 def solve_n_queens(n):
2     def create_board(positions):
3         board = [['.' * n for _ in range(n)]]
4         for r, c in enumerate(positions):
5             board[r][c] = 'Q'
6         return [''.join(row) for row in board]
7
8     def is_safe(row, col, queens):
9         for r, c in enumerate(queens[:row]):
10            if c == col or r - c == row - col or r
              + c == row + col:

```

```

11         return False
12     return True
13
14     def backtrack(row, queens):
15         if row == n:
16             result.append(create_board(queens))
17             return
18         for col in range(n):
19             if is_safe(row, col, queens):
20                 queens[row] = col
21                 backtrack(row + 1, queens)
22
23     result = []
24     backtrack(0, [0] * n)
25     return result

```

## 8 N-Queens II

### 8.1 Problem Statement

Given an  $n \times n$  chessboard, return the number of distinct N-Queens solutions.

### 8.2 Dry Run on Test Cases

- **Test Case 1:**  $n = 4 \rightarrow$  Output: 2
- **Test Case 2:**  $n = 1 \rightarrow$  Output: 1
- **Test Case 3:**  $n = 2 \rightarrow$  Output: 0
- **Test Case 4:**  $n = 3 \rightarrow$  Output: 0

### 8.3 Algorithm

1. Use backtracking to place queens row by row.
2. Check if position is safe (no conflicts).
3. Increment count when a valid configuration is found.

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

### 8.4 Python Solution

```

1 def total_n_queens(n):
2     def is_safe(row, col, queens):
3         for r, c in enumerate(queens[:row]):
4             if c == col or r - c == row - col or r
              + c == row + col:

```

```

5         return False
6     return True

7
8     def backtrack(row, queens):
9         if row == n:
10             return 1
11         count = 0
12         for col in range(n):
13             if is_safe(row, col, queens):
14                 queens[row] = col
15                 count += backtrack(row + 1, queens)
16         return count
17
18     return backtrack(0, [0] * n)

```

## 9 Generate Parentheses

### 9.1 Problem Statement

Given  $n$  pairs of parentheses, generate all valid combinations.

### 9.2 Dry Run on Test Cases

- **Test Case 1:**  $n = 3 \rightarrow$  Output: ["((()))", "(()())", "(())()", "()(())", "()()()"]
- **Test Case 2:**  $n = 1 \rightarrow$  Output: ["()"]
- **Test Case 3:**  $n = 0 \rightarrow$  Output: [""]
- **Test Case 4:**  $n = 2 \rightarrow$  Output: ["(())", "()()"]

### 9.3 Algorithm

1. Use backtracking with open/close counts.
2. Add '(' if open <  $n$ , add ')' if close < open.
3. Add to result when length =  $2n$ .

**Time Complexity:**  $O(4^n/\sqrt{n})$  (Catalan number)    **Space Complexity:**  $O(n)$

### 9.4 Python Solution

```

1 def generate_parenthesis(n):
2     result = []
3     def backtrack(curr, open_count, close_count):
4         if len(curr) == 2 * n:
5             result.append(curr)
6         return

```

```

7         if open_count < n:
8             backtrack(curr + '(', open_count + 1,
9                         close_count)
10        if close_count < open_count:
11            backtrack(curr + ')', open_count,
12                      close_count + 1)
13    backtrack('', 0, 0)
14    return result

```

## 10 Letter Combinations of a Phone Number

### 10.1 Problem Statement

Given a string of digits (2-9), return all possible letter combinations.

### 10.2 Dry Run on Test Cases

- **Test Case 1:** digits = "23" → Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
- **Test Case 2:** digits = "" → Output: []
- **Test Case 3:** digits = "2" → Output: ["a","b","c"]
- **Test Case 4:** digits = "7" → Output: ["p","q","r","s"]

### 10.3 Algorithm

1. Use backtracking to try each letter for each digit.
2. Map digits to letters (e.g., 2 -> "abc").
3. Add combination when length equals digits length.

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

### 10.4 Python Solution

```

1 def letter_combinations(digits):
2     if not digits:
3         return []
4     mapping = {'2': 'abc', '3': 'def', '4': 'ghi',
5               '5': 'jkl',
6               '6': 'mno', '7': 'pqrs', '8': 'tuv',
7               '9': 'wxyz'}
8     result = []
9     def backtrack(curr, i):
10         if i == len(digits):
11             result.append(curr)

```

```

10         return
11         for c in mapping[digits[i]]:
12             backtrack(curr + c, i + 1)
13     backtrack('', 0)
14     return result

```

## 11 Word Search

### 11.1 Problem Statement

Given a 2D board and a word, find if the word exists in the grid (adjacent cells).

### 11.2 Dry Run on Test Cases

- **Test Case 1:** board = `[["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, word = "ABCCED" → Output: True
- **Test Case 2:** board = `[["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, word = "SEE" → Output: True
- **Test Case 3:** board = `[["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, word = "ABCB" → Output: False
- **Test Case 4:** board = `[]`, word = "A" → Output: False

### 11.3 Algorithm

1. Use backtracking from each cell.
2. Check if current path matches word prefix.
3. Mark visited cells, explore 4 directions, backtrack.

**Time Complexity:**  $O(m \cdot n \cdot 4^{|word|})$     **Space Complexity:**  $O(m \cdot n)$

### 11.4 Python Solution

```

1 def exist(board, word):
2     if not board or not board[0]:
3         return False
4     rows, cols = len(board), len(board[0])
5
6     def backtrack(i, j, k):
7         if k == len(word):
8             return True
9         if i < 0 or i >= rows or j < 0 or j >= cols
10            or board[i][j] != word[k]:
11             return False
12         temp = board[i][j]

```

```

12         board[i][j] = '#'
13         result = (backtrack(i+1, j, k+1) or
14                   backtrack(i-1, j, k+1) or
15                   backtrack(i, j+1, k+1) or
16                   backtrack(i, j-1, k+1))
17         board[i][j] = temp
18         return result
19
20     for i in range(rows):
21         for j in range(cols):
22             if backtrack(i, j, 0):
23                 return True
24     return False

```

## 12 Word Search II

### 12.1 Problem Statement

Given a 2D board and a list of words, find all words in the board.

### 12.2 Dry Run on Test Cases

- **Test Case 1:** board = `[["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]]`, words = `["oath","pea","eat","rain"]` → Output: `["eat","oath"]`
- **Test Case 2:** board = `[["a","b"],["c","d"]]`, words = `["abcb"]` → Output: `[]`
- **Test Case 3:** board = `[["a"]]`, words = `["a"]` → Output: `["a"]`
- **Test Case 4:** board = `[]`, words = `["a"]` → Output: `[]`

### 12.3 Algorithm

1. Build a trie for the word list.
2. Use backtracking from each cell, follow trie nodes.
3. Add found words to result, remove from trie to avoid duplicates.

**Time Complexity:**  $O(m \cdot n \cdot 4^{|maxword|})$     **Space Complexity:**  $O(|words|)$

### 12.4 Python Solution

```

1 class TrieNode:
2     def __init__(self):
3         self.children = {}
4         self.word = None
5

```

```

6 def find_words(board, words):
7     if not board or not board[0]:
8         return []
9
10    # Build trie
11    root = TrieNode()
12    for word in words:
13        node = root
14        for c in word:
15            if c not in node.children:
16                node.children[c] = TrieNode()
17            node = node.children[c]
18        node.word = word
19
20    rows, cols = len(board), len(board[0])
21    result = []
22
23    def backtrack(i, j, node):
24        if i < 0 or i >= rows or j < 0 or j >= cols
25            or board[i][j] not in node.children:
26            return
27        c = board[i][j]
28        node = node.children[c]
29        if node.word:
30            result.append(node.word)
31            node.word = None
32        temp = board[i][j]
33        board[i][j] = '#'
34        for di, dj in [(1,0), (-1,0), (0,1), (0,-1)]:
35            backtrack(i + di, j + dj, node)
36        board[i][j] = temp
37
38    for i in range(rows):
39        for j in range(cols):
40            backtrack(i, j, root)
41    return result

```

## 13 Design Linked List

### 13.1 Problem Statement

Design a singly linked list with methods: get, addAtHead, addAtTail, addAtIndex, deleteAtIndex.

### 13.2 Dry Run on Test Cases

- **Test Case 1:** ["MyLinkedList", "addAtHead", "addAtTail", "addAtIndex", "get", "deleteAtIndex"] → Output: [null, null, null, null, 2, null, 3]

- **Test Case 2:** ["MyLinkedList","addAtHead","get"] [[],[1],[0]] → Output: [null,null,1]
- **Test Case 3:** ["MyLinkedList","get"] [[],[0]] → Output: [null,-1]
- **Test Case 4:** ["MyLinkedList","addAtTail","get"] [[],[1],[0]] → Output: [null,null,1]

### 13.3 Algorithm

1. Use a node class with value and next pointer.
2. Implement methods to manipulate linked list.
3. Track size for index validation.

**Time Complexity:**  $O(1)$  for addAtHead, get(0);  $O(n)$  for others  
**Space Complexity:**  $O(n)$

### 13.4 Python Solution

```

1 class ListNode:
2     def __init__(self, val=0, next=None):
3         self.val = val
4         self.next = next
5
6 class MyLinkedList:
7     def __init__(self):
8         self.head = None
9         self.size = 0
10
11     def get(self, index):
12         if index < 0 or index >= self.size:
13             return -1
14         curr = self.head
15         for _ in range(index):
16             curr = curr.next
17         return curr.val
18
19     def addAtHead(self, val):
20         self.head = ListNode(val, self.head)
21         self.size += 1
22
23     def addAtTail(self, val):
24         if not self.head:
25             self.head = ListNode(val)
26         else:
27             curr = self.head
28             while curr.next:
29                 curr = curr.next
30             curr.next = ListNode(val)
31         self.size += 1

```

```

32
33     def addAtIndex(self, index, val):
34         if index < 0 or index > self.size:
35             return
36         if index == 0:
37             self.addAtHead(val)
38             return
39         curr = self.head
40         for _ in range(index - 1):
41             curr = curr.next
42         curr.next = ListNode(val, curr.next)
43         self.size += 1
44
45     def deleteAtIndex(self, index):
46         if index < 0 or index >= self.size:
47             return
48         if index == 0:
49             self.head = self.head.next
50             self.size -= 1
51             return
52         curr = self.head
53         for _ in range(index - 1):
54             curr = curr.next
55         curr.next = curr.next.next
56         self.size -= 1

```

## 14 Design HashMap

### 14.1 Problem Statement

Design a HashMap with put, get, and remove operations.

### 14.2 Dry Run on Test Cases

- **Test Case 1:** ["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]  
[[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]] → Output: [null, null, null, 1, -1, null, 1, null, -1]
- **Test Case 2:** ["MyHashMap", "put", "get"] [[], [1, 1], [1]] → Output: [null, null, 1]
- **Test Case 3:** ["MyHashMap", "get"] [[], [0]] → Output: [null, -1]
- **Test Case 4:** ["MyHashMap", "put", "remove", "get"] [[], [1, 1], [1], [1]] → Output: [null, null, null, -1]

### 14.3 Algorithm

1. Use array of linked lists (chaining) for collision handling.
2. Hash function: key

3. Implement put, get, remove with linked list traversal.

**Time Complexity:**  $O(1)$  average,  $O(n)$  worst    **Space Complexity:**  $O(n)$

## 14.4 Python Solution

```
1 class ListNode:
2     def __init__(self, key=-1, val=-1, next=None):
3         self.key = key
4         self.val = val
5         self.next = next
6
7 class MyHashMap:
8     def __init__(self):
9         self.size = 1000
10        self.buckets = [None] * self.size
11
12    def _hash(self, key):
13        return key % self.size
14
15    def put(self, key, value):
16        index = self._hash(key)
17        if not self.buckets[index]:
18            self.buckets[index] = ListNode(key,
19                                           value)
20        else:
21            curr = self.buckets[index]
22            while True:
23                if curr.key == key:
24                    curr.val = value
25                    return
26                if not curr.next:
27                    break
28                curr = curr.next
29            curr.next = ListNode(key, value)
30
31    def get(self, key):
32        index = self._hash(key)
33        curr = self.buckets[index]
34        while curr:
35            if curr.key == key:
36                return curr.val
37            curr = curr.next
38        return -1
39
40    def remove(self, key):
41        index = self._hash(key)
42        curr = self.buckets[index]
43        if not curr:
44            return
45        if curr.key == key:
```

```

45         self.buckets[index] = curr.next
46         return
47     while curr.next:
48         if curr.next.key == key:
49             curr.next = curr.next.next
50             return
51         curr = curr.next

```

## 15 Design Stack Using Queues

### 15.1 Problem Statement

Implement a stack using queues with push, pop, top, and empty operations.

### 15.2 Dry Run on Test Cases

- **Test Case 1:** ["MyStack", "push", "push", "top", "pop", "empty"] [[], [1], [2], [], [], []] → Output: [null, null, null, 2, 2, false]
- **Test Case 2:** ["MyStack", "push", "top", "empty"] [[], [1], [], []] → Output: [null, null, 1, false]
- **Test Case 3:** ["MyStack", "empty"] [[], []] → Output: [null, true]
- **Test Case 4:** ["MyStack", "push", "pop"] [[], [1], []] → Output: [null, null, 1]

### 15.3 Algorithm

1. Use one queue.
2. Push: add element, rotate queue to maintain stack order.
3. Pop/top: return/remove front element.
4. Empty: check if queue is empty.

**Time Complexity:**  $O(n)$  for push,  $O(1)$  for others    **Space Complexity:**  $O(n)$

### 15.4 Python Solution

```

1 from collections import deque
2
3 class MyStack:
4     def __init__(self):
5         self.q = deque()
6
7     def push(self, x):
8         self.q.append(x)

```

```

9         for _ in range(len(self.q) - 1):
10             self.q.append(self.q.popleft())
11
12     def pop(self):
13         return self.q.popleft()
14
15     def top(self):
16         return self.q[0]
17
18     def empty(self):
19         return len(self.q) == 0

```

## 16 Design Queue Using Stacks

### 16.1 Problem Statement

Implement a queue using stacks with enqueue, dequeue, peek, and empty operations.

### 16.2 Dry Run on Test Cases

- **Test Case 1:** ["MyQueue", "push", "push", "peek", "pop", "empty"] [[], [1], [2], [], [], []] → Output: [null, null, null, 1, 1, false]
- **Test Case 2:** ["MyQueue", "push", "peek", "empty"] [[], [1], [], []] → Output: [null, null, 1, false]
- **Test Case 3:** ["MyQueue", "empty"] [[], []] → Output: [null, true]
- **Test Case 4:** ["MyQueue", "push", "pop"] [[], [1], []] → Output: [null, null, 1]

### 16.3 Algorithm

1. Use two stacks: input for push, output for pop/peek.
2. Push: add to input stack.
3. Pop/peek: move input to output if output empty, then pop/peek.

**Time Complexity:**  $O(1)$  amortized for push/pop,  $O(n)$  worst for pop/peek    **Space Complexity:**  $O(n)$

### 16.4 Python Solution

```

1 class MyQueue:
2     def __init__(self):
3         self.input = []
4         self.output = []
5
6     def push(self, x):

```

```

7         self.input.append(x)
8
9     def pop(self):
10         self.peak()
11         return self.output.pop()
12
13     def peek(self):
14         if not self.output:
15             while self.input:
16                 self.output.append(self.input.pop())
17             return self.output[-1]
18
19     def empty(self):
20         return not self.input and not self.output

```

## 17 Design Circular Queue

### 17.1 Problem Statement

Design a circular queue with enqueue, dequeue, front, rear, isEmpty, isFull operations.

### 17.2 Dry Run on Test Cases

- **Test Case 1:** ["MyCircularQueue", "enqueue", "enqueue", "enqueue", "enqueue", "Rear", "Front", "isEmpty", "isFull"] → Output: [null, true, true, true, false, 3, true, true, true, 4]
- **Test Case 2:** ["MyCircularQueue", "enqueue", "Rear", "Front"] → Output: [null, true, 1, 1]
- **Test Case 3:** ["MyCircularQueue", "isEmpty"] → Output: [null, true]
- **Test Case 4:** ["MyCircularQueue", "enqueue", "dequeue"] → Output: [null, true, true]

### 17.3 Algorithm

1. Use array with front and rear pointers.
2. Enqueue: add at rear, increment rear modulo size.
3. Dequeue: increment front modulo size.
4. Track size for empty/full checks.

**Time Complexity:**  $O(1)$  for all operations    **Space Complexity:**  $O(k)$

## 17.4 Python Solution

```
1 class MyCircularQueue:
2     def __init__(self, k):
3         self.size = k
4         self.queue = [None] * k
5         self.front = -1 # Index of front element
6         self.rear = -1  # Index of last element
7         self.count = 0  # Number of elements
8
9     def enqueue(self, value):
10        if self.isFull():
11            return False
12        if self.isEmpty():
13            self.front = 0
14        self.rear = (self.rear + 1) % self.size
15        self.queue[self.rear] = value
16        self.count += 1
17        return True
18
19    def dequeue(self):
20        if self.isEmpty():
21            return False
22        self.front = (self.front + 1) % self.size
23        self.count -= 1
24        if self.isEmpty():
25            self.front = -1
26            self.rear = -1
27        return True
28
29    def Front(self):
30        return self.queue[self.front] if not self.
31            isEmpty() else -1
32
33    def Rear(self):
34        return self.queue[self.rear] if not self.
35            isEmpty() else -1
36
37    def isEmpty(self):
38        return self.count == 0
39
40    def isFull(self):
41        return self.count == self.size
```

## 18 LRU Cache

### 18.1 Problem Statement

Design an LRU (Least Recently Used) cache with get and put operations.

## 18.2 Dry Run on Test Cases

- **Test Case 1:** ["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]  
[[2],[1,1],[2,2],[1],[3,3],[2],[4,4],[1],[3],[4]] → Output: [null,null,null,1,null,-1,null,-1,3,4]
- **Test Case 2:** ["LRUCache", "put", "get"] [[1],[1,1],[1]] → Output: [null,null,1]
- **Test Case 3:** ["LRUCache", "get"] [[1],[1]] → Output: [null,-1]
- **Test Case 4:** ["LRUCache", "put", "put", "get"] [[1],[1,1],[2,2],[1]] → Output: [null,null,null,-1]

## 18.3 Algorithm

1. Use doubly linked list and hashmap.
2. Get: return value from hashmap, move node to front.
3. Put: update or add node, move to front, remove tail if full.

**Time Complexity:**  $O(1)$  for get/put    **Space Complexity:**  $O(capacity)$

## 18.4 Python Solution

```
1 class ListNode:
2     def __init__(self, key=0, value=0):
3         self.key = key
4         self.value = value
5         self.prev = None
6         self.next = None
7
8 class LRUCache:
9     def __init__(self, capacity):
10        self.capacity = capacity
11        self.cache = {}
12        self.head = ListNode()
13        self.tail = ListNode()
14        self.head.next = self.tail
15        self.tail.prev = self.head
16
17    def _add_node(self, node):
18        node.prev = self.head
19        node.next = self.head.next
20        self.head.next.prev = node
21        self.head.next = node
22
23    def _remove_node(self, node):
24        node.prev.next = node.next
25        node.next.prev = node.prev
26
27    def get(self, key):
```

```

28         if key in self.cache:
29             node = self.cache[key]
30             self._remove_node(node)
31             self._add_node(node)
32             return node.value
33         return -1
34
35     def put(self, key, value):
36         if key in self.cache:
37             self._remove_node(self.cache[key])
38         node = ListNode(key, value)
39         self._add_node(node)
40         self.cache[key] = node
41         if len(self.cache) > self.capacity:
42             lru = self.tail.prev
43             self._remove_node(lru)
44             del self.cache[lru.key]

```

## 19 Min Stack

### 19.1 Problem Statement

Design a stack that supports push, pop, top, and getMin in constant time.

### 19.2 Dry Run on Test Cases

- **Test Case 1:** ["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]  
[[], [-2], [0], [-3], [], [], [], []] → Output: [null, null, null, null, -3, null, 0, -2]
- **Test Case 2:** ["MinStack", "push", "getMin"] [[], [1], []] → Output: [null, null, 1]
- **Test Case 3:** ["MinStack", "getMin"] [[], []] → Output: [null, None]
- **Test Case 4:** ["MinStack", "push", "pop", "getMin"] [[], [1], [], []] → Output: [null, null, null, None]

### 19.3 Algorithm

1. Use two stacks: one for values, one for minimums.
2. Push: add value, update min stack with current min.
3. Pop: remove from both stacks.
4. Top/getMin: access top of respective stacks.

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

## 19.4 Python Solution

```
1 class MinStack:
2     def __init__(self):
3         self.stack = []
4         self.min_stack = []
5
6     def push(self, val):
7         self.stack.append(val)
8         if not self.min_stack or val <= self.
           min_stack[-1]:
9             self.min_stack.append(val)
10
11    def pop(self):
12        if self.stack:
13            val = self.stack.pop()
14            if val == self.min_stack[-1]:
15                self.min_stack.pop()
16
17    def top(self):
18        return self.stack[-1] if self.stack else
           None
19
20    def getMin(self):
21        return self.min_stack[-1] if self.min_stack
           else None
```

## 20 Longest Substring Without Repeating Characters

### 20.1 Problem Statement

Given a string, find the length of the longest substring without repeating characters.

### 20.2 Dry Run on Test Cases

- **Test Case 1:**  $s = \text{"abcabcbb"} \rightarrow \text{Output: } 3 \text{ ("abc")}$
- **Test Case 2:**  $s = \text{"bbbbbb"} \rightarrow \text{Output: } 1 \text{ ("b")}$
- **Test Case 3:**  $s = \text{"pwwkew"} \rightarrow \text{Output: } 3 \text{ ("wke")}$
- **Test Case 4:**  $s = \text{""} \rightarrow \text{Output: } 0$

### 20.3 Algorithm

1. Use sliding window with hashmap to track character indices.
2. Move right pointer, update start if repeated character found.

3. Track max length of window.

**Time Complexity:**  $O(n)$     **Space Complexity:**  $O(\min(m, n))$  ( $m$  = charset size)

## 20.4 Python Solution

```
1 def length_of_longest_substring(s):
2     char_index = {}
3     max_len = 0
4     start = 0
5     for end, char in enumerate(s):
6         if char in char_index and char_index[char]
7             >= start:
8             start = char_index[char] + 1
9         char_index[char] = end
10        max_len = max(max_len, end - start + 1)
11    return max_len
```

# 21 Longest Repeating Character Replacement

## 21.1 Problem Statement

Given a string and  $k$ , find the length of the longest substring with at most  $k$  replacements.

## 21.2 Dry Run on Test Cases

- **Test Case 1:**  $s = \text{"ABAB"}, k = 2 \rightarrow \text{Output: } 4$
- **Test Case 2:**  $s = \text{"AABABBA"}, k = 1 \rightarrow \text{Output: } 4$
- **Test Case 3:**  $s = \text{""}, k = 1 \rightarrow \text{Output: } 0$
- **Test Case 4:**  $s = \text{"AAAA"}, k = 0 \rightarrow \text{Output: } 4$

## 21.3 Algorithm

1. Use sliding window with frequency count.
2. Track max frequency in window.
3. Shrink window if replacements needed  $> k$ .

**Time Complexity:**  $O(n)$     **Space Complexity:**  $O(26)$

## 21.4 Python Solution

```

1 def character_replacement(s, k):
2     freq = {}
3     max_len = 0
4     start = 0
5     max_freq = 0
6     for end, char in enumerate(s):
7         freq[char] = freq.get(char, 0) + 1
8         max_freq = max(max_freq, freq[char])
9         if end - start + 1 - max_freq > k:
10             freq[s[start]] -= 1
11             start += 1
12         max_len = max(max_len, end - start + 1)
13     return max_len

```

## 22 Minimum Window Substring

### 22.1 Problem Statement

Given strings  $s$  and  $t$ , find minimum window in  $s$  containing all characters of  $t$ .

### 22.2 Dry Run on Test Cases

- **Test Case 1:**  $s = \text{"ADOBECODEBANC"}, t = \text{"ABC"} \rightarrow \text{Output: "BANC"}$
- **Test Case 2:**  $s = \text{"a"}, t = \text{"a"} \rightarrow \text{Output: "a"}$
- **Test Case 3:**  $s = \text{"a"}, t = \text{"aa"} \rightarrow \text{Output: ""}$
- **Test Case 4:**  $s = \text{""}, t = \text{"a"} \rightarrow \text{Output: ""}$

### 22.3 Algorithm

1. Use sliding window with two hashmaps.
2. Expand window until all characters in  $t$  are found.
3. Shrink window to minimize while maintaining validity.

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

### 22.4 Python Solution

```

1 def min_window(s, t):
2     if not s or not t:
3         return ""
4     t_count = {}
5     for c in t:

```

```

6         t_count[c] = t_count.get(c, 0) + 1
7     required = len(t_count)
8     formed = 0
9     window_counts = {}
10
11     start, min_len = 0, float('inf')
12     min_window_substr = ""
13     left = right = 0
14
15     while right < len(s):
16         c = s[right]
17         window_counts[c] = window_counts.get(c, 0)
18             + 1
19         if c in t_count and window_counts[c] ==
20             t_count[c]:
21             formed += 1
22         while left <= right and formed == required:
23             c = s[left]
24             if right - left + 1 < min_len:
25                 min_len = right - left + 1
26                 min_window_substr = s[left:right +
27                     1]
28             window_counts[c] -= 1
29             if c in t_count and window_counts[c] <
30                 t_count[c]:
31                 formed -= 1
32             left += 1
33         right += 1
34     return min_window_substr

```

## 23 Sliding Window Maximum

### 23.1 Problem Statement

Given an array and window size  $k$ , find max element in each sliding window.

### 23.2 Dry Run on Test Cases

- **Test Case 1:**  $\text{nums} = [1, 3, -1, -3, 5, 3, 6, 7]$ ,  $k = 3 \rightarrow \text{Output: } [3, 3, 5, 5, 6, 7]$
- **Test Case 2:**  $\text{nums} = [1]$ ,  $k = 1 \rightarrow \text{Output: } [1]$
- **Test Case 3:**  $\text{nums} = []$ ,  $k = 1 \rightarrow \text{Output: } []$
- **Test Case 4:**  $\text{nums} = [1, -1]$ ,  $k = 1 \rightarrow \text{Output: } [1, -1]$

### 23.3 Algorithm

1. Use deque to store indices of potential max elements.

2. Remove indices outside window and smaller elements.
3. Add max of each window to result.

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

## 23.4 Python Solution

```

1 from collections import deque
2
3 def max_sliding_window(nums, k):
4     if not nums:
5         return []
6     result = []
7     dq = deque()
8     for i in range(len(nums)):
9         while dq and dq[0] <= i - k:
10             dq.popleft()
11         while dq and nums[dq[-1]] < nums[i]:
12             dq.pop()
13         dq.append(i)
14         if i >= k - 1:
15             result.append(nums[dq[0]])
16     return result

```

## 24 Merge Intervals

### 24.1 Problem Statement

Given a list of intervals, merge overlapping intervals.

### 24.2 Dry Run on Test Cases

- **Test Case 1:** intervals = [[1,3],[2,6],[8,10],[15,18]] → Output: [[1,6],[8,10],[15,18]]
- **Test Case 2:** intervals = [[1,4],[4,5]] → Output: [[1,5]]
- **Test Case 3:** intervals = [] → Output: []
- **Test Case 4:** intervals = [[1,4]] → Output: [[1,4]]

### 24.3 Algorithm

1. Sort intervals by start time.
2. Merge overlapping intervals by updating end time.
3. Add non-overlapping intervals to result.

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

## 24.4 Python Solution

```
1 def merge(intervals):
2     if not intervals:
3         return []
4     intervals.sort(key=lambda x: x[0])
5     result = [intervals[0]]
6     for start, end in intervals[1:]:
7         if start <= result[-1][1]:
8             result[-1][1] = max(result[-1][1], end)
9         else:
10            result.append([start, end])
11    return result
```

## 25 Non-overlapping Intervals

### 25.1 Problem Statement

Given a list of intervals, find minimum number of intervals to remove to make rest non-overlapping.

### 25.2 Dry Run on Test Cases

- **Test Case 1:** intervals = [[1,2],[2,3],[3,4],[1,3]] → Output: 1
- **Test Case 2:** intervals = [[1,2],[1,2],[1,2]] → Output: 2
- **Test Case 3:** intervals = [[1,2],[2,3]] → Output: 0
- **Test Case 4:** intervals = [] → Output: 0

### 25.3 Algorithm

1. Sort intervals by end time.
2. Count overlapping intervals by comparing start with previous end.
3. Return count of intervals to remove.

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

## 25.4 Python Solution

```
1 def erase_overlap_intervals(intervals):
2     if not intervals:
3         return 0
4     intervals.sort(key=lambda x: x[1])
5     count = 0
6     prev_end = intervals[0][1]
7     for start, end in intervals[1:]:
8         if start < prev_end:
```

```

9         count += 1
10    else:
11        prev_end = end
12    return count

```

## 26 Insert Interval

### 26.1 Problem Statement

Given sorted non-overlapping intervals and a new interval, insert and merge if necessary.

### 26.2 Dry Run on Test Cases

- **Test Case 1:** intervals = [[1,3],[6,9]], newInterval = [2,5] → Output: [[1,5],[6,9]]
- **Test Case 2:** intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8] → Output: [[1,2],[3,10],[12,16]]
- **Test Case 3:** intervals = [], newInterval = [5,7] → Output: [[5,7]]
- **Test Case 4:** intervals = [[1,5]], newInterval = [6,8] → Output: [[1,5],[6,8]]

### 26.3 Algorithm

1. Add intervals before new interval.
2. Merge overlapping intervals with new interval.
3. Add remaining intervals.

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

### 26.4 Python Solution

```

1 def insert(intervals, newInterval):
2     result = []
3     i = 0
4     n = len(intervals)
5
6     # Add non-overlapping intervals before
7     # newInterval
8     while i < n and intervals[i][1] < newInterval
9         [0]:
10         result.append(intervals[i])
11         i += 1
12
13     # Merge overlapping intervals

```

```

12     while i < n and intervals[i][0] <= newInterval
13         [1]:
14             newInterval[0] = min(newInterval[0],
15                                   intervals[i][0])
16             newInterval[1] = max(newInterval[1],
17                                   intervals[i][1])
18             i += 1
19     result.append(newInterval)
20
21     # Add remaining intervals
22     while i < n:
23         result.append(intervals[i])
24         i += 1
25
26     return result

```

## 27 Meeting Rooms

### 27.1 Problem Statement

Given an array of meeting intervals, determine if a person can attend all meetings.

### 27.2 Dry Run on Test Cases

- **Test Case 1:** intervals = [[0,30],[5,10],[15,20]] → Output: False
- **Test Case 2:** intervals = [[7,10],[2,4]] → Output: True
- **Test Case 3:** intervals = [] → Output: True
- **Test Case 4:** intervals = [[1,2]] → Output: True

### 27.3 Algorithm

1. Sort intervals by start time.
2. Check if any two meetings overlap.

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

### 27.4 Python Solution

```

1 def can_attend_meetings(intervals):
2     intervals.sort(key=lambda x: x[0])
3     for i in range(1, len(intervals)):
4         if intervals[i][0] < intervals[i-1][1]:
5             return False
6     return True

```

## 28 Meeting Rooms II

### 28.1 Problem Statement

Given an array of meeting intervals, find minimum number of conference rooms needed.

### 28.2 Dry Run on Test Cases

- **Test Case 1:** intervals =  $[[0,30],[5,10],[15,20]] \rightarrow$  Output: 2
- **Test Case 2:** intervals =  $[[7,10],[2,4]] \rightarrow$  Output: 1
- **Test Case 3:** intervals =  $[] \rightarrow$  Output: 0
- **Test Case 4:** intervals =  $[[1,5],[5,10]] \rightarrow$  Output: 1

### 28.3 Algorithm

1. Sort start and end times separately.
2. Use two pointers to track active meetings.
3. Max overlap gives number of rooms needed.

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

### 28.4 Python Solution

```
1 def min_meeting_rooms(intervals):
2     if not intervals:
3         return 0
4     start = sorted(s for s, e in intervals)
5     end = sorted(e for s, e in intervals)
6     rooms = 0
7     max_rooms = 0
8     i = j = 0
9     while i < len(intervals):
10         if start[i] < end[j]:
11             rooms += 1
12             max_rooms = max(max_rooms, rooms)
13             i += 1
14         else:
15             rooms -= 1
16             j += 1
17     return max_rooms
```

## 29 Spiral Matrix

## 29.1 Problem Statement

Given an  $m \times n$  matrix, return all elements in spiral order.

## 29.2 Dry Run on Test Cases

- **Test Case 1:** matrix =  $[[1,2,3],[4,5,6],[7,8,9]] \rightarrow$  Output:  $[1,2,3,6,9,8,7,4,5]$
- **Test Case 2:** matrix =  $[[1,2,3,4],[5,6,7,8],[9,10,11,12]] \rightarrow$  Output:  $[1,2,3,4,8,12,11,10,9,5,6,7]$
- **Test Case 3:** matrix =  $[] \rightarrow$  Output:  $[]$
- **Test Case 4:** matrix =  $[[1]] \rightarrow$  Output:  $[1]$

## 29.3 Algorithm

1. Use boundaries (top, bottom, left, right).
2. Traverse right, down, left, up, updating boundaries.
3. Stop when boundaries cross.

**Time Complexity:**  $O(m \cdot n)$     **Space Complexity:**  $O(1)$

## 29.4 Python Solution

```
1 def spiral_order(matrix):
2     if not matrix or not matrix[0]:
3         return []
4     result = []
5     top, bottom = 0, len(matrix) - 1
6     left, right = 0, len(matrix[0]) - 1
7
8     while top <= bottom and left <= right:
9         # Traverse right
10        for j in range(left, right + 1):
11            result.append(matrix[top][j])
12        top += 1
13        # Traverse down
14        for i in range(top, bottom + 1):
15            result.append(matrix[i][right])
16        right -= 1
17        if top <= bottom:
18            # Traverse left
19            for j in range(right, left - 1, -1):
20                result.append(matrix[bottom][j])
21            bottom -= 1
22        if left <= right:
23            # Traverse up
24            for i in range(bottom, top - 1, -1):
25                result.append(matrix[i][left])
```

```
26         left += 1
27
28     return result
```