Solutions to 25 Array-Based DSA Questions For 1-2 Years Experience Roles at EPAM Compiled on

September 26, 2025

Introduction

This document provides detailed solutions for 25 array-based Data Structures and Algorithms (DSA) problems, 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 and ease of understanding. The problems cover fundamental to intermediate array concepts frequently tested in technical interviews.

Contents

1 Find the Second Largest Element

1.1 Problem Statement

Given an array of integers, find and return the second largest distinct element. If fewer than 2 distinct elements exist, return -1. The array can contain duplicates and is non-empty.

1.2 Dry Run on Test Cases

- **Test Case 1**: Input = [3, 1, 4, 1, 5, 9] → Largest = 9, Second Largest = 5, Output: 5
- **Test Case 2**: Input = [10, 10, 10] → Only one distinct element, Output: -1
- **Test Case 3**: Input = [5, 3] → Largest = 5, Second Largest = 3, Output: 3
- **Test Case 4**: Input = [-1, -5, -3] → Largest = -1, Second Largest = -3, Output: -3

1.3 Algorithm

1. Initialize `first_max` and `second_max` to negative infinity.
2. Iterate through the array:

- If current element > `first_max`, update `second_max` = `first_max`, `first_max` = current.
- Else if current element > `second_max` and not equal to `first_max`, update `second_max`.

3. Return `second_max` if not negative infinity; else return -1.

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

1.4 Python Solution

```python
def second_largest(arr):
    if len(arr) < 2:
        return -1

    first_max = float('-inf')
    second_max = float('-inf')

    for num in arr:
        if num > first_max:
            second_max = first_max
            first_max = num
        elif num > second_max and num != first_max:
            second_max = num

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

# Example usage
print(second_largest([3, 1, 4, 1, 5, 9]))  # Output: 5
```

2 Rotate an Array by k Positions

2.1 Problem Statement

Given an array of integers and an integer k, rotate the array to the right by k steps. k can be larger than the array length, so handle modulo. Modify the array in-place.

2.2 Dry Run on Test Cases

- **Test Case 1**: Input = [1, 2, 3, 4, 5], k = 2 → Output: [4, 5, 1, 2, 3]
- **Test Case 2**: Input = [7, 8, 9], k = 4 → Effective k = 1 (4 % 3), Output: [9, 7, 8]
- **Test Case 3**: Input = [1], k = 5 → Output: [1]
- **Test Case 4**: Input = [-1, -2, -3], k = 0 → Output: [-1, -2, -3]

2.3 Algorithm

1. Compute effective k: $k = k\%\text{len(arr)}$.
2. Reverse the entire array.
3. Reverse the first k elements.
4. Reverse the remaining elements from k to end.

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

2.4 Python Solution

```python
def rotate_array(arr, k):
    n = len(arr)
    if n == 0:
        return
    k = k % n

    # Helper to reverse subarray
    def reverse(start, end):
        while start < end:
            arr[start], arr[end] = arr[end], arr[start]
            start += 1
            end -= 1

    reverse(0, n - 1)   # Reverse entire
    reverse(0, k - 1)   # Reverse first k
    reverse(k, n - 1)   # Reverse rest

# Example usage
arr = [1, 2, 3, 4, 5]
rotate_array(arr, 2)
print(arr)   # Output: [4, 5, 1, 2, 3]
```

3 Maximum Sum Subarray (Kadane's Algorithm)

3.1 Problem Statement

Given an array of integers (positive and negative), find the contiguous subarray with the largest sum and return that sum.

3.2 Dry Run on Test Cases

- **Test Case 1**: Input = [-2, 1, -3, 4, -1, 2, 1, -5, 4] → Max subarray [4, -1, 2, 1] = 6
- **Test Case 2**: Input = [1] → Output: 1
- **Test Case 3**: Input = [-1, -2, -3] → Output: -1

- **Test Case 4**: Input = [5, 4, -1, 7, 8] → Output: 23

3.3 Algorithm

1. Initialize `max_current` and `max_global` to first element.
2. For each element from second onwards:
   - `max_current` = max(element, `max_current` + element)
   - If `max_current` > `max_global`, update `max_global`.
3. Return `max_global`.

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

3.4 Python Solution

```python
def max_subarray_sum(arr):
    if not arr:
        return 0

    max_current = max_global = arr[0]

    for num in arr[1:]:
        max_current = max(num, max_current + num)
        if max_current > max_global:
            max_global = max_current

    return max_global

# Example usage
print(max_subarray_sum([-2, 1, -3, 4, -1, 2, 1, -5, 4]))  # Output: 6
```

4 Merge Two Sorted Arrays Without Extra Space

4.1 Problem Statement

Given two sorted arrays arr1 and arr2, merge them into arr1 (assuming arr1 has enough space at the end) without using extra space.

4.2 Dry Run on Test Cases

- **Test Case 1**: arr1 = [1, 3, 5, 7, 0, 0, 0], m = 4; arr2 = [2, 4, 6], n = 3 → arr1: [1, 2, 3, 4, 5, 6, 7]
- **Test Case 2**: arr1 = [1], m = 1; arr2 = [], n = 0 → arr1: [1]
- **Test Case 3**: arr1 = [0, 0], m = 0; arr2 = [2, 3], n = 2 → arr1: [2, 3]

- **Test Case 4**: arr1 = [4, 5, 6, 0, 0], m = 3; arr2 = [1, 2], n = 2 → arr1: [1, 2, 4, 5, 6]

4.3 Algorithm

1. Start from end: i = m-1 (arr1), j = n-1 (arr2), k = m+n-1 (arr1 end).
2. While i ≥ 0 and j ≥ 0:
   - If arr1[i] > arr2[j], arr1[k] = arr1[i], i–, k–
   - Else, arr1[k] = arr2[j], j–, k–
3. If j ≥ 0, copy remaining arr2 to arr1.

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

4.4 Python Solution

```python
def merge_sorted_arrays(arr1, m, arr2, n):
    i = m - 1
    j = n - 1
    k = m + n - 1

    while i >= 0 and j >= 0:
        if arr1[i] > arr2[j]:
            arr1[k] = arr1[i]
            i -= 1
        else:
            arr1[k] = arr2[j]
            j -= 1
        k -= 1

    while j >= 0:
        arr1[k] = arr2[j]
        j -= 1
        k -= 1

# Example usage
arr1 = [1, 3, 5, 7, 0, 0, 0]
arr2 = [2, 4, 6]
merge_sorted_arrays(arr1, 4, arr2, 3)
print(arr1)  # Output: [1, 2, 3, 4, 5, 6, 7]
```

5 Find Duplicates in an Array

5.1 Problem Statement

Given an array of integers where each integer is in [1, n] and n is the array length, find all duplicates (considering frequency).

5.2 Dry Run on Test Cases

- **Test Case 1**: Input = [4, 3, 2, 7, 8, 2, 3, 1] → Duplicates: [2, 3]
- **Test Case 2**: Input = [1, 2, 3] → No duplicates: []
- **Test Case 3**: Input = [1, 1, 1] → Duplicates: [1]
- **Test Case 4**: Input = [5, 4, 3, 2, 1, 5] → Duplicates: [5]

5.3 Algorithm

1. Use array as hash (values 1 to n).
2. For each num, go to index abs(num) - 1.
3. If arr[abs(num)-1] positive, make negative.
4. If already negative, num is duplicate, add to result.
5. Return unique duplicates.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$ (modifies input)

5.4 Python Solution

```python
def find_duplicates(arr):
    duplicates = []
    for num in arr:
        index = abs(num) - 1
        if arr[index] < 0:
            if abs(num) not in duplicates:
                duplicates.append(abs(num))
        else:
            arr[index] = -arr[index]
    return duplicates

# Example usage
print(find_duplicates([4, 3, 2, 7, 8, 2, 3, 1]))  # Output: [2, 3]
```

6 Move Zeros to the End

6.1 Problem Statement

Given an array of integers, move all zeros to the end while maintaining relative order of non-zero elements, in-place.

6.2 Dry Run on Test Cases

- **Test Case 1**: Input = [0, 1, 0, 3, 12] → Output: [1, 3, 12, 0, 0]

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

6.3 Algorithm

1. Use pointer `non_zero_index` starting at 0.
2. Iterate array: if current is non-zero, swap with arr[non_zero_index], increment `non_zero_index`.
3. Zeros move to end naturally.

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

6.4 Python Solution

```python
def move_zeros(arr):
    non_zero_index = 0
    for i in range(len(arr)):
        if arr[i] != 0:
            arr[non_zero_index], arr[i] = arr[i], arr[
               non_zero_index]
            non_zero_index += 1

# Example usage
arr = [0, 1, 0, 3, 12]
move_zeros(arr)
print(arr)  # Output: [1, 3, 12, 0, 0]
```

7 Find the Missing Number

7.1 Problem Statement

Given an array with n distinct numbers from 0 to n, find the missing number.

7.2 Dry Run on Test Cases

- **Test Case 1**: Input = [3, 0, 1] → Missing: 2
- **Test Case 2**: Input = [0, 1] → Missing: 2
- **Test Case 3**: Input = [9, 6, 4, 2, 3, 5, 7, 0, 1] → Missing: 8
- **Test Case 4**: Input = [1] → Missing: 0

7.3 Algorithm

1. Calculate expected sum = $n \cdot (n+1)/2$ (for 0 to n).
2. Compute actual sum of array.
3. Missing = expected - actual.

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

7.4 Python Solution

```python
def missing_number(arr):
    n = len(arr)
    expected_sum = n * (n + 1) // 2
    actual_sum = sum(arr)
    return expected_sum - actual_sum

# Example usage
print(missing_number([3, 0, 1]))  # Output: 2
```

8 Sort Array of 0s, 1s, and 2s (Dutch National Flag)

8.1 Problem Statement

Given an array with only 0s, 1s, and 2s, sort it in-place in one pass.

8.2 Dry Run on Test Cases

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

8.3 Algorithm

1. Use three pointers: low = 0, mid = 0, high = n-1.
2. While mid $\leq$ high:
    - If arr[mid] = 0, swap with low, low++, mid++
    - If arr[mid] = 1, mid++
    - If arr[mid] = 2, swap with high, high–

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

8.4 Python Solution

```python
def sort_colors(arr):
    low, mid, high = 0, 0, len(arr) - 1
    while mid <= high:
        if arr[mid] == 0:
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 1:
            mid += 1
        else:
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1

# Example usage
arr = [2, 0, 2, 1, 1, 0]
sort_colors(arr)
print(arr)  # Output: [0, 0, 1, 1, 2, 2]
```

9 Find Intersection of Two Arrays

9.1 Problem Statement

Given two arrays, find their intersection (common elements, considering frequency).

9.2 Dry Run on Test Cases

- **Test Case 1**: arr1 = [1, 2, 2, 1], arr2 = [2, 2] → Intersection: [2, 2]
- **Test Case 2**: arr1 = [4, 9, 5], arr2 = [9, 4, 9, 8, 4] → Intersection: [4, 9]
- **Test Case 3**: arr1 = [1], arr2 = [2] → []
- **Test Case 4**: arr1 = [1, 1], arr2 = [1] → [1]

9.3 Algorithm

1. Use hashmap to count frequency in smaller array.
2. Iterate second array: if in map and count > 0, add to result, decrement count.
3. Return result.

Time Complexity: $O(m+n)$ **Space Complexity**: $O(\min(m,n))$

9.4 Python Solution

```python
from collections import Counter

def array_intersection(arr1, arr2):
```

```
    if len(arr1) > len(arr2):
        arr1, arr2 = arr2, arr1

    count = Counter(arr1)
    result = []
    for num in arr2:
        if count[num] > 0:
            result.append(num)
            count[num] -= 1
    return result

# Example usage
print(array_intersection([1, 2, 2, 1], [2, 2]))  # Output: [2, 2]
```

10 Product of Array Elements Except Self

10.1 Problem Statement

Given an array of integers, return an array where each element is the product of all elements except itself, without division, O(1) extra space.

10.2 Dry Run on Test Cases

- **Test Case 1**: Input = [1, 2, 3, 4] → Output: [24, 12, 8, 6]
- **Test Case 2**: Input = [-1, 1, 0, -3, 3] → Output: [0, 0, 9, 0, 0]
- **Test Case 3**: Input = [5] → Output: [1]
- **Test Case 4**: Input = [2, 3] → Output: [3, 2]

10.3 Algorithm

1. Initialize result array of 1s.
2. Left pass: for i from 1 to n-1, result[i] = result[i-1] * arr[i-1].
3. Right pass: initialize right = 1, for i from n-1 to 0, result[i] *= right, right *= arr[i].

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$ extra

10.4 Python Solution

```
def product_except_self(arr):
    n = len(arr)
    result = [1] * n

    left = 1
    for i in range(1, n):
        left *= arr[i-1]
```

```
        result[i] = left

    right = 1
    for i in range(n-1, -1, -1):
        result[i] *= right
        right *= arr[i]

    return result

# Example usage
print(product_except_self([1, 2, 3, 4]))  # Output: [24, 12, 8, 6]
```

11 Trapping Rain Water

11.1 Problem Statement

Given an array of non-negative integers representing heights, compute how much water can be trapped after raining.

11.2 Dry Run on Test Cases

- **Test Case 1**: Input = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1] → Water = 6
- **Test Case 2**: Input = [4, 2, 0, 3, 2, 5] → Water = 9
- **Test Case 3**: Input = [1, 2, 3] → Water = 0
- **Test Case 4**: Input = [0, 0] → Water = 0

11.3 Algorithm

1. Use two pointers: left = 0, right = n-1, left_max = right_max = 0.
2. While left < right:
    - If arr[left] < arr[right]:
        - If arr[left] ≥ left_max, update left_max.
        - Else, add (left_max - arr[left]) to water.
        - left++
    - Else:
        - If arr[right] ≥ right_max, update right_max.
        - Else, add (right_max - arr[right]) to water.

– right–

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

11.4 Python Solution

```python
def trap_rain_water(height):
    if not height:
        return 0

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

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

# Example usage
print(trap_rain_water([0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]))  # Output: 6
```

12 Best Time to Buy and Sell Stock

12.1 Problem Statement

Given an array of stock prices, find the maximum profit from one buy and one sell.

12.2 Dry Run on Test Cases

- **Test Case 1**: Input = [7, 1, 5, 3, 6, 4] → Buy at 1, sell at 6, Profit = 5
- **Test Case 2**: Input = [7, 6, 4, 3, 1] → No profit, Output: 0
- **Test Case 3**: Input = [1] → Output: 0
- **Test Case 4**: Input = [2, 4, 1] → Profit = 2

12.3 Algorithm

1. Initialize min_price to first element, max_profit to 0.
2. For each price:
   - Update min_price if current $<$ min_price.
   - Update max_profit if (current - min_price) $>$ max_profit.
3. Return max_profit.

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

12.4 Python Solution

```python
def max_profit(prices):
    if not prices:
        return 0

    min_price = prices[0]
    max_profit = 0

    for price in prices[1:]:
        if price < min_price:
            min_price = price
        else:
            max_profit = max(max_profit, price - min_price)

    return max_profit

# Example usage
print(max_profit([7, 1, 5, 3, 6, 4]))  # Output: 5
```

13 Container with Most Water

13.1 Problem Statement

Given an array of heights, find two lines that form a container with the most water (area = min(height) * distance).

13.2 Dry Run on Test Cases

- **Test Case 1**: Input = [1, 8, 6, 2, 5, 4, 8, 3, 7] $\rightarrow$ Max area = 49
- **Test Case 2**: Input = [1, 1] $\rightarrow$ Area = 1
- **Test Case 3**: Input = [4, 3, 2, 1, 4] $\rightarrow$ Area = 16
- **Test Case 4**: Input = [1] $\rightarrow$ Area = 0

13.3 Algorithm

1. Use two pointers: left = 0, right = n-1.
2. While left < right:
    - Compute area = min(arr[left], arr[right]) * (right - left).
    - Update max_area if current area > max_area.
    - Move pointer with smaller height inward.

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

13.4 Python Solution

```python
def max_area(height):
    left, right = 0, len(height) - 1
    max_area = 0

    while left < right:
        area = min(height[left], height[right]) * (right - left)
        max_area = max(max_area, area)
        if height[left] < height[right]:
            left += 1
        else:
            right -= 1
    return max_area

# Example usage
print(max_area([1, 8, 6, 2, 5, 4, 8, 3, 7]))  # Output: 49
```

14 Find Pairs with Given Sum

14.1 Problem Statement

Given an array and a target sum, find all pairs that sum to the target.

14.2 Dry Run on Test Cases

- **Test Case 1**: arr = [1, 5, 7, -1], target = 6 → Pairs: [(1, 5), (-1, 7)]
- **Test Case 2**: arr = [2, 3, 4], target = 10 → []
- **Test Case 3**: arr = [0, 0], target = 0 → [(0, 0)]
- **Test Case 4**: arr = [3], target = 6 → []

14.3 Algorithm

1. Use hashmap to store frequency of numbers.

2. For each num, check if (target - num) exists in map.
3. Handle duplicates carefully (e.g., target = 8, num = 4).
4. Return list of pairs.

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

14.4 Python Solution

```python
from collections import Counter

def find_pairs(arr, target):
    count = Counter(arr)
    pairs = []

    for num in arr:
        complement = target - num
        if complement in count and count[complement] > 0:
            if num == complement and count[num] > 1:
                pairs.append((num, complement))
                count[num] -= 1
            elif num != complement and count[num] > 0:
                pairs.append((num, complement))
                count[num] -= 1
                count[complement] -= 1
    return pairs

# Example usage
print(find_pairs([1, 5, 7, -1], 6))  # Output: [(1, 5), (-1, 7)]
```

15 Remove Duplicates from Sorted Array

15.1 Problem Statement

Given a sorted array, remove duplicates in-place and return new length.

15.2 Dry Run on Test Cases

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

15.3 Algorithm

1. If array empty, return 0.

2. Use pointer `write` = 1.
3. Iterate from i = 1: if arr[i] != arr[i-1], copy to arr[write], increment write.
4. Return write.

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

15.4 Python Solution

```python
def remove_duplicates(arr):
    if not arr:
        return 0

    write = 1
    for i in range(1, len(arr)):
        if arr[i] != arr[i-1]:
            arr[write] = arr[i]
            write += 1
    return write

# Example usage
arr = [1, 1, 2]
length = remove_duplicates(arr)
print(length, arr[:length])  # Output: 2, [1, 2]
```

16 Find kth Largest Element

16.1 Problem Statement

Given an array and integer k, find the kth largest element.

16.2 Dry Run on Test Cases

- **Test Case 1**: arr = [3, 2, 1, 5, 6, 4], k = 2 → Output: 5
- **Test Case 2**: arr = [3, 2, 3, 1, 2, 4, 5, 5, 6], k = 4 → Output: 4
- **Test Case 3**: arr = [1], k = 1 → Output: 1
- **Test Case 4**: arr = [7, 4, 6], k = 2 → Output: 6

16.3 Algorithm

1. Use quickselect with random pivot.
2. Partition array around pivot, get pivot index.
3. If pivot index = n-k, return pivot.

4. Else recurse on left or right partition.

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

16.4 Python Solution

```python
import random

def find_kth_largest(arr, k):
    def quickselect(left, right, k_smallest):
        if left == right:
            return arr[left]

        pivot_idx = random.randint(left, right)
        arr[pivot_idx], arr[right] = arr[right], arr[pivot_idx]
        pivot = arr[right]

        i = left
        for j in range(left, right):
            if arr[j] <= pivot:
                arr[i], arr[j] = arr[j], arr[i]
                i += 1
        arr[i], arr[right] = arr[right], arr[i]

        if i == k_smallest:
            return arr[i]
        elif i > k_smallest:
            return quickselect(left, i - 1, k_smallest)
        else:
            return quickselect(i + 1, right, k_smallest)

    return quickselect(0, len(arr) - 1, len(arr) - k)

# Example usage
print(find_kth_largest([3, 2, 1, 5, 6, 4], 2))  # Output: 5
```

17 Subarray with Sum k

17.1 Problem Statement

Given an array of integers and a target k, find the number of subarrays with sum k.

17.2 Dry Run on Test Cases

- **Test Case 1**: arr = [1, 1, 1], k = 2 → Output: 2 ([1, 1])
- **Test Case 2**: arr = [1, 2, 3], k = 3 → Output: 2 ([1, 2], [3])
- **Test Case 3**: arr = [1], k = 2 → Output: 0

- **Test Case 4**: arr = [-1, -1, 1], k = 0 → Output: 1

17.3 Algorithm

1. Use hashmap to store cumulative sum frequencies.
2. Initialize sum = 0, count = 0.
3. For each num, update sum, check if (sum - k) in map, add map[sum - k] to count.
4. Update map with current sum.

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

17.4 Python Solution

```python
from collections import defaultdict

def subarray_sum(arr, k):
    count = 0
    curr_sum = 0
    sum_map = defaultdict(int)
    sum_map[0] = 1

    for num in arr:
        curr_sum += num
        if curr_sum - k in sum_map:
            count += sum_map[curr_sum - k]
        sum_map[curr_sum] += 1
    return count

# Example usage
print(subarray_sum([1, 1, 1], 2))  # Output: 2
```

18 Longest Consecutive Sequence

18.1 Problem Statement

Given an unsorted array, find the length of the longest consecutive elements sequence.

18.2 Dry Run on Test Cases

- **Test Case 1**: Input = [100, 4, 200, 1, 3, 2] → Sequence [1, 2, 3, 4], Length = 4
- **Test Case 2**: Input = [0, 3, 7, 2, 5, 8, 4, 6, 0, 1] → Length = 9
- **Test Case 3**: Input = [1] → Length = 1
- **Test Case 4**: Input = [] → Length = 0

18.3 Algorithm

1. Convert array to set for O(1) lookup.
2. For each num, if num-1 not in set, check sequence length starting from num.
3. Update max length.

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

18.4 Python Solution

```python
def longest_consecutive(arr):
    if not arr:
        return 0

    num_set = set(arr)
    max_length = 0

    for num in num_set:
        if num - 1 not in num_set:
            curr_num = num
            curr_length = 1
            while curr_num + 1 in num_set:
                curr_num += 1
                curr_length += 1
            max_length = max(max_length, curr_length)
    return max_length

# Example usage
print(longest_consecutive([100, 4, 200, 1, 3, 2]))  # Output: 4
```

19 Rotate Matrix by 90 Degrees

19.1 Problem Statement

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

19.2 Dry Run on Test Cases

- **Test Case 1**: Input = [[1,2,3],[4,5,6],[7,8,9]] → Output: [[7,4,1],[8,5,2],[9,6,3]]
- **Test Case 2**: Input = [[1]] → Output: [[1]]
- **Test Case 3**: Input = [[1,2],[3,4]] → Output: [[3,1],[4,2]]
- **Test Case 4**: Input = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]] → Rotated matrix

19.3 Algorithm

1. Transpose matrix (swap elements across diagonal).
2. Reverse each row.

Time Complexity: $O(n^2)$ **Space Complexity**: $O(1)$

19.4 Python Solution

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

    # Transpose
    for i in range(n):
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][
               j]

    # Reverse each row
    for i in range(n):
        matrix[i].reverse()

# Example usage
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
rotate_matrix(matrix)
print(matrix)  # Output: [[7, 4, 1], [8, 5, 2], [9, 6, 3]]
```

20 Spiral Traversal of Matrix

20.1 Problem Statement

Given an m x n matrix, return all elements in spiral order (clockwise from top-left).

20.2 Dry Run on Test Cases

- **Test Case 1**: Input = [[1,2,3],[4,5,6],[7,8,9]] → Output: [1,2,3,6,9,8,7,4,5]
- **Test Case 2**: Input = [[1,2],[3,4]] → Output: [1,2,4,3]
- **Test Case 3**: Input = [[1]] → Output: [1]
- **Test Case 4**: Input = [[1,2,3]] → Output: [1,2,3]

20.3 Algorithm

1. Initialize boundaries: top, bottom, left, right.
2. While top $\leq$ bottom and left $\leq$ right:
    - Traverse right, top++, left to right.

- Traverse down, right–, top to bottom.
- Traverse left, bottom–, right to left.
- Traverse up, left++, bottom to top.

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

20.4 Python Solution

```python
def spiral_order(matrix):
    if not matrix:
        return []

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

    while top <= bottom and left <= right:
        # Traverse right
        for i in range(left, right + 1):
            result.append(matrix[top][i])
        top += 1
        # Traverse down
        if top <= bottom:
            for i in range(top, bottom + 1):
                result.append(matrix[i][right])
            right -= 1
        # Traverse left
        if top <= bottom and left <= right:
            for i in range(right, left - 1, -1):
                result.append(matrix[bottom][i])
            bottom -= 1
        # Traverse up
        if top <= bottom and left <= right:
            for i in range(bottom, top - 1, -1):
                result.append(matrix[i][left])
            left += 1
    return result

# Example usage
print(spiral_order([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))  # Output:
    [1, 2, 3, 6, 9, 8, 7, 4, 5]
```

21 Maximum Area of Island

21.1 Problem Statement

Given a binary matrix (0s and 1s), find the maximum area of an island (connected 1s).

21.2 Dry Run on Test Cases

- **Test Case 1**: Input = [[0,0,1,0],[0,1,1,0],[0,0,0,0]] → Max area = 2
- **Test Case 2**: Input = [[0,0,0],[0,0,0]] → Max area = 0
- **Test Case 3**: Input = [[1]] → Max area = 1
- **Test Case 4**: Input = [[1,1],[1,1]] → Max area = 4

21.3 Algorithm

1. Iterate through each cell.
2. If cell = 1, use DFS to compute area, mark visited cells.
3. Track max area.

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

21.4 Python Solution

```python
def max_area_of_island(grid):
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    max_area = 0

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols or grid[i][j]
            != 1:
            return 0
        grid[i][j] = 0  # Mark visited
        return 1 + dfs(i+1, j) + dfs(i-1, j) + dfs(i, j+1) + dfs(
           i, j-1)

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
                max_area = max(max_area, dfs(i, j))
    return max_area

# Example usage
grid = [[0,0,1,0],[0,1,1,0],[0,0,0,0]]
print(max_area_of_island(grid))  # Output: 2
```

22 Search in Row-Wise and Column-Wise Sorted Matrix

22.1 Problem Statement

Given an m x n matrix sorted row-wise and column-wise, search for a target.

22.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[10,20,30],[15,25,35],[27,29,37]], target = 25 → True
- **Test Case 2**: matrix = [[1,3],[2,4]], target = 5 → False
- **Test Case 3**: matrix = [[1]], target = 1 → True
- **Test Case 4**: matrix = [], target = 1 → False

22.3 Algorithm

1. Start from top-right (row = 0, col = n-1).
2. While row $<$ m and col $\geq$ 0:
   - If matrix[row][col] = target, return True.
   - If $>$ target, col–.
   - If $<$ target, row++.
3. Return False.

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

22.4 Python Solution

```python
def search_matrix(matrix, target):
    if not matrix or not matrix[0]:
        return False

    m, n = len(matrix), len(matrix[0])
    row, col = 0, n - 1

    while row < m and col >= 0:
        if matrix[row][col] == target:
            return True
        elif matrix[row][col] > target:
            col -= 1
        else:
            row += 1
    return False

# Example usage
matrix = [[10, 20, 30], [15, 25, 35], [27, 29, 37]]
print(search_matrix(matrix, 25))  # Output: True
```

23 Merge Overlapping Intervals

23.1 Problem Statement

Given a collection of intervals, merge overlapping intervals.

23.2 Dry Run on Test Cases

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

23.3 Algorithm

1. Sort intervals by start time.
2. Initialize result with first interval.
3. For each interval, merge with last in result if overlapping, else append.

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

23.4 Python Solution

```python
def merge_intervals(intervals):
    if not intervals:
        return []

    intervals.sort(key=lambda x: x[0])
    result = [intervals[0]]

    for curr in intervals[1:]:
        if curr[0] <= result[-1][1]:
            result[-1][1] = max(result[-1][1], curr[1])
        else:
            result.append(curr)
    return result

# Example usage
print(merge_intervals([[1, 3], [2, 6], [8, 10], [15, 18]]))  # Output: [[1, 6], [8, 10], [15, 18]]
```

24 Minimum Size Subarray Sum

24.1 Problem Statement

Given an array of positive integers and a target sum, find the minimum length of a contiguous subarray with sum $\geq$ target.

24.2 Dry Run on Test Cases

- **Test Case 1**: arr = [2,3,1,2,4,3], target = 7 $\rightarrow$ Output: 2 ([4,3])
- **Test Case 2**: arr = [1,4,4], target = 4 $\rightarrow$ Output: 1
- **Test Case 3**: arr = [1,1,1], target = 5 $\rightarrow$ Output: 0
- **Test Case 4**: arr = [1], target = 1 $\rightarrow$ Output: 1

24.3 Algorithm

1. Use two pointers: left, right.
2. Maintain current sum, min_length = infinity.
3. Move right to add elements; if sum $\geq$ target, update min_length, shrink left.

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

24.4 Python Solution

```python
def min_subarray_len(target, arr):
    if not arr:
        return 0

    min_length = float('inf')
    curr_sum = 0
    left = 0

    for right in range(len(arr)):
        curr_sum += arr[right]
        while curr_sum >= target:
            min_length = min(min_length, right - left + 1)
            curr_sum -= arr[left]
            left += 1
    return min_length if min_length != float('inf') else 0

# Example usage
print(min_subarray_len(7, [2, 3, 1, 2, 4, 3]))  # Output: 2
```

25 Stock Span Problem

25.1 Problem Statement

Given an array of stock prices, return an array where span[i] is the number of consecutive days for which the price for the day i is less than or equal to the price of day i+1.

25.2 Dry Run on Test Cases

- **Test Case 1**: Input = [100, 80, 60, 70, 60, 75, 85] → Output: [1, 1, 2, 1, 4, 2, 1]
- **Test Case 2**: Input = [10, 20, 30] → Output: [3, 2, 1]
- **Test Case 3**: Input = [100] → Output: [1]
- **Test Case 4**: Input = [30, 20, 10] → Output: [1, 1, 1]

25.3 Algorithm

1. Use a stack to store indices of prices.
2. For each price, pop stack while price >= stack top price.
3. Span[i] = i - stack top (or i + 1 if stack empty).
4. Push i to stack.

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

25.4 Python Solution

```python
def stock_span(prices):
    n = len(prices)
    result = [1] * n
    stack = [0]

    for i in range(1, n):
        while stack and prices[i] >= prices[stack[-1]]:
            stack.pop()
        result[i] = i - (stack[-1] if stack else -1)
        stack.append(i)
    return result

# Example usage
print(stock_span([100, 80, 60, 70, 60, 75, 85]))  # Output: [1,
   1, 2, 1, 4, 2, 1]
```

Solutions to DSA Questions 26-50 (Strings and Linked Lists) For 1-2 Years

Experience Roles at EPAM Compiled on September 26, 2025

Introduction

This document provides detailed solutions for 25 Data Structures and Algorithms (DSA) problems (questions 26 to 50) from the Strings and Linked Lists 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 and ease of understanding. These problems cover fundamental to intermediate concepts frequently tested in technical interviews.

Contents

1 Check if a String is a Palindrome

1.1 Problem Statement

Given a string, determine if it is a palindrome (reads the same forward and backward), considering only alphanumeric characters and ignoring cases.

1.2 Dry Run on Test Cases

- **Test Case 1**: Input = "A man, a plan, a canal: Panama" → Output: True
- **Test Case 2**: Input = "race a car" → Output: False
- **Test Case 3**: Input = "" → Output: True
- **Test Case 4**: Input = "0P" → Output: False

1.3 Algorithm

1. Convert string to lowercase and filter alphanumeric characters.
2. Use two pointers: left from start, right from end.
3. Compare characters; if mismatch, return False.
4. If pointers meet, return True.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$ or $O(n)$ if new string created

1.4 Python Solution

```python
def is_palindrome(s):
    # Filter alphanumeric and convert to lowercase
    filtered = ''.join(c.lower() for c in s if c.isalnum())
    left, right = 0, len(filtered) - 1

    while left < right:
        if filtered[left] != filtered[right]:
            return False
        left += 1
        right -= 1
    return True

# Example usage
print(is_palindrome("A man, a plan, a canal: Panama"))  # Output:
    True
```

2 Reverse Words in a String

2.1 Problem Statement

Given a string, reverse the order of words, removing extra spaces.

2.2 Dry Run on Test Cases

- **Test Case 1**: Input = "the sky is blue" → Output: "blue is sky the"
- **Test Case 2**: Input = " hello world " → Output: "world hello"
- **Test Case 3**: Input = "a" → Output: "a"
- **Test Case 4**: Input = "" → Output: ""

2.3 Algorithm

1. Split string into words, filter out empty strings.
2. Reverse the list of words.
3. Join words with single space.

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

2.4 Python Solution

```python
def reverse_words(s):
    words = [word for word in s.split() if word]
```

```
    words.reverse()
    return ' '.join(words)

# Example usage
print(reverse_words("the sky is blue"))  # Output: "blue is sky
    the"
```

3 Longest Substring Without Repeating Characters

3.1 Problem Statement

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

3.2 Dry Run on Test Cases

- **Test Case 1**: Input = "abcabcbb" → Output: 3 ("abc")
- **Test Case 2**: Input = "bbbbb" → Output: 1 ("b")
- **Test Case 3**: Input = "pwwkew" → Output: 3 ("wke")
- **Test Case 4**: Input = "" → Output: 0

3.3 Algorithm

1. Use sliding window with hashmap to store last seen index of characters.
2. Move right pointer, update max length.
3. If character repeats, move left pointer to last seen + 1.

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

3.4 Python Solution

```
def length_of_longest_substring(s):
    char_index = {}
    max_length = 0
    left = 0

    for right, char in enumerate(s):
        if char in char_index and char_index[char] >= left:
            left = char_index[char] + 1
        else:
            max_length = max(max_length, right - left + 1)
        char_index[char] = right
    return max_length

# Example usage
print(length_of_longest_substring("abcabcbb"))  # Output: 3
```

4 Valid Parentheses

4.1 Problem Statement

Given a string containing only '(', ')', '', '', '[', ']', determine if it is valid (matching pairs).

4.2 Dry Run on Test Cases

- **Test Case 1**: Input = "()" → Output: True
- **Test Case 2**: Input = "()[]" → Output: True
- **Test Case 3**: Input = "(]" → Output: False
- **Test Case 4**: Input = "([)]" → Output: False

4.3 Algorithm

1. Use a stack to track opening brackets.
2. For each character:
   - If opening, push to stack.
   - If closing, check if matches stack top; pop if match, else return False.
3. Return True if stack empty.

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

4.4 Python Solution

```python
def is_valid(s):
    stack = []
    brackets = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in brackets.values():
            stack.append(char)
        elif char in brackets:
            if not stack or stack.pop() != brackets[char]:
                return False
    return len(stack) == 0

# Example usage
print(is_valid("()[]{}"))  # Output: True
```

5 Longest Palindromic Substring

5.1 Problem Statement

Given a string, find the longest substring that is a palindrome.

5.2 Dry Run on Test Cases

- **Test Case 1**: Input = "babad" → Output: "bab" or "aba"
- **Test Case 2**: Input = "cbbd" → Output: "bb"
- **Test Case 3**: Input = "a" → Output: "a"
- **Test Case 4**: Input = "" → Output: ""

5.3 Algorithm

1. For each index, expand around center for odd and even length palindromes.
2. Track max length and substring.
3. Return longest palindrome found.

Time Complexity: $O(n^2)$ **Space Complexity**: $O(1)$

5.4 Python Solution

```python
def longest_palindrome(s):
    def expand_around_center(left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return left + 1, right - 1

    start, end = 0, 0
    for i in range(len(s)):
        left1, right1 = expand_around_center(i, i)  # Odd length
        left2, right2 = expand_around_center(i, i + 1)  # Even length
        if right1 - left1 > end - start:
            start, end = left1, right1
        if right2 - left2 > end - start:
            start, end = left2, right2
    return s[start:end + 1]

# Example usage
print(longest_palindrome("babad"))  # Output: "bab" or "aba"
```

6 Generate All Permutations of a String

6.1 Problem Statement

Given a string, return all possible permutations.

6.2 Dry Run on Test Cases

- **Test Case 1**: Input = "abc" → Output: ["abc", "acb", "bac", "bca", "cab", "cba"]
- **Test Case 2**: Input = "a" → Output: ["a"]
- **Test Case 3**: Input = "" → Output: []
- **Test Case 4**: Input = "aa" → Output: ["aa"]

6.3 Algorithm

1. Use backtracking: swap characters at each position.
2. Recurse to generate permutations for remaining characters.
3. Collect all permutations in result.

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

6.4 Python Solution

```python
def permute(s):
    def backtrack(arr, start, result):
        if start == len(arr):
            result.append(''.join(arr))
        for i in range(start, len(arr)):
            arr[start], arr[i] = arr[i], arr[start]
            backtrack(arr, start + 1, result)
            arr[start], arr[i] = arr[i], arr[start]

    result = []
    backtrack(list(s), 0, result)
    return result

# Example usage
print(permute("abc"))  # Output: ["abc", "acb", "bac", "bca", "
    cab", "cba"]
```

7 Check if Strings are Rotations of Each Other

7.1 Problem Statement

Given two strings, check if one is a rotation of the other.

7.2 Dry Run on Test Cases

- **Test Case 1**: s1 = "abcde", s2 = "cdeab" → Output: True
- **Test Case 2**: s1 = "abcde", s2 = "abced" → Output: False
- **Test Case 3**: s1 = "", s2 = "" → Output: True
- **Test Case 4**: s1 = "a", s2 = "a" → Output: True

7.3 Algorithm

1. Check if lengths are equal; if not, return False.
2. Concatenate s1 with itself.
3. Check if s2 is a substring of s1 + s1.

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

7.4 Python Solution

```python
def are_rotations(s1, s2):
    if len(s1) != len(s2):
        return False
    if not s1 and not s2:
        return True
    return s2 in (s1 + s1)

# Example usage
print(are_rotations("abcde", "cdeab"))  # Output: True
```

8 Find First Non-Repeating Character

8.1 Problem Statement

Given a string, find the index of the first non-repeating character.

8.2 Dry Run on Test Cases

- **Test Case 1**: Input = "leetcode" → Output: 0 ('l')
- **Test Case 2**: Input = "loveleetcode" → Output: 2 ('v')
- **Test Case 3**: Input = "aabb" → Output: -1
- **Test Case 4**: Input = "" → Output: -1

8.3 Algorithm

1. Use hashmap to count character frequencies.

2. Iterate string again to find first character with count 1.
3. Return its index or -1 if none.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$ (26 chars max)

8.4 Python Solution

```python
def first_non_repeating(s):
    count = {}
    for char in s:
        count[char] = count.get(char, 0) + 1

    for i, char in enumerate(s):
        if count[char] == 1:
            return i
    return -1

# Example usage
print(first_non_repeating("leetcode"))  # Output: 0
```

9 String to Integer (atoi)

9.1 Problem Statement

Convert a string to a 32-bit signed integer, handling whitespace, signs, and overflow.

9.2 Dry Run on Test Cases

- **Test Case 1**: Input = "42" → Output: 42
- **Test Case 2**: Input = " -42" → Output: -42
- **Test Case 3**: Input = "4193 with words" → Output: 4193
- **Test Case 4**: Input = "2147483648" → Output: 2147483647

9.3 Algorithm

1. Strip leading whitespace.
2. Check sign (+ or -).
3. Build number digit by digit, check for overflow.
4. Return number or clamp to 32-bit range.

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

9.4 Python Solution

```python
def atoi(s):
    s = s.strip()
    if not s:
        return 0

    sign = 1
    i = 0
    if s[0] in ['+', '-']:
        sign = -1 if s[0] == '-' else 1
        i += 1

    result = 0
    while i < len(s) and s[i].isdigit():
        result = result * 10 + int(s[i])
        if result * sign > 2**31 - 1:
            return 2**31 - 1
        if result * sign < -2**31:
            return -2**31
        i += 1
    return result * sign

# Example usage
print(atoi("   -42"))  # Output: -42
```

10 Longest Common Prefix

10.1 Problem Statement

Given an array of strings, find the longest common prefix among them.

10.2 Dry Run on Test Cases

- **Test Case 1**: Input = ["flower", "flow", "flight"] → Output: "fl"
- **Test Case 2**: Input = ["dog", "racecar", "car"] → Output: ""
- **Test Case 3**: Input = ["interspecies", "interstellar"] → Output: "inter"
- **Test Case 4**: Input = ["a"] → Output: "a"

10.3 Algorithm

1. If empty array, return "".
2. Take first string as prefix.
3. For each string, reduce prefix while it doesn't match.

Time Complexity: $O(S)$ (S = total characters) **Space Complexity**: $O(1)$

10.4 Python Solution

```python
def longest_common_prefix(strs):
    if not strs:
        return ""
    prefix = strs[0]

    for s in strs[1:]:
        while s[:len(prefix)] != prefix:
            prefix = prefix[:-1]
            if not prefix:
                return ""
    return prefix

# Example usage
print(longest_common_prefix(["flower", "flow", "flight"]))  # Output: "fl"
```

11 Group Anagrams

11.1 Problem Statement

Given an array of strings, group anagrams together.

11.2 Dry Run on Test Cases

- **Test Case 1**: Input = ["eat", "tea", "tan", "ate", "nat", "bat"] → Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]
- **Test Case 2**: Input = [""] → Output: [[""]]
- **Test Case 3**: Input = ["a"] → Output: [["a"]]
- **Test Case 4**: Input = [] → Output: []

11.3 Algorithm

1. Use hashmap with sorted string as key, list of strings as value.
2. For each string, sort and add to map.
3. Return map values.

Time Complexity: $O(n \cdot k \log k)$ (k = max string length) **Space Complexity**: $O(n \cdot k)$

11.4 Python Solution

```python
def group_anagrams(strs):
    anagrams = {}
    for s in strs:
```

```
        key = ''.join(sorted(s))
        anagrams[key] = anagrams.get(key, []) + [s]
    return list(anagrams.values())

# Example usage
print(group_anagrams(["eat", "tea", "tan", "ate", "nat", "bat"]))
```

12 Valid IP Address

12.1 Problem Statement

Given a string, determine if it is a valid IPv4 address.

12.2 Dry Run on Test Cases

- **Test Case 1**: Input = "192.168.1.1" → Output: True
- **Test Case 2**: Input = "192.168.01.1" → Output: False
- **Test Case 3**: Input = "256.1.2.3" → Output: False
- **Test Case 4**: Input = "1.2.3" → Output: False

12.3 Algorithm

1. Split string by ”.” and check for 4 parts.
2. For each part:
   - Check length, leading zeros, and range (0-255).
   - Ensure only digits.
3. Return True if all valid.

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

12.4 Python Solution

```
def valid_ip_address(s):
    parts = s.split('.')
    if len(parts) != 4:
        return False

    for part in parts:
        if not part or (part[0] == '0' and len(part) > 1) or not
           part.isdigit():
            return False
        num = int(part)
        if num < 0 or num > 255:
```

```
            return False
    return True

# Example usage
print(valid_ip_address("192.168.1.1"))  # Output: True
```

13 Edit Distance

13.1 Problem Statement

Given two strings, find minimum operations (insert, delete, replace) to convert one to another.

13.2 Dry Run on Test Cases

- **Test Case 1**: word1 = "horse", word2 = "ros" → Output: 3
- **Test Case 2**: word1 = "intention", word2 = "execution" → Output: 5
- **Test Case 3**: word1 = "", word2 = "abc" → Output: 3
- **Test Case 4**: word1 = "a", word2 = "a" → Output: 0

13.3 Algorithm (Memoization)

1. Use recursive function with memoization.
2. If strings empty, return length of other.
3. If characters match, recurse on rest.
4. Else, take min of insert, delete, replace.

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

13.4 Python Solution (Memoization)

```
def edit_distance(word1, word2):
    memo = {}

    def dp(i, j):
        if i == 0:
            return j
        if j == 0:
            return i
        if (i, j) in memo:
            return memo[(i, j)]

        if word1[i-1] == word2[j-1]:
            memo[(i, j)] = dp(i-1, j-1)
```

```
        else:
            memo[(i, j)] = min(
                dp(i-1, j) + 1,   # Delete
                dp(i, j-1) + 1,   # Insert
                dp(i-1, j-1) + 1  # Replace
            )
        return memo[(i, j)]

    return dp(len(word1), len(word2))

# Example usage
print(edit_distance("horse", "ros"))  # Output: 3
```

13.5 Python Solution (Tabulation)

```
def edit_distance_tab(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i-1] == word2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = min(
                    dp[i-1][j] + 1,   # Delete
                    dp[i][j-1] + 1,   # Insert
                    dp[i-1][j-1] + 1  # Replace
                )
    return dp[m][n]

# Example usage
print(edit_distance_tab("horse", "ros"))  # Output: 3
```

14 Smallest Window Containing All Characters

14.1 Problem Statement

Given two strings s and t, find the smallest window in s containing all characters of t.

14.2 Dry Run on Test Cases

- **Test Case 1**: s = "ADOBECODEBANC", t = "ABC" → Output: "BANC"

- **Test Case 2**: s = "a", t = "a" → Output: "a"
- **Test Case 3**: s = "a", t = "aa" → Output: ""
- **Test Case 4**: s = "abc", t = "d" → Output: ""

14.3 Algorithm

1. Use sliding window with two hashmaps.
2. Move right pointer until window contains all t characters.
3. Shrink left pointer to minimize window.
4. Track smallest window.

Time Complexity: $O(n)$ **Space Complexity**: $O(k)$ (k = charset size)

14.4 Python Solution

```
from collections import Counter

def min_window(s, t):
    if not s or not t:
        return ""

    t_count = Counter(t)
    required = len(t_count)
    formed = 0
    window_counts = {}

    left = right = 0
    min_len = float('inf')
    min_window_substr = ""

    while right < len(s):
        window_counts[s[right]] = window_counts.get(s[right], 0)
           + 1
        if s[right] in t_count and window_counts[s[right]] ==
           t_count[s[right]]:
            formed += 1

        while left <= right and formed == required:
            if right - left + 1 < min_len:
                min_len = right - left + 1
                min_window_substr = s[left:right + 1]

            window_counts[s[left]] -= 1
            if s[left] in t_count and window_counts[s[left]] <
               t_count[s[left]]:
                formed -= 1
            left += 1
```

```
        right += 1
    return min_window_substr

# Example usage
print(min_window("ADOBECODEBANC", "ABC"))  # Output: "BANC"
```

15 Longest Increasing Subsequence in String

15.1 Problem Statement

Given a string, find the length of the longest increasing subsequence of characters.

15.2 Dry Run on Test Cases

- **Test Case 1**: Input = "aebbcg" → Output: 3 ("abc")
- **Test Case 2**: Input = "abcde" → Output: 5
- **Test Case 3**: Input = "a" → Output: 1
- **Test Case 4**: Input = "" → Output: 0

15.3 Algorithm (Memoization)

1. Use recursive function with memoization.
2. For each index, consider including character if greater than previous.
3. Return max length.

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

15.4 Python Solution (Memoization)

```
def longest_increasing_subsequence(s):
    memo = {}

    def lis(index, prev_char):
        if index == len(s):
            return 0
        if (index, prev_char) in memo:
            return memo[(index, prev_char)]

        not_take = lis(index + 1, prev_char)
        take = 0
        if prev_char < s[index]:
            take = 1 + lis(index + 1, s[index])

        memo[(index, prev_char)] = max(take, not_take)
        return memo[(index, prev_char)]
```

```
    return lis(0, chr(0))

# Example usage
print(longest_increasing_subsequence("aebbcg"))  # Output: 3
```

15.5 Python Solution (Tabulation)

```
def longest_increasing_subsequence_tab(s):
    if not s:
        return 0

    n = len(s)
    dp = [1] * n

    for i in range(1, n):
        for j in range(i):
            if s[j] < s[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

# Example usage
print(longest_increasing_subsequence_tab("aebbcg"))  # Output: 3
```

16 Check for Valid Shuffle of Two Strings

16.1 Problem Statement

Given strings s1, s2, and result, check if result is a valid shuffle of s1 and s2.

16.2 Dry Run on Test Cases

- **Test Case 1**: s1 = "abc", s2 = "def", result = "adbcef" → Output: True
- **Test Case 2**: s1 = "abc", s2 = "def", result = "abcdefg" → Output: False
- **Test Case 3**: s1 = "", s2 = "", result = "" → Output: True
- **Test Case 4**: s1 = "a", s2 = "b", result = "ba" → Output: True

16.3 Algorithm

1. Check if len(result) = len(s1) + len(s2).
2. Use two pointers for s1 and s2, one for result.
3. Match characters; if no match, return False.

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

16.4 Python Solution

```python
def is_valid_shuffle(s1, s2, result):
    if len(result) != len(s1) + len(s2):
        return False

    i = j = k = 0
    while k < len(result):
        if i < len(s1) and s1[i] == result[k]:
            i += 1
        elif j < len(s2) and s2[j] == result[k]:
            j += 1
        else:
            return False
        k += 1
    return i == len(s1) and j == len(s2)

# Example usage
print(is_valid_shuffle("abc", "def", "adbcef"))  # Output: True
```

17 Remove Duplicate Letters

17.1 Problem Statement

Given a string, remove duplicate letters so each letter appears once, in smallest lexicographical order.

17.2 Dry Run on Test Cases

- **Test Case 1**: Input = "bcabc" → Output: "abc"
- **Test Case 2**: Input = "cbacdcbc" → Output: "acdb"
- **Test Case 3**: Input = "a" → Output: "a"
- **Test Case 4**: Input = "" → Output: ""

17.3 Algorithm

1. Track last occurrence of each character.
2. Use stack to build result, ensuring lexicographical order.
3. Pop from stack if current char is smaller and later occurrences exist.

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

17.4 Python Solution

```python
def remove_duplicate_letters(s):
    last_occurrence = {}
```

```
    for i, char in enumerate(s):
        last_occurrence[char] = i

    stack = []
    seen = set()

    for i, char in enumerate(s):
        if char not in seen:
            while stack and char < stack[-1] and i <
               last_occurrence[stack[-1]]:
                seen.remove(stack.pop())
            stack.append(char)
            seen.add(char)
    return ''.join(stack)

# Example usage
print(remove_duplicate_letters("cbacdcbc"))  # Output: "acdb"
```

18 Find All Palindromic Substrings

18.1 Problem Statement

Given a string, find the count of all palindromic substrings.

18.2 Dry Run on Test Cases

- **Test Case 1**: Input = "aaa" → Output: 6 ("a", "a", "a", "aa", "aa", "aaa")
- **Test Case 2**: Input = "abc" → Output: 3 ("a", "b", "c")
- **Test Case 3**: Input = "" → Output: 0
- **Test Case 4**: Input = "aba" → Output: 4 ("a", "b", "a", "aba")

18.3 Algorithm

1. For each index, expand around center for odd and even palindromes.
2. Count all valid palindromes.

Time Complexity: $O(n^2)$ **Space Complexity**: $O(1)$

18.4 Python Solution

```
def count_palindromic_substrings(s):
    def expand_around_center(left, right):
        count = 0
        while left >= 0 and right < len(s) and s[left] == s[right
           ]:
            count += 1
```

```
            left -= 1
            right += 1
        return count

    total = 0
    for i in range(len(s)):
        total += expand_around_center(i, i)  # Odd length
        total += expand_around_center(i, i + 1)  # Even length
    return total

# Example usage
print(count_palindromic_substrings("aaa"))  # Output: 6
```

19 Rabin-Karp String Matching

19.1 Problem Statement

Given a text and pattern, find all occurrences of pattern in text using Rabin-Karp.

19.2 Dry Run on Test Cases

- **Test Case 1**: text = "AABAACAADA", pattern = "AA" → Output: [0, 3, 6]
- **Test Case 2**: text = "abcd", pattern = "xyz" → Output: []
- **Test Case 3**: text = "", pattern = "a" → Output: []
- **Test Case 4**: text = "aaa", pattern = "aaa" → Output: [0]

19.3 Algorithm

1. Compute hash of pattern and first window of text.
2. Slide window, update hash, compare if equal.
3. Verify matches to avoid hash collisions.

Time Complexity: $O(n+m)$ average **Space Complexity**: $O(1)$

19.4 Python Solution

```
def rabin_karp(text, pattern):
    if not pattern or not text:
        return []

    d = 256  # Number of characters
    q = 101  # Prime number
    m, n = len(pattern), len(text)
    result = []

```

```
    h = pow(d, m-1) % q
    p = t = 0

    for i in range(m):
        p = (d * p + ord(pattern[i])) % q
        t = (d * t + ord(text[i])) % q

    for i in range(n - m + 1):
        if p == t:
            if text[i:i+m] == pattern:
                result.append(i)
        if i < n - m:
            t = (d * (t - ord(text[i]) * h) + ord(text[i + m])) %
                q
            if t < 0:
                t += q
    return result

# Example usage
print(rabin_karp("AABAACAADA", "AA"))  # Output: [0, 3, 6]
```

20 KMP Algorithm for Pattern Searching

20.1 Problem Statement

Given a text and pattern, find all occurrences of pattern in text using KMP algorithm.

20.2 Dry Run on Test Cases

- **Test Case 1**: text = "AABAACAADA", pattern = "AA" → Output: [0, 3, 6]
- **Test Case 2**: text = "abcd", pattern = "xyz" → Output: []
- **Test Case 3**: text = "", pattern = "a" → Output: []
- **Test Case 4**: text = "aaa", pattern = "aaa" → Output: [0]

20.3 Algorithm

1. Compute LPS (longest prefix suffix) array for pattern.
2. Use LPS to skip redundant comparisons while matching.
3. Collect all match indices.

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

20.4 Python Solution

```python
def kmp_search(text, pattern):
    def compute_lps(pattern):
        m = len(pattern)
        lps = [0] * m
        length = 0
        i = 1
        while i < m:
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    length = lps[length - 1]
                else:
                    lps[i] = 0
                    i += 1
        return lps

    result = []
    if not pattern or not text:
        return result

    m, n = len(pattern), len(text)
    lps = compute_lps(pattern)
    i = j = 0

    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == m:
            result.append(i - j)
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return result

# Example usage
print(kmp_search("AABAACAADA", "AA"))  # Output: [0, 3, 6]
```

21 Reverse a Linked List

21.1 Problem Statement

Given a singly linked list, reverse it and return the new head.

21.2 Dry Run on Test Cases

- **Test Case 1**: Input = 1->2->3->4->5 → Output: 5->4->3->2->1
- **Test Case 2**: Input = 1 → Output: 1
- **Test Case 3**: Input = None → Output: None
- **Test Case 4**: Input = 1->2 → Output: 2->1

21.3 Algorithm

1. Initialize prev = None, curr = head.
2. While curr, save next, set curr.next = prev, move prev and curr.
3. Return prev as new head.

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

21.4 Python Solution

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

def reverse_list(head):
    prev = None
    curr = head

    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node
    return prev

# Example usage (simplified)
# head = ListNode(1, ListNode(2, ListNode(3)))
# reversed_head = reverse_list(head)
```

22 Detect Cycle in a Linked List

22.1 Problem Statement

Given a linked list, determine if it has a cycle.

22.2 Dry Run on Test Cases

- **Test Case 1**: 1->2->3->4->2(cycle) → Output: True

- **Test Case 2**: 1->2->3 → Output: False
- **Test Case 3**: None → Output: False
- **Test Case 4**: 1 → Output: False

22.3 Algorithm

1. Use two pointers: slow (1 step), fast (2 steps).
2. If they meet, cycle exists.
3. If fast reaches end, no cycle.

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

22.4 Python Solution

```python
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False

# Example usage
# head = ListNode(1, ListNode(2, ListNode(3)))
# head.next.next.next = head.next  # Creates cycle
# print(has_cycle(head))  # Output: True
```

23 Merge Two Sorted Linked Lists

23.1 Problem Statement

Given two sorted linked lists, merge them into one sorted list.

23.2 Dry Run on Test Cases

- **Test Case 1**: l1 = 1->2->4, l2 = 1->3->4 → Output: 1->1->2->3->4->4
- **Test Case 2**: l1 = None, l2 = None → Output: None
- **Test Case 3**: l1 = 1, l2 = None → Output: 1
- **Test Case 4**: l1 = 2, l2 = 1 → Output: 1->2

23.3 Algorithm

1. Use dummy node to simplify merging.
2. Compare heads of l1 and l2, append smaller to result.
3. Move to next node of chosen list.
4. Append remaining nodes.

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

23.4 Python Solution

```python
def merge_two_lists(l1, l2):
    dummy = ListNode(0)
    curr = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            curr.next = l1
            l1 = l1.next
        else:
            curr.next = l2
            l2 = l2.next
        curr = curr.next

    curr.next = l1 if l1 else l2
    return dummy.next

# Example usage
# l1 = ListNode(1, ListNode(2, ListNode(4)))
# l2 = ListNode(1, ListNode(3, ListNode(4)))
# merged = merge_two_lists(l1, l2)
```

24 Remove Nth Node from End

24.1 Problem Statement

Given a linked list and integer n, remove the nth node from the end.

24.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5, n = 2 → Output: 1->2->3->5
- **Test Case 2**: head = 1, n = 1 → Output: None
- **Test Case 3**: head = 1->2, n = 2 → Output: 2
- **Test Case 4**: head = None, n = 1 → Output: None

24.3 Algorithm

1. Use two pointers: fast moves n steps ahead.
2. Move slow and fast until fast reaches end.
3. Slow points to node before nth from end; remove it.

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

24.4 Python Solution

```python
def remove_nth_from_end(head, n):
    dummy = ListNode(0, head)
    slow = fast = dummy

    for _ in range(n):
        fast = fast.next

    while fast.next:
        slow = slow.next
        fast = fast.next

    slow.next = slow.next.next
    return dummy.next

# Example usage
# head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
# new_head = remove_nth_from_end(head, 2)
```

25 Find Middle of Linked List

25.1 Problem Statement

Given a linked list, return the middle node (if even, second middle node).

25.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5 → Output: 3
- **Test Case 2**: head = 1->2->3->4 → Output: 3
- **Test Case 3**: head = 1 → Output: 1
- **Test Case 4**: head = None → Output: None

25.3 Algorithm

1. Use two pointers: slow (1 step), fast (2 steps).

2. When fast reaches end, slow is at middle.

3. Return slow.

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

25.4 Python Solution

```python
def middle_node(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow

# Example usage
# head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
# middle = middle_node(head)
```

Solutions to DSA Questions 51-80 (Linked Lists, Stacks, Queues, Trees)

For 1-2 Years Experience Roles at EPAM Compiled on September 26, 2025

Introduction

This document provides detailed solutions for 30 Data Structures and Algorithms (DSA) problems (questions 51 to 80) from the Linked Lists, Stacks, Queues, and Trees 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. Dynamic programming problems include both memoization and tabulation approaches where applicable.

Contents

1 Add Two Numbers as Linked Lists

1.1 Problem Statement

Given two non-empty linked lists representing non-negative integers (digits in reverse order), add them and return the sum as a linked list.

1.2 Dry Run on Test Cases

- **Test Case 1**: l1 = 2->4->3, l2 = 5->6->4 → Output: 7->0->8 (342 + 465 = 807)
- **Test Case 2**: l1 = 0, l2 = 0 → Output: 0
- **Test Case 3**: l1 = 9->9->9, l2 = 1 → Output: 0->0->0->1
- **Test Case 4**: l1 = 1->8, l2 = 0 → Output: 1->8

1.3 Algorithm

1. Initialize dummy node and current pointer.
2. Traverse both lists, adding digits and carry.
3. Create new nodes for sum digits, handle carry.
4. Return dummy.next.

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

1.4 Python Solution

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

def add_two_numbers(l1, l2):
    dummy = ListNode(0)
    curr = dummy
    carry = 0

    while l1 or l2 or carry:
        x = l1.val if l1 else 0
        y = l2.val if l2 else 0
        total = x + y + carry
        carry = total // 10
        curr.next = ListNode(total % 10)
        curr = curr.next
        l1 = l1.next if l1 else None
        l2 = l2.next if l2 else None
```

```
    return dummy.next
```

2 Intersection of Two Linked Lists

2.1 Problem Statement

Given two linked lists, find the node where they intersect (same reference) or return None.

2.2 Dry Run on Test Cases

- **Test Case 1**: l1 = 4->1->(8->4->5), l2 = 5->6->1->(8->4->5) → Output: Node 8
- **Test Case 2**: l1 = 1->2, l2 = 3->4 → Output: None
- **Test Case 3**: l1 = None, l2 = 1 → Output: None
- **Test Case 4**: l1 = 1->2->3, l2 = 3 → Output: Node 3

2.3 Algorithm

1. Traverse both lists, switching to other list when reaching end.
2. If pointers meet, that's the intersection.
3. If both reach None, no intersection.

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

2.4 Python Solution

```
def get_intersection_node(headA, headB):
    if not headA or not headB:
        return None

    a, b = headA, headB
    while a != b:
        a = a.next if a else headB
        b = b.next if b else headA
    return a
```

3 Reverse Nodes in k-Group

3.1 Problem Statement

Given a linked list, reverse every k nodes and return the new head.

3.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5, k = 2 → Output: 2->1->4->3->5
- **Test Case 2**: head = 1->2->3->4->5, k = 3 → Output: 3->2->1->4->5
- **Test Case 3**: head = 1, k = 1 → Output: 1
- **Test Case 4**: head = None, k = 2 → Output: None

3.3 Algorithm

1. Check if k nodes exist.
2. Reverse k nodes using iterative reversal.
3. Recursively process next k-group.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$ or $O(n/k)$ for recursion

3.4 Python Solution

```python
def reverse_k_group(head, k):
    def get_kth(curr, k):
        while curr and k > 0:
            curr = curr.next
            k -= 1
        return curr

    dummy = ListNode(0, head)
    prev_group = dummy

    while head:
        kth = get_kth(head, k - 1)
        if not kth:
            break
        next_group = kth.next
        kth.next = None

        # Reverse current group
        prev = next_group
        curr = head
        while curr:
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node

        # Connect to previous group
        prev_group.next = prev
        prev_group = head
        head = next_group
```

```
31
32      return dummy.next
```

4 Palindrome Linked List

4.1 Problem Statement

Given a linked list, determine if it is a palindrome.

4.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->2->1 → Output: True
- **Test Case 2**: head = 1->2 → Output: False
- **Test Case 3**: head = 1 → Output: True
- **Test Case 4**: head = None → Output: True

4.3 Algorithm

1. Find middle using slow and fast pointers.
2. Reverse second half.
3. Compare first and second halves.

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

4.4 Python Solution

```
def is_palindrome(head):
    if not head or not head.next:
        return True

    # Find middle
    slow = fast = head
    while fast.next and fast.next.next:
        slow = slow.next
        fast = fast.next.next

    # Reverse second half
    second_half = slow.next
    slow.next = None
    prev = None
    while second_half:
        next_node = second_half.next
        second_half.next = prev
        prev = second_half
        second_half = next_node
```

```
    # Compare
    first_half = head
    while prev:
        if first_half.val != prev.val:
            return False
        first_half = first_half.next
        prev = prev.next
    return True
```

5 Remove Linked List Elements

5.1 Problem Statement

Given a linked list and a value, remove all nodes with that value.

5.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->6->3->4->6, val = 6 → Output: 1->2->3->4
- **Test Case 2**: head = None, val = 1 → Output: None
- **Test Case 3**: head = 7->7->7, val = 7 → Output: None
- **Test Case 4**: head = 1, val = 2 → Output: 1

5.3 Algorithm

1. Use dummy node to handle head removal.
2. Traverse list, skip nodes with given value.
3. Return dummy.next.

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

5.4 Python Solution

```
def remove_elements(head, val):
    dummy = ListNode(0, head)
    curr = dummy

    while curr.next:
        if curr.next.val == val:
            curr.next = curr.next.next
        else:
            curr = curr.next
    return dummy.next
```

6 Swap Nodes in Pairs

6.1 Problem Statement

Given a linked list, swap every two adjacent nodes and return the head.

6.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4 → Output: 2->1->4->3
- **Test Case 2**: head = 1 → Output: 1
- **Test Case 3**: head = None → Output: None
- **Test Case 4**: head = 1->2->3 → Output: 2->1->3

6.3 Algorithm

1. If less than 2 nodes, return head.
2. Swap current pair, recursively swap rest.
3. Adjust pointers to connect swapped pairs.

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

6.4 Python Solution

```
def swap_pairs(head):
    if not head or not head.next:
        return head

    next_node = head.next
    head.next = swap_pairs(next_node.next)
    next_node.next = head
    return next_node
```

7 Odd Even Linked List

7.1 Problem Statement

Given a linked list, group odd-indexed nodes together followed by even-indexed nodes.

7.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5 → Output: 1->3->5->2->4
- **Test Case 2**: head = 2->1->3->5->6->4->7 → Output: 2->3->6->7->1->5->4
- **Test Case 3**: head = 1 → Output: 1

- **Test Case 4**: head = None → Output: None

7.3 Algorithm

1. Maintain odd and even pointers.
2. Link odd nodes together, even nodes together.
3. Connect odd list to even list.

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

7.4 Python Solution

```python
def odd_even_list(head):
    if not head or not head.next:
        return head

    odd = head
    even = head.next
    even_head = even

    while even and even.next:
        odd.next = even.next
        odd = odd.next
        even.next = odd.next
        even = even.next

    odd.next = even_head
    return head
```

8 Partition List

8.1 Problem Statement

Given a linked list and value x, partition list so all nodes less than x come before nodes greater than or equal to x.

8.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->4->3->2->5->2, x = 3 → Output: 1->2->2->4->3->5
- **Test Case 2**: head = 2->1, x = 2 → Output: 1->2
- **Test Case 3**: head = None, x = 0 → Output: None
- **Test Case 4**: head = 1, x = 2 → Output: 1

8.3 Algorithm

1. Maintain two lists: less and greater.
2. Traverse list, append nodes to appropriate list.
3. Connect less to greater.

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

8.4 Python Solution

```python
def partition(head, x):
    less_dummy = ListNode(0)
    greater_dummy = ListNode(0)
    less = less_dummy
    greater = greater_dummy

    curr = head
    while curr:
        if curr.val < x:
            less.next = curr
            less = less.next
        else:
            greater.next = curr
            greater = greater.next
        curr = curr.next

    greater.next = None
    less.next = greater_dummy.next
    return less_dummy.next
```

9 Rotate List

9.1 Problem Statement

Given a linked list, rotate it to the right by k places.

9.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5, k = 2 → Output: 4->5->1->2->3
- **Test Case 2**: head = 0->1->2, k = 4 → Output: 2->0->1
- **Test Case 3**: head = 1, k = 1 → Output: 1
- **Test Case 4**: head = None, k = 1 → Output: None

9.3 Algorithm

1. Find length and last node.

2. Compute effective k = k % length.

3. Find new tail (length - k - 1), set next to None, connect last to head.

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

9.4 Python Solution

```python
def rotate_right(head, k):
    if not head or not head.next or k == 0:
        return head

    # Find length and last node
    length = 1
    last = head
    while last.next:
        last = last.next
        length += 1

    k = k % length
    if k == 0:
        return head

    # Find new tail
    new_tail = head
    for _ in range(length - k - 1):
        new_tail = new_tail.next

    new_head = new_tail.next
    new_tail.next = None
    last.next = head
    return new_head
```

10 Reorder List

10.1 Problem Statement

Given a linked list L0->L1->...->Ln-1->Ln, reorder it to L0->Ln->L1->Ln-1->...

10.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4 → Output: 1->4->2->3
- **Test Case 2**: head = 1->2->3->4->5 → Output: 1->5->2->4->3
- **Test Case 3**: head = 1 → Output: 1
- **Test Case 4**: head = None → Output: None

10.3 Algorithm

1. Find middle using slow and fast pointers.
2. Reverse second half.
3. Merge first and second halves alternately.

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

10.4 Python Solution

```python
def reorder_list(head):
    if not head or not head.next:
        return

    # Find middle
    slow = fast = head
    while fast.next and fast.next.next:
        slow = slow.next
        fast = fast.next.next

    # Reverse second half
    second = slow.next
    slow.next = None
    prev = None
    while second:
        next_node = second.next
        second.next = prev
        prev = second
        second = next_node

    # Merge
    first = head
    while prev:
        next_first = first.next
        next_prev = prev.next
        first.next = prev
        prev.next = next_first
        first = next_first
        prev = next_prev
```

11 Valid Number

11.1 Problem Statement

Given a string, determine if it is a valid number (integer, decimal, or scientific notation).

11.2 Dry Run on Test Cases

- **Test Case 1**: Input = "0" → Output: True

- **Test Case 2**: Input = "e" → Output: False
- **Test Case 3**: Input = "2e10" → Output: True
- **Test Case 4**: Input = "-0.1" → Output: True

11.3 Algorithm

1. Use regex or manual parsing to check:
    - Optional sign, digits, optional decimal, optional 'e' followed by integer.
2. Ensure proper format (e.g., no multiple decimals).

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

11.4 Python Solution

```python
def is_number(s):
    s = s.strip()
    if not s:
        return False

    # Split on 'e' or 'E'
    parts = s.lower().split('e')
    if len(parts) > 2:
        return False

    # Validate base part
    base = parts[0]
    if not base or base == '+' or base == '-':
        return False

    decimal_count = 0
    digit_seen = False
    for i, char in enumerate(base):
        if char == '.':
            decimal_count += 1
            if decimal_count > 1:
                return False
        elif char.isdigit():
            digit_seen = True
        elif char not in ['+', '-'] or i != 0:
            return False

    if not digit_seen:
        return False

    # Validate exponent if present
    if len(parts) == 2:
        exponent = parts[1]
        if not exponent or exponent == '+' or exponent == '-':
```

```
            return False
        digit_seen = False
        for i, char in enumerate(exponent):
            if char.isdigit():
                digit_seen = True
            elif char not in ['+', '-'] or i != 0:
                return False
        if not digit_seen:
            return False

    return True

# Example usage
print(is_number("2e10"))  # Output: True
```

12 Min Stack

12.1 Problem Statement

Design a stack that supports push, pop, top, and retrieving the minimum element in O(1) time.

12.2 Dry Run on Test Cases

- **Test Case 1**: push(3), push(5), getMin() → 3, push(2), getMin() → 2, pop(), getMin() → 3
- **Test Case 2**: push(1), pop(), top() → None
- **Test Case 3**: push(2), push(1), getMin() → 1
- **Test Case 4**: empty stack, getMin() → None

12.3 Algorithm

1. Use two stacks: one for values, one for minimums.
2. Push: append value, update min stack if needed.
3. Pop: remove from both stacks if popped value was min.
4. Top/GetMin: return top of respective stacks.

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

12.4 Python Solution

```
class MinStack:
    def __init__(self):
        self.stack = []
```

```
        self.min_stack = []

    def push(self, val):
        self.stack.append(val)
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self):
        if not self.stack:
            return
        val = self.stack.pop()
        if val == self.min_stack[-1]:
            self.min_stack.pop()

    def top(self):
        return self.stack[-1] if self.stack else None

    def getMin(self):
        return self.min_stack[-1] if self.min_stack else None

# Example usage
# minStack = MinStack()
# minStack.push(3)
# minStack.push(5)
# print(minStack.getMin())  # Output: 3
```

13 Evaluate Reverse Polish Notation

13.1 Problem Statement

Given an array of strings representing an RPN expression, evaluate it.

13.2 Dry Run on Test Cases

- **Test Case 1**: tokens = ["2", "1", "+", "3", "*"] → Output: 9 ((2 + 1) * 3)
- **Test Case 2**: tokens = ["4", "13", "5", "/", "+"] → Output: 6 (4 + 13/5)
- **Test Case 3**: tokens = ["10"] → Output: 10
- **Test Case 4**: tokens = ["10", "6", "/"] → Output: 1

13.3 Algorithm

1. Use a stack to store operands.
2. For each token:
    - If number, push to stack.

- If operator, pop two operands, compute, push result.

3. Return stack top.

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

13.4 Python Solution

```python
def eval_rpn(tokens):
    stack = []
    operators = {
        '+': lambda x, y: x + y,
        '-': lambda x, y: x - y,
        '*': lambda x, y: x * y,
        '/': lambda x, y: int(x / y)
    }

    for token in tokens:
        if token in operators:
            b = stack.pop()
            a = stack.pop()
            stack.append(operators[token](a, b))
        else:
            stack.append(int(token))
    return stack[0]

# Example usage
print(eval_rpn(["2", "1", "+", "3", "*"]))  # Output: 9
```

14 Valid Parentheses with Wildcard

14.1 Problem Statement

Given a string with '(', ')', and '*', where '*' can be '(', ')', or empty, check if valid.

14.2 Dry Run on Test Cases

- **Test Case 1**: s = "()" → Output: True
- **Test Case 2**: s = "(*)" → Output: True
- **Test Case 3**: s = "(*))" → Output: True
- **Test Case 4**: s = "((*)" → Output: True

14.3 Algorithm

1. Track min and max open brackets (min can be 0, max can increase with *).
2. For each char:

- '(': min++, max++
- ')': min–, max–
- '*': min–, max++

3. Ensure min >= 0, reset min to 0 if negative.
4. Return True if min == 0 at end.

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

14.4 Python Solution

```python
def check_valid_string(s):
    min_open = max_open = 0

    for char in s:
        if char == '(':
            min_open += 1
            max_open += 1
        elif char == ')':
            min_open -= 1
            max_open -= 1
        else:  # '*'
            min_open -= 1
            max_open += 1
        if max_open < 0:
            return False
        if min_open < 0:
            min_open = 0
    return min_open == 0

# Example usage
print(check_valid_string("(*)"))  # Output: True
```

15 Next Greater Element

15.1 Problem Statement

Given two arrays nums1 and nums2, for each element in nums1, find the next greater element in nums2.

15.2 Dry Run on Test Cases

- **Test Case 1**: nums1 = [4,1,2], nums2 = [1,3,4,2] → Output: [-1,3,-1]
- **Test Case 2**: nums1 = [2,4], nums2 = [1,2,3,4] → Output: [3,-1]
- **Test Case 3**: nums1 = [1], nums2 = [1] → Output: [-1]

- **Test Case 4**: nums1 = [], nums2 = [1,2] → Output: []

15.3 Algorithm

1. Use stack to find next greater for each element in nums2.
2. Store results in hashmap.
3. Map nums1 elements to their next greater.

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

15.4 Python Solution

```
def next_greater_element(nums1, nums2):
    stack = []
    next_greater = {}

    for num in nums2:
        while stack and stack[-1] < num:
            next_greater[stack.pop()] = num
        stack.append(num)

    while stack:
        next_greater[stack.pop()] = -1

    return [next_greater[num] for num in nums1]

# Example usage
print(next_greater_element([4,1,2], [1,3,4,2]))  # Output:
    [-1,3,-1]
```

16 Daily Temperatures

16.1 Problem Statement

Given an array of temperatures, return an array where each element is the number of days until a warmer day.

16.2 Dry Run on Test Cases

- **Test Case 1**: temperatures = [73,74,75,71,69,72,76,73] → Output: [1,1,4,2,1,1,0,0]
- **Test Case 2**: temperatures = [30,40,50,60] → Output: [1,1,1,0]
- **Test Case 3**: temperatures = [30] → Output: [0]
- **Test Case 4**: temperatures = [30,20,10] → Output: [0,0,0]

16.3 Algorithm

1. Use stack to store indices of temperatures.
2. For each temperature, pop stack while current > stack top, calculate days.
3. Push current index to stack.

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

16.4 Python Solution

```python
def daily_temperatures(temperatures):
    n = len(temperatures)
    result = [0] * n
    stack = []

    for i in range(n):
        while stack and temperatures[i] > temperatures[stack
            [-1]]:
            prev = stack.pop()
            result[prev] = i - prev
        stack.append(i)
    return result

# Example usage
print(daily_temperatures([73,74,75,71,69,72,76,73]))  # Output:
    [1,1,4,2,1,1,0,0]
```

17 Implement Stack Using Queues

17.1 Problem Statement

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

17.2 Dry Run on Test Cases

- **Test Case 1**: push(1), push(2), top() → 2, pop() → 2, empty() → False
- **Test Case 2**: push(1), pop(), empty() → True
- **Test Case 3**: empty() → True
- **Test Case 4**: push(1), top() → 1

17.3 Algorithm

1. Use two queues; main queue holds stack elements.
2. Push: add to main queue.

3. Pop/Top: move all but last element to second queue, process last, swap queues.

Time Complexity: $O(n)$ for pop/top, $O(1)$ for push/empty **Space Complexity**: $O(n)$

17.4 Python Solution

```python
from collections import deque

class MyStack:
    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x):
        self.q1.append(x)

    def pop(self):
        if not self.q1:
            return None
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())
        result = self.q1.popleft()
        self.q1, self.q2 = self.q2, self.q1
        return result

    def top(self):
        if not self.q1:
            return None
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())
        result = self.q1[0]
        self.q2.append(self.q1.popleft())
        self.q1, self.q2 = self.q2, self.q1
        return result

    def empty(self):
        return len(self.q1) == 0
```

18 Implement Queue Using Stacks

18.1 Problem Statement

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

18.2 Dry Run on Test Cases

- **Test Case 1**: enqueue(1), enqueue(2), peek() $\rightarrow$ 1, dequeue() $\rightarrow$ 1
- **Test Case 2**: enqueue(1), dequeue(), empty() $\rightarrow$ True

- **Test Case 3**: empty() → True
- **Test Case 4**: enqueue(1), peek() → 1

18.3 Algorithm

1. Use two stacks: $\text{push}_stack and pop_stack. Enqueue : push to push_stack.$

2. Dequeue/Peek: move $\text{push}_stack to pop_stack if empty, pop/peek from pop_stack.$ **Time Complexity**: $O(1)$ amortized for all operations **Space Complexity**: $O(n)$

18.4 Python Solution

```
class MyQueue:
    def __init__(self):
        self.push_stack = []
        self.pop_stack = []

    def push(self, x):
        self.push_stack.append(x)

    def pop(self):
        if not self.pop_stack:
            while self.push_stack:
                self.pop_stack.append(self.push_stack.pop())
        return self.pop_stack.pop() if self.pop_stack else
           None

    def peek(self):
        if not self.pop_stack:
            while self.push_stack:
                self.pop_stack.append(self.push_stack.pop())
        return self.pop_stack[-1] if self.pop_stack else None

    def empty(self):
        return not self.push_stack and not self.pop_stack
```

19 Design Circular Queue

19.1 Problem Statement

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

19.2 Dry Run on Test Cases

- **Test Case 1**: k=3, enQueue(1), enQueue(2), enQueue(3), isFull() → True, deQueue() → 1
- **Test Case 2**: k=1, enQueue(1), deQueue(), isEmpty() → True

– **Test Case 3**: k=2, enQueue(1), Front() → 1, Rear() → 1

– **Test Case 4**: k=1, isEmpty() → True

19.3 Algorithm

1. Use array of size k with front and rear pointers.
2. Enqueue: if not full, add at rear, increment rear % k.
3. Dequeue: if not empty, increment front % k.
4. Track size for empty/full checks.

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

19.4 Python Solution

```
class MyCircularQueue:
    def __init__(self, k):
        self.size = k
        self.queue = [None] * k
        self.front = -1  # Index of front element
        self.rear = -1   # Index of last element
        self.count = 0   # Number of elements

    def enQueue(self, value):
        if self.isFull():
            return False
        if self.isEmpty():
            self.front = 0
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = value
        self.count += 1
        return True

    def deQueue(self):
        if self.isEmpty():
            return False
        self.front = (self.front + 1) % self.size
        self.count -= 1
        if self.isEmpty():
            self.front = -1
            self.rear = -1
        return True

    def Front(self):
        return self.queue[self.front] if not self.isEmpty
           () else -1

    def Rear(self):
```

```
        return self.queue[self.rear] if not self.isEmpty
            () else -1

    def isEmpty(self):
        return self.count == 0

    def isFull(self):
        return self.count == self.size
```

20 Sliding Window Maximum

20.1 Problem Statement

Given an array and window size k, find the maximum element in each sliding window.

20.2 Dry Run on Test Cases

* **Test Case 1**: nums = [1,3,-1,-3,5,3,6,7], k = 3 → Output: [3,3,5,5,6,7]
* **Test Case 2**: nums = [1], k = 1 → Output: [1]
* **Test Case 3**: nums = [1,-1], k = 1 → Output: [1,-1]
* **Test Case 4**: nums = [], k = 1 → Output: []

20.3 Algorithm

1. Use deque to store indices of potential max elements.
2. For each element:
   - Remove indices outside window.
   - Remove smaller elements from back.
   - Add current index.
3. After k elements, append max (front of deque) for each window.

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

20.4 Python Solution

```
from collections import deque

def max_sliding_window(nums, k):
    if not nums or k == 0:
        return []

```

```
    result = []
    deq = deque()

    for i in range(len(nums)):
        # Remove indices outside window
        while deq and deq[0] <= i - k:
            deq.popleft()
        # Remove smaller elements
        while deq and nums[deq[-1]] < nums[i]:
            deq.pop()
        deq.append(i)
        # Add max for window
        if i >= k - 1:
            result.append(nums[deq[0]])
    return result

# Example usage
print(max_sliding_window([1,3,-1,-3,5,3,6,7], 3))  # Output: [3,3,5,5,6,7]
```

21 Largest Rectangle in Histogram

21.1 Problem Statement

Given an array of bar heights, find the largest rectangle area in the histogram.

21.2 Dry Run on Test Cases

- **Test Case 1**: heights = [2,1,5,6,2,3] → Output: 10 (height 5, width 2)
- **Test Case 2**: heights = [2,4] → Output: 4
- **Test Case 3**: heights = [1] → Output: 1
- **Test Case 4**: heights = [] → Output: 0

21.3 Algorithm

1. Use stack to store indices of increasing heights.
2. For each bar, pop stack while current height $<$ stack top height.
3. Calculate area for each popped bar: height * (current index - previous index - 1).
4. Handle remaining bars after loop.

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

21.4 Python Solution

```python
def largest_rectangle_area(heights):
    stack = [-1]
    max_area = 0
    heights.append(0)  # Sentinel to process remaining bars

    for i in range(len(heights)):
        while stack[-1] != -1 and heights[i] < heights[stack[-1]]:
            h = heights[stack.pop()]
            w = i - stack[-1] - 1
            max_area = max(max_area, h * w)
        stack.append(i)

    heights.pop()  # Remove sentinel
    return max_area

# Example usage
print(largest_rectangle_area([2,1,5,6,2,3]))  # Output: 10
```

22 Minimum Remove to Make Valid Parentheses

22.1 Problem Statement

Given a string with '(', ')', and letters, remove minimum characters to make it valid.

22.2 Dry Run on Test Cases

- **Test Case 1**: s = "lee(t(c)o)de)" → Output: "lee(t(c)o)de"
- **Test Case 2**: s = "a)b(c)d" → Output: "ab(c)d"
- **Test Case 3**: s = "))((" → Output: ""
- **Test Case 4**: s = "(a(b(c)d)" → Output: "a(b(c)d)"

22.3 Algorithm

1. Use stack to track indices of open parentheses.
2. Mark invalid parentheses for removal.
3. Build result string, skipping marked indices.

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

22.4 Python Solution

```python
def min_remove_to_make_valid(s):
    s = list(s)
    stack = []

    # Mark invalid parentheses
    for i, char in enumerate(s):
        if char == '(':
            stack.append(i)
        elif char == ')':
            if stack:
                stack.pop()
            else:
                s[i] = ''

    # Mark unmatched open parentheses
    while stack:
        s[stack.pop()] = ''

    return ''.join(s)

# Example usage
print(min_remove_to_make_valid("lee(t(c)o)de)"))  # Output: "lee(t(c)o)de"
```

23 Binary Tree Inorder Traversal

23.1 Problem Statement

Given a binary tree, return its inorder traversal (left, root, right).

23.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,null,2,3] → Output: [1,3,2]
- **Test Case 2**: root = [] → Output: []
- **Test Case 3**: root = [1] → Output: [1]
- **Test Case 4**: root = [1,2,3] → Output: [2,1,3]

23.3 Algorithm

1. Use iterative approach with stack.
2. Push all left nodes to stack.
3. Pop node, add to result, process right subtree.

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$ (h = tree height)

23.4 Python Solution

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

def inorder_traversal(root):
    result = []
    stack = []
    curr = root

    while curr or stack:
        while curr:
            stack.append(curr)
            curr = curr.left
        curr = stack.pop()
        result.append(curr.val)
        curr = curr.right
    return result
```

24 Binary Tree Preorder Traversal

24.1 Problem Statement

Given a binary tree, return its preorder traversal (root, left, right).

24.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,null,2,3] → Output: [1,2,3]
- **Test Case 2**: root = [] → Output: []
- **Test Case 3**: root = [1] → Output: [1]
- **Test Case 4**: root = [1,2,3] → Output: [1,2,3]

24.3 Algorithm

1. Use iterative approach with stack.
2. Push root, pop and process node, push right then left (stack reverses order).

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

24.4 Python Solution

```python
def preorder_traversal(root):
    if not root:
        return []

    result = []
    stack = [root]

    while stack:
        node = stack.pop()
        result.append(node.val)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
    return result
```

25 Binary Tree Postorder Traversal

25.1 Problem Statement

Given a binary tree, return its postorder traversal (left, right, root).

25.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,null,2,3] → Output: [3,2,1]
- **Test Case 2**: root = [] → Output: []
- **Test Case 3**: root = [1] → Output: [1]
- **Test Case 4**: root = [1,2,3] → Output: [2,3,1]

25.3 Algorithm

1. Use two stacks: first for preorder (root, left, right), second to reverse.
2. Pop from first stack, push to second, add children in reverse order.
3. Pop from second stack for result.

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

25.4 Python Solution

```python
def postorder_traversal(root):
    if not root:
        return []
```

```
    result = []
    stack1 = [root]
    stack2 = []

    while stack1:
        node = stack1.pop()
        stack2.append(node)
        if node.left:
            stack1.append(node.left)
        if node.right:
            stack1.append(node.right)

    while stack2:
        result.append(stack2.pop().val)
    return result
```

26 Binary Tree Level Order Traversal

26.1 Problem Statement

Given a binary tree, return its level order traversal (level by level, left to right).

26.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: [[3],[9,20],[15,7]]
- **Test Case 2**: root = [1] → Output: [[1]]
- **Test Case 3**: root = [] → Output: []
- **Test Case 4**: root = [1,2,3] → Output: [[1],[2,3]]

26.3 Algorithm

1. Use queue to process nodes level by level.
2. For each level, process all nodes, add children to queue.
3. Collect nodes per level in result.

Time Complexity: $O(n)$ **Space Complexity**: $O(w)$ (w = max width)

26.4 Python Solution

```
from collections import deque

def level_order(root):
```

```
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        current_level = []
        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(current_level)
    return result
```

27 Maximum Depth of Binary Tree

27.1 Problem Statement

Given a binary tree, find its maximum depth (number of nodes along longest path from root to leaf).

27.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: 3
- **Test Case 2**: root = [1,null,2] → Output: 2
- **Test Case 3**: root = [] → Output: 0
- **Test Case 4**: root = [1] → Output: 1

27.3 Algorithm

1. Use recursive DFS: max depth = 1 + max(left depth, right depth).
2. Base case: return 0 for None.

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$ for recursion

27.4 Python Solution (Recursive)

```
def max_depth(root):
    if not root:
        return 0
```

```
    return 1 + max(max_depth(root.left), max_depth(
        root.right))

# Example usage
# root = TreeNode(3, TreeNode(9), TreeNode(20,
    TreeNode(15), TreeNode(7)))
# print(max_depth(root))  # Output: 3
```

27.5 Python Solution (Iterative)

```
def max_depth_iterative(root):
    if not root:
        return 0

    queue = deque([(root, 1)])
    max_depth = 0

    while queue:
        node, depth = queue.popleft()
        max_depth = max(max_depth, depth)
        if node.left:
            queue.append((node.left, depth + 1))
        if node.right:
            queue.append((node.right, depth + 1))
    return max_depth
```

28 Same Tree

28.1 Problem Statement

Given two binary trees, check if they are structurally identical and have the same values.

28.2 Dry Run on Test Cases

- **Test Case 1**: p = [1,2,3], q = [1,2,3] → Output: True
- **Test Case 2**: p = [1,2], q = [1,null,2] → Output: False
- **Test Case 3**: p = [1,2,1], q = [1,1,2] → Output: False
- **Test Case 4**: p = [], q = [] → Output: True

28.3 Algorithm

1. Recursively check:
2. If both None, return True.

3. If one None or values differ, return False.

4. Recurse on left and right subtrees.

Time Complexity: $O(\min(n, m))$ **Space Complexity**: $O(h)$

28.4 Python Solution

```
def is_same_tree(p, q):
    if not p and not q:
        return True
    if not p or not q or p.val != q.val:
        return False
    return is_same_tree(p.left, q.left) and
       is_same_tree(p.right, q.right)
```

29 Symmetric Tree

29.1 Problem Statement

Given a binary tree, check if it is mirror symmetric.

29.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,2,3,4,4,3] → Output: True
- **Test Case 2**: root = [1,2,2,null,3,null,3] → Output: False
- **Test Case 3**: root = [1] → Output: True
- **Test Case 4**: root = [] → Output: True

29.3 Algorithm

1. Recursively check left and right subtrees:

2. If both None, return True.

3. If one None or values differ, return False.

4. Compare left.left with right.right and left.right with right.left.

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

29.4 Python Solution

```
def is_symmetric(root):
    def is_mirror(left, right):
        if not left and not right:
            return True
```

```
        if not left or not right or left.val !=
            right.val:
            return False
        return is_mirror(left.left, right.right)
            and is_mirror(left.right, right.left)

    return is_mirror(root, root) if root else True
```

30 Balanced Binary Tree

30.1 Problem Statement

Given a binary tree, determine if it is height-balanced (difference in heights of left and right subtrees <= 1).

30.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: True
- **Test Case 2**: root = [1,2,2,3,3,null,null,4,4] → Output: False
- **Test Case 3**: root = [] → Output: True
- **Test Case 4**: root = [1] → Output: True

30.3 Algorithm

1. Use DFS to compute height of each subtree.
2. Return -1 if unbalanced, else height.
3. Tree is balanced if root's height != -1.

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

30.4 Python Solution

```
def is_balanced(root):
    def check_height(node):
        if not node:
            return 0
        left_height = check_height(node.left)
        if left_height == -1:
            return -1
        right_height = check_height(node.right)
        if right_height == -1 or abs(left_height -
            right_height) > 1:
            return -1
        return max(left_height, right_height) + 1

```

```
13      return check_height(root) != -1
```

Solutions to DSA Questions 80-110 (Trees, BST, Heaps, Graphs) For 1-2 Years

Experience Roles at EPAM Compiled on September 26, 2025

Introduction

This document provides detailed solutions for 31 Data Structures and Algorithms (DSA) problems (questions 80 to 110) from the Trees, Binary Search Trees (BST), Heaps, and Graphs 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. Dynamic programming problems include both memoization and tabulation approaches where applicable.

Contents

1 Balanced Binary Tree

1.1 Problem Statement

Given a binary tree, determine if it is height-balanced (difference in heights of left and right subtrees <= 1).

1.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: True
- **Test Case 2**: root = [1,2,2,3,3,null,null,4,4] → Output: False
- **Test Case 3**: root = [] → Output: True
- **Test Case 4**: root = [1] → Output: True

1.3 Algorithm

1. Use DFS to compute height of each subtree.
2. Return -1 if unbalanced, else height.
3. Tree is balanced if root's height != -1.

Time Complexity: $O(n)$ **Space Complexity**: $O(h)$ (h = tree height)

1.4 Python Solution

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

def is_balanced(root):
    def check_height(node):
        if not node:
            return 0
        left_height = check_height(node.left)
        if left_height == -1:
            return -1
```

```
        right_height = check_height(node.right)
        if right_height == -1 or abs(left_height - right_height)
           > 1:
            return -1
        return max(left_height, right_height) + 1

    return check_height(root) != -1
```

2 Minimum Depth of Binary Tree

2.1 Problem Statement

Given a binary tree, find its minimum depth (shortest path from root to leaf).

2.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: 2
- **Test Case 2**: root = [2,null,3,null,4,null,5,null,6] → Output: 5
- **Test Case 3**: root = [] → Output: 0
- **Test Case 4**: root = [1] → Output: 1

2.3 Algorithm

1. Use BFS to find first leaf node.
2. Track depth while processing nodes level by level.
3. Return depth of first leaf.

Time Complexity: $O(n)$ **Space Complexity**: $O(w)$ (w = max width)

2.4 Python Solution

```
from collections import deque

def min_depth(root):
    if not root:
        return 0

    queue = deque([(root, 1)])
    while queue:
        node, depth = queue.popleft()
        if not node.left and not node.right:
            return depth
        if node.left:
            queue.append((node.left, depth + 1))
        if node.right:
```

```
            queue.append((node.right, depth + 1))
```

3 Path Sum

3.1 Problem Statement

Given a binary tree and target sum, determine if there is a root-to-leaf path summing to target.

3.2 Dry Run on Test Cases

- **Test Case 1**: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], target = 22 → Output: True
- **Test Case 2**: root = [1,2,3], target = 5 → Output: False
- **Test Case 3**: root = [], target = 0 → Output: False
- **Test Case 4**: root = [1], target = 1 → Output: True

3.3 Algorithm

1. Use DFS, subtract node value from target.
2. At leaf, check if remaining sum is 0.
3. Return True if any path satisfies.

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

3.4 Python Solution

```python
def has_path_sum(root, target):
    if not root:
        return False
    if not root.left and not root.right:
        return target == root.val
    return has_path_sum(root.left, target - root.val) or
       has_path_sum(root.right, target - root.val)
```

4 Path Sum II

4.1 Problem Statement

Given a binary tree and target sum, return all root-to-leaf paths summing to target.

4.2 Dry Run on Test Cases

- **Test Case 1**: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], target = 22 → Output: [[5,4,11,2],[5,8,4,5]]
- **Test Case 2**: root = [1,2,3], target = 5 → Output: []
- **Test Case 3**: root = [1], target = 1 → Output: [[1]]
- **Test Case 4**: root = [], target = 0 → Output: []

4.3 Algorithm

1. Use DFS with backtracking.
2. Track current path and sum.
3. At leaf, add path to result if sum equals target.

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

4.4 Python Solution

```
def path_sum(root, target):
    result = []

    def dfs(node, curr_sum, path):
        if not node:
            return
        curr_sum += node.val
        path.append(node.val)
        if not node.left and not node.right and curr_sum ==
           target:
            result.append(path[:])
        dfs(node.left, curr_sum, path)
        dfs(node.right, curr_sum, path)
        path.pop()

    dfs(root, 0, [])
    return result
```

5 Flatten Binary Tree to Linked List

5.1 Problem Statement

Given a binary tree, flatten it to a linked list in-place (preorder traversal).

5.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,5,3,4,null,6] → Output: [1,null,2,null,3,null,4,null,5,null,6]

- **Test Case 2**: root = [] → Output: []
- **Test Case 3**: root = [1] → Output: [1]
- **Test Case 4**: root = [1,2,null,3] → Output: [1,null,2,null,3]

5.3 Algorithm

1. Use recursive preorder traversal.
2. For each node, flatten left and right subtrees.
3. Connect left subtree to right, set left to None.

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

5.4 Python Solution

```python
def flatten(root):
    def flatten_helper(node):
        if not node:
            return None
        if not node.left and not node.right:
            return node

        left_tail = flatten_helper(node.left)
        right_tail = flatten_helper(node.right)

        if left_tail:
            left_tail.right = node.right
            node.right = node.left
            node.left = None

        return right_tail if right_tail else left_tail if
           left_tail else node

    flatten_helper(root)
```

6 Construct Binary Tree from Preorder and Inorder Traversal

6.1 Problem Statement

Given preorder and inorder traversals, construct the binary tree.

6.2 Dry Run on Test Cases

- **Test Case 1**: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7] → Output: [3,9,20,null,null,15,7]
- **Test Case 2**: preorder = [1], inorder = [1] → Output: [1]

- **Test Case 3**: preorder = [], inorder = [] → Output: []
- **Test Case 4**: preorder = [1,2], inorder = [2,1] → Output: [1,2]

6.3 Algorithm

1. Use preorder's first element as root.
2. Find root in inorder to split left and right subtrees.
3. Recursively build left and right subtrees.

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

6.4 Python Solution

```
def build_tree(preorder, inorder):
    if not preorder or not inorder:
        return None

    inorder_index = {val: idx for idx, val in enumerate(inorder)}

    def build(pre_start, pre_end, in_start, in_end):
        if pre_start > pre_end:
            return None

        root_val = preorder[pre_start]
        root = TreeNode(root_val)
        root_idx = inorder_index[root_val]

        left_size = root_idx - in_start
        root.left = build(pre_start + 1, pre_start + left_size,
            in_start, root_idx - 1)
        root.right = build(pre_start + left_size + 1, pre_end,
            root_idx + 1, in_end)
        return root

    return build(0, len(preorder) - 1, 0, len(inorder) - 1)
```

7 Validate Binary Search Tree

7.1 Problem Statement

Given a binary tree, determine if it is a valid BST (left subtree < node < right subtree).

7.2 Dry Run on Test Cases

- **Test Case 1**: root = [2,1,3] → Output: True
- **Test Case 2**: root = [5,1,4,null,null,3,6] → Output: False

- **Test Case 3**: root = [] → Output: True
- **Test Case 4**: root = [1] → Output: True

7.3 Algorithm

1. Use DFS with range checking.
2. Each node must be within (min, max) range.
3. Update ranges for left (min, node.val) and right (node.val, max).

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

7.4 Python Solution

```
def is_valid_bst(root):
    def validate(node, min_val, max_val):
        if not node:
            return True
        if node.val <= min_val or node.val >= max_val:
            return False
        return validate(node.left, min_val, node.val) and
           validate(node.right, node.val, max_val)

    return validate(root, float('-inf'), float('inf'))
```

8 Kth Smallest Element in a BST

8.1 Problem Statement

Given a BST and integer k, find the kth smallest element.

8.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,1,4,null,2], k = 1 → Output: 1
- **Test Case 2**: root = [5,3,6,2,4,null,null,1], k = 3 → Output: 3
- **Test Case 3**: root = [1], k = 1 → Output: 1
- **Test Case 4**: root = [], k = 1 → Output: None

8.3 Algorithm

1. Perform inorder traversal (iterative).
2. Track count of visited nodes.
3. Return value when count == k.

Time Complexity: $O(h + k)$ **Space Complexity**: $O(h)$

8.4 Python Solution

```python
def kth_smallest(root, k):
    stack = []
    curr = root
    count = 0

    while curr or stack:
        while curr:
            stack.append(curr)
            curr = curr.left
        curr = stack.pop()
        count += 1
        if count == k:
            return curr.val
        curr = curr.right
    return None
```

9 Lowest Common Ancestor in BST

9.1 Problem Statement

Given a BST and two nodes, find their lowest common ancestor.

9.2 Dry Run on Test Cases

- **Test Case 1**: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8 → Output: 6
- **Test Case 2**: root = [6,2,8], p = 2, q = 4 → Output: 2
- **Test Case 3**: root = [2,1], p = 2, q = 1 → Output: 2
- **Test Case 4**: root = [1], p = 1, q = 1 → Output: 1

9.3 Algorithm

1. Traverse from root.
2. If both p and q are less than root, go left.
3. If both greater, go right.
4. Else, root is LCA.

Time Complexity: $O(h)$ **Space Complexity**: $O(1)$

9.4 Python Solution

```python
def lowest_common_ancestor(root, p, q):
    curr = root
    while curr:
        if p.val < curr.val and q.val < curr.val:
            curr = curr.left
        elif p.val > curr.val and q.val > curr.val:
            curr = curr.right
        else:
            return curr
    return None
```

10 Binary Tree Zigzag Level Order Traversal

10.1 Problem Statement

Given a binary tree, return its zigzag level order traversal (left to right, then right to left).

10.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: [[3],[20,9],[15,7]]
- **Test Case 2**: root = [1] → Output: [[1]]
- **Test Case 3**: root = [] → Output: []
- **Test Case 4**: root = [1,2,3,4,null,null,5] → Output: [[1],[3,2],[4,5]]

10.3 Algorithm

1. Use BFS with queue for level order.
2. Track direction (left-to-right or right-to-left).
3. Reverse level nodes if right-to-left.

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

10.4 Python Solution

```python
from collections import deque

def zigzag_level_order(root):
    if not root:
        return []

    result = []
    queue = deque([root])
    left_to_right = True
```

```
    while queue:
        level_size = len(queue)
        current_level = []
        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        if not left_to_right:
            current_level.reverse()
        result.append(current_level)
        left_to_right = not left_to_right
    return result
```

11 Binary Tree Maximum Path Sum

11.1 Problem Statement

Given a binary tree, find the maximum path sum (path can include any node).

11.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,3] → Output: 6 (2->1->3)
- **Test Case 2**: root = [-10,9,20,null,null,15,7] → Output: 42 (15->20->7)
- **Test Case 3**: root = [1] → Output: 1
- **Test Case 4**: root = [-3] → Output: -3

11.3 Algorithm

1. Use DFS to compute max path sum through each node.
2. Track global max sum.
3. For each node, return max path sum to parent (single branch).

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

11.4 Python Solution

```
def max_path_sum(root):
    max_sum = float('-inf')

    def max_gain(node):
        nonlocal max_sum
```

```
        if not node:
            return 0
        left_gain = max(max_gain(node.left), 0)
        right_gain = max(max_gain(node.right), 0)
        current_sum = node.val + left_gain + right_gain
        max_sum = max(max_sum, current_sum)
        return node.val + max(left_gain, right_gain)

    max_gain(root)
    return max_sum
```

12 Insert into a BST

12.1 Problem Statement

Given a BST and a value, insert the value and return the root.

12.2 Dry Run on Test Cases

- **Test Case 1**: root = [4,2,7,1,3], val = 5 → Output: [4,2,7,1,3,5]
- **Test Case 2**: root = [], val = 1 → Output: [1]
- **Test Case 3**: root = [1], val = 2 → Output: [1,null,2]
- **Test Case 4**: root = [4,2,7], val = 4 → Output: [4,2,7,null,null,null,4]

12.3 Algorithm

1. If root is None, create new node.
2. If val $<$ root.val, insert into left subtree.
3. If val $>$ root.val, insert into right subtree.

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

12.4 Python Solution

```
def insert_into_bst(root, val):
    if not root:
        return TreeNode(val)

    if val < root.val:
        root.left = insert_into_bst(root.left, val)
    elif val > root.val:
        root.right = insert_into_bst(root.right, val)
    return root
```

13 Delete Node in a BST

13.1 Problem Statement

Given a BST and a key, delete the node with that key and return the root.

13.2 Dry Run on Test Cases

* **Test Case 1**: root = [5,3,6,2,4,null,7], key = 3 → Output: [5,4,6,2,null,null,7]
* **Test Case 2**: root = [5,3,6,2,4,null,7], key = 0 → Output: [5,3,6,2,4,null,7]
* **Test Case 3**: root = [], key = 0 → Output: []
* **Test Case 4**: root = [1], key = 1 → Output: []

13.3 Algorithm

1. Find node to delete.
2. If leaf, remove it.
3. If one child, replace with child.
4. If two children, replace with successor (smallest in right subtree).

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

13.4 Python Solution

```
def delete_node(root, key):
    if not root:
        return None

    if key < root.val:
        root.left = delete_node(root.left, key)
    elif key > root.val:
        root.right = delete_node(root.right, key)
    else:
        if not root.left:
            return root.right
        if not root.right:
            return root.left
        successor = root.right
        while successor.left:
            successor = successor.left
        root.val = successor.val
        root.right = delete_node(root.right, successor
            .val)
    return root
```

14 Binary Search Tree Iterator

14.1 Problem Statement

Implement an iterator over a BST with next() and hasNext() operations.

14.2 Dry Run on Test Cases

- **Test Case 1**: root = [7,3,15,null,null,9,20], next() → 3, next() → 7, hasNext() → True
- **Test Case 2**: root = [1], next() → 1, hasNext() → False
- **Test Case 3**: root = [], hasNext() → False
- **Test Case 4**: root = [3,1,4], next() → 1, next() → 3

14.3 Algorithm

1. Use stack for inorder traversal.
2. Push all left nodes from root.
3. next(): pop node, push all left nodes of its right subtree.
4. hasNext(): check if stack is non-empty.

Time Complexity: $O(1)$ average for next/hasNext **Space Complexity**: $O(h)$

14.4 Python Solution

```python
class BSTIterator:
    def __init__(self, root):
        self.stack = []
        self._push_left(root)

    def _push_left(self, node):
        while node:
            self.stack.append(node)
            node = node.left

    def next(self):
        node = self.stack.pop()
        self._push_left(node.right)
        return node.val

    def hasNext(self):
        return len(self.stack) > 0
```

15 Binary Tree Right Side View

15.1 Problem Statement

Given a binary tree, return the values of nodes visible from the right side.

15.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,3,null,5,null,4] → Output: [1,3,4]
- **Test Case 2**: root = [1,null,3] → Output: [1,3]
- **Test Case 3**: root = [] → Output: []
- **Test Case 4**: root = [1] → Output: [1]

15.3 Algorithm

1. Use BFS, process level by level.
2. Add last node of each level to result.

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

15.4 Python Solution

```python
from collections import deque

def right_side_view(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        for i in range(level_size):
            node = queue.popleft()
            if i == level_size - 1:
                result.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
    return result
```

16 Count Complete Tree Nodes

16.1 Problem Statement

Given a complete binary tree, count the number of nodes.

16.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,3,4,5,6] → Output: 6
- **Test Case 2**: root = [] → Output: 0
- **Test Case 3**: root = [1] → Output: 1
- **Test Case 4**: root = [1,2,3,4] → Output: 4

16.3 Algorithm

1. Check if left and right heights are equal.
2. If equal, return $2^h - 1$ (perfect binary tree).
3. Else, recursively count left and right subtrees.

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

16.4 Python Solution

```python
def count_nodes(root):
    def get_height(node, direction):
        height = 0
        while node:
            height += 1
            node = node.left if direction == 'left'
                else node.right
        return height

    if not root:
        return 0

    left_height = get_height(root, 'left')
    right_height = get_height(root, 'right')

    if left_height == right_height:
        return (1 << left_height) - 1
    return 1 + count_nodes(root.left) + count_nodes
       (root.right)
```

17 Kth Largest Element in an Array

17.1 Problem Statement

Given an array and integer k, find the kth largest element.

17.2 Dry Run on Test Cases

- **Test Case 1**: nums = [3,2,1,5,6,4], k = 2 → Output: 5
- **Test Case 2**: nums = [3,2,3,1,2,4,5,5,6], k = 4 → Output: 4
- **Test Case 3**: nums = [1], k = 1 → Output: 1
- **Test Case 4**: nums = [], k = 1 → Output: None

17.3 Algorithm

1. Use min-heap of size k.
2. Push elements; if heap size > k, pop smallest.
3. Return heap top.

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

17.4 Python Solution

```python
import heapq

def find_kth_largest(nums, k):
    if not nums:
        return None
    heap = []
    for num in nums:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)
    return heap[0]
```

18 Top K Frequent Elements

18.1 Problem Statement

Given an array and integer k, return the k most frequent elements.

18.2 Dry Run on Test Cases

- **Test Case 1**: nums = [1,1,1,2,2,3], k = 2 → Output: [1,2]
- **Test Case 2**: nums = [1], k = 1 → Output: [1]
- **Test Case 3**: nums = [1,2], k = 2 → Output: [1,2]
- **Test Case 4**: nums = [], k = 1 → Output: []

18.3 Algorithm

1. Count frequency of each element.
2. Use min-heap of size k to store elements by frequency.
3. Return heap elements.

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

18.4 Python Solution

```python
from collections import Counter
import heapq

def top_k_frequent(nums, k):
    if not nums:
        return []

    count = Counter(nums)
    heap = []
    for num, freq in count.items():
        heapq.heappush(heap, (freq, num))
        if len(heap) > k:
            heapq.heappop(heap)

    return [num for _, num in heap]
```

19 Median from Data Stream

19.1 Problem Statement

Design a data structure to find the median of a stream of numbers.

19.2 Dry Run on Test Cases

- **Test Case 1**: addNum(1), addNum(2), findMedian() → 1.5
- **Test Case 2**: addNum(3), findMedian() → 3
- **Test Case 3**: addNum(1), addNum(2), addNum(3), findMedian() → 2
- **Test Case 4**: addNum(4), addNum(5), findMedian() → 4.5

19.3 Algorithm

1. Use two heaps: max-heap for lower half, min-heap for upper half.
2. Balance heaps so max-heap has at most one more element.

3. Median is average of tops or top of max-heap.

Time Complexity: $O(\log n)$ for addNum, $O(1)$ for findMedian **Space Complexity**: $O(n)$

19.4 Python Solution

```python
import heapq

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

    def addNum(self, num):
        if len(self.small) == 0 or num < -self.small[0]:
            heapq.heappush(self.small, -num)
        else:
            heapq.heappush(self.large, num)

        # Balance heaps
        if len(self.small) > len(self.large) + 1:
            heapq.heappush(self.large, -heapq.heappop(self.small))
        elif len(self.large) > len(self.small):
            heapq.heappush(self.small, -heapq.heappop(self.large))

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

20 Merge k Sorted Lists

20.1 Problem Statement

Given k sorted linked lists, merge them into one sorted linked list.

20.2 Dry Run on Test Cases

- **Test Case 1**: lists = [[1,4,5],[1,3,4],[2,6]] → Output: [1,1,2,3,4,4,5,6]
- **Test Case 2**: lists = [] → Output: []
- **Test Case 3**: lists = [[]] → Output: []
- **Test Case 4**: lists = [[1]] → Output: [1]

20.3 Algorithm

1. Use min-heap to store heads of lists.
2. Pop smallest node, add to result, push next node if exists.
3. Continue until heap is empty.

Time Complexity: $O(n \log k)$ (n = total nodes) **Space Complexity**: $O(k)$

20.4 Python Solution

```python
import heapq

def merge_k_lists(lists):
    dummy = ListNode(0)
    curr = dummy
    heap = []

    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(heap, (lst.val, i, lst))

    while heap:
        val, i, node = heapq.heappop(heap)
        curr.next = node
        curr = curr.next
        if node.next:
            heapq.heappush(heap, (node.next.val, i,
                node.next))

    return dummy.next
```

21 Find Median in Two Sorted Arrays

21.1 Problem Statement

Given two sorted arrays, find the median of the merged array.

21.2 Dry Run on Test Cases

- **Test Case 1**: nums1 = [1,3], nums2 = [2] → Output: 2.0
- **Test Case 2**: nums1 = [1,2], nums2 = [3,4] → Output: 2.5
- **Test Case 3**: nums1 = [], nums2 = [1] → Output: 1.0
- **Test Case 4**: nums1 = [2], nums2 = [] → Output: 2.0

21.3 Algorithm

1. Use binary search on smaller array to find partition.
2. Ensure left side of partition <= right side for both arrays.
3. Compute median based on total length (odd or even).

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

21.4 Python Solution

```python
def find_median_sorted_arrays(nums1, nums2):
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    x, y = len(nums1), len(nums2)
    left, right = 0, x

    while left <= right:
        partition_x = (left + right) // 2
        partition_y = (x + y + 1) // 2 -
           partition_x

        left_x = nums1[partition_x - 1] if
           partition_x > 0 else float('-inf')
        right_x = nums1[partition_x] if partition_x
            < x else float('inf')
        left_y = nums2[partition_y - 1] if
           partition_y > 0 else float('-inf')
        right_y = nums2[partition_y] if partition_y
            < y else float('inf')

        if left_x <= right_y and left_y <= right_x:
            if (x + y) % 2 == 0:
                return (max(left_x, left_y) + min(
                   right_x, right_y)) / 2
            return max(left_x, left_y)
        elif left_x > right_y:
            right = partition_x - 1
        else:
            left = partition_x + 1
```

22 Design Min Heap

22.1 Problem Statement

Implement a min-heap with insert, $\text{extract}_m in, and get_m in operations$.

22.2 Dry Run on Test Cases

- **Test Case 1**: insert(3), insert(2), get$_min()\rightarrow$ 2, extract$_min()\rightarrow$ 2
- **Test Case 2**: insert(1), get$_min()\rightarrow$ 1
- **Test Case 3**: extract$_min()\rightarrow$ None
- **Test Case 4**: insert(5), insert(1), extract$_min()\rightarrow$ 1

22.3 Algorithm

1. Use array to store heap.
2. Insert: append and bubble up.
3. Extract$_min$: $removeroot, placelastelementatroot, bubbledown. Get_min$: $returnroot$. **Time Complexity**: $O(\log n)$ for insert/extract$_min$,O(1)$forget_min$ **Space Complexity** :O(n)

22.4 Python Solution

```python
class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):
        self.heap.append(val)
        self._bubble_up(len(self.heap) - 1)

    def extract_min(self):
        if not self.heap:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()

        min_val = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._bubble_down(0)
        return min_val

    def get_min(self):
        return self.heap[0] if self.heap else None

    def _bubble_up(self, index):
        parent = (index - 1) // 2
        if index > 0 and self.heap[index] < self.
           heap[parent]:
            self.heap[index], self.heap[parent] =
               self.heap[parent], self.heap[index]
            self._bubble_up(parent)

    def _bubble_down(self, index):
```

```
        smallest = index
        left = 2 * index + 1
        right = 2 * index + 2

        if left < len(self.heap) and self.heap[left
           ] < self.heap[smallest]:
            smallest = left
        if right < len(self.heap) and self.heap[
           right] < self.heap[smallest]:
            smallest = right

        if smallest != index:
            self.heap[index], self.heap[smallest] =
                self.heap[smallest], self.heap[
               index]
            self._bubble_down(smallest)
```

23 Find K Closest Points to Origin

23.1 Problem Statement

Given an array of points and integer k, return the k closest points to the origin (0,0).

23.2 Dry Run on Test Cases

4. **Test Case 1**: points = [[1,3],[-2,2]], k = 1 → Output: [[-2,2]]

- **Test Case 2**: points = [[3,3],[5,-1],[-2,4]], k = 2 → Output: [[3,3],[-2,4]]
- **Test Case 3**: points = [[1,1]], k = 1 → Output: [[1,1]]
- **Test Case 4**: points = [], k = 1 → Output: []

23.3 Algorithm

1. Use max-heap of size k to store points by distance.
2. For each point, compute distance, push to heap, pop if size > k.
3. Return heap contents.

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

23.4 Python Solution

```
import heapq

def k_closest(points, k):
    if not points:
        return []
```

```
    heap = []
    for x, y in points:
        dist = -(x*x + y*y)  # Negate for max heap
        heapq.heappush(heap, (dist, x, y))
        if len(heap) > k:
            heapq.heappop(heap)

    return [[x, y] for _, x, y in heap]
```

24 Course Schedule

24.1 Problem Statement

Given a number of courses and prerequisites, determine if all courses can be completed.

24.2 Dry Run on Test Cases

- **Test Case 1**: numCourses = 2, prerequisites = [[1,0]] → Output: True
- **Test Case 2**: numCourses = 2, prerequisites = [[1,0],[0,1]] → Output: False
- **Test Case 3**: numCourses = 1, prerequisites = [] → Output: True
- **Test Case 4**: numCourses = 3, prerequisites = [[1,0],[2,1]] → Output: True

24.3 Algorithm

1. Build adjacency list and in-degree count.
2. Use topological sort with queue for nodes with in-degree 0.
3. If all courses visited, return True; else, cycle exists.

Time Complexity: $O(V + E)$ **Space Complexity**: $O(V + E)$

24.4 Python Solution

```
from collections import defaultdict, deque

def can_finish(numCourses, prerequisites):
    graph = defaultdict(list)
    in_degree = [0] * numCourses

    for dest, src in prerequisites:
        graph[src].append(dest)
        in_degree[dest] += 1

    queue = deque([i for i in range(numCourses) if
       in_degree[i] == 0])
```

```
    count = 0

    while queue:
        node = queue.popleft()
        count += 1
        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    return count == numCourses
```

25 Course Schedule II

25.1 Problem Statement

Given a number of courses and prerequisites, return the order to take all courses.

25.2 Dry Run on Test Cases

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

25.3 Algorithm

1. Build adjacency list and in-degree count.
2. Use topological sort, add nodes with in-degree 0 to queue.
3. Collect order; return empty list if cycle detected.

Time Complexity: $O(V + E)$ **Space Complexity**: $O(V + E)$

25.4 Python Solution

```
from collections import defaultdict, deque

def find_order(numCourses, prerequisites):
    graph = defaultdict(list)
    in_degree = [0] * numCourses

    for dest, src in prerequisites:
        graph[src].append(dest)
        in_degree[dest] += 1
```

```
10
11      queue = deque([i for i in range(numCourses) if
            in_degree[i] == 0])
12      order = []
13
14      while queue:
15          node = queue.popleft()
16          order.append(node)
17          for neighbor in graph[node]:
18              in_degree[neighbor] -= 1
19              if in_degree[neighbor] == 0:
20                  queue.append(neighbor)
21
22      return order if len(order) == numCourses else
            []
```

26 Clone Graph

26.1 Problem Statement

Given a graph node, return a deep copy (clone) of the graph.

26.2 Dry Run on Test Cases

- **Test Case 1**: node = [[2,4],[1,3],[2,4],[1,3]] → Output: Cloned graph
- **Test Case 2**: node = [[]] → Output: Cloned single node
- **Test Case 3**: node = [] → Output: None
- **Test Case 4**: node = [[2],[1]] → Output: Cloned graph

26.3 Algorithm

1. Use DFS with hashmap to store cloned nodes.
2. For each node, create clone, recursively clone neighbors.
3. Avoid cycles using hashmap.

Time Complexity: $O(V + E)$ **Space Complexity**: $O(V)$

26.4 Python Solution

```
1  class Node:
2      def __init__(self, val=0, neighbors=None):
3          self.val = val
4          self.neighbors = neighbors if neighbors is
               not None else []
5
```

```python
def clone_graph(node):
    if not node:
        return None

    cloned = {}

    def dfs(node):
        if node in cloned:
            return cloned[node]

        clone = Node(node.val)
        cloned[node] = clone
        for neighbor in node.neighbors:
            clone.neighbors.append(dfs(neighbor))
        return clone

    return dfs(node)
```

27 Number of Islands

27.1 Problem Statement

Given a 2D grid of '1's (land) and '0's (water), count the number of islands.

27.2 Dry Run on Test Cases

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

27.3 Algorithm

1. Use DFS to mark all connected '1's as visited.
2. For each unvisited '1', increment island count and perform DFS.
3. Return total islands.

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

27.4 Python Solution

```python
def num_islands(grid):
    if not grid:
        return 0
```

```
    rows, cols = len(grid), len(grid[0])
    islands = 0

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or grid[i][j] != "1":
            return
        grid[i][j] = "0"  # Mark as visited
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == "1":
                islands += 1
                dfs(i, j)
    return islands
```

28 Flood Fill

28.1 Problem Statement

Given an image, starting pixel, and new color, flood fill the connected pixels of the same color.

28.2 Dry Run on Test Cases

- **Test Case 1**: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, newColor = 2 → Output: [[2,2,2],[2,2,0],[2,0,1]]
- **Test Case 2**: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, newColor = 0 → Output: [[0,0,0],[0,0,0]]
- **Test Case 3**: image = [[1]], sr = 0, sc = 0, newColor = 2 → Output: [[2]]
- **Test Case 4**: image = [], sr = 0, sc = 0, newColor = 1 → Output: []

28.3 Algorithm

1. Use DFS to change color of connected pixels.
2. If pixel matches old color, change to new color and recurse on neighbors.
3. Return modified image.

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

28.4 Python Solution

```python
def flood_fill(image, sr, sc, newColor):
    if not image or image[sr][sc] == newColor:
        return image

    rows, cols = len(image), len(image[0])
    old_color = image[sr][sc]

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or image[i][j] != old_color:
            return
        image[i][j] = newColor
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    dfs(sr, sc)
    return image
```

29 Rotting Oranges

29.1 Problem Statement

Given a grid of oranges (fresh=1, rotten=2, empty=0), return the minimum time for all to rot.

29.2 Dry Run on Test Cases

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

29.3 Algorithm

1. Use BFS starting from all rotten oranges.
2. Track time and fresh oranges.
3. If fresh remain, return -1; else return time.

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

29.4 Python Solution

```python
from collections import deque

def oranges_rotting(grid):
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    queue = deque()
    fresh = 0
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 2:
                queue.append((i, j))
            elif grid[i][j] == 1:
                fresh += 1

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

    while queue and fresh:
        for _ in range(len(queue)):
            i, j = queue.popleft()
            for di, dj in directions:
                ni, nj = i + di, j + dj
                if 0 <= ni < rows and 0 <= nj <
                   cols and grid[ni][nj] == 1:
                    grid[ni][nj] = 2
                    fresh -= 1
                    queue.append((ni, nj))
        time += 1

    return time if fresh == 0 else -1
```

30 Word Ladder

30.1 Problem Statement

Given two words and a word list, find the shortest transformation sequence length from beginWord to endWord.

30.2 Dry Run on Test Cases

- **Test Case 1**: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"] → Output: 5
- **Test Case 2**: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"] → Output: 0
- **Test Case 3**: beginWord = "a", endWord = "c", wordList = ["a","b","c"] → Output: 2

- **Test Case 4**: beginWord = "hit", endWord = "hit", wordList = [] → Output: 1

30.3 Algorithm

1. Use BFS to find shortest path.
2. For each word, generate all possible one-letter changes.
3. Track visited words and level.

Time Complexity: $O(N \cdot 26 \cdot L)$ (N = wordList size, L = word length) **Space Complexity**: $O(N)$

30.4 Python Solution

```python
from collections import deque, defaultdict

def ladder_length(beginWord, endWord, wordList):
    if endWord not in wordList:
        return 0

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

    while queue:
        word, level = queue.popleft()
        if word == endWord:
            return level

        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                new_word = word[:i] + c + word[i
                    +1:]
                if new_word in word_set and
                    new_word not in visited:
                    visited.add(new_word)
                    queue.append((new_word, level +
                        1))

    return 0
```

31 Shortest Path in Binary Matrix

31.1 Problem Statement

Given an n x n binary matrix, find the shortest path length from (0,0) to (n-1,n-1) with 0s.

31.2 Dry Run on Test Cases

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

31.3 Algorithm

1. Use BFS starting from (0,0).
2. Explore 8 directions for each cell.
3. Return distance to (n-1,n-1) or -1 if unreachable.

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

31.4 Python Solution

```python
from collections import deque

def shortest_path_binary_matrix(grid):
    if not grid or grid[0][0] == 1:
        return -1

    n = len(grid)
    if n == 1:
        return 1 if grid[0][0] == 0 else -1

    queue = deque([(0, 0, 1)])
    grid[0][0] = 1  # Mark as visited
    directions = [(1,0), (-1,0), (0,1), (0,-1),
       (1,1), (-1,-1), (1,-1), (-1,1)]

    while queue:
        i, j, dist = queue.popleft()
        if i == n-1 and j == n-1:
            return dist

        for di, dj in directions:
            ni, nj = i + di, j + dj
            if 0 <= ni < n and 0 <= nj < n and grid
               [ni][nj] == 0:
                grid[ni][nj] = 1
                queue.append((ni, nj, dist + 1))

    return -1
```

Solutions to DSA Questions 111-141 (Graphs, DP, Bit Manipulation) For 1-2 Years Experience Roles at EPAM

Compiled on September 27, 2025

Introduction

This document provides detailed solutions for 31 Data Structures and Algorithms (DSA) problems (questions 111 to 141) from the Graphs, Dynamic Programming, and Bit Manipulation 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. Dynamic programming problems include both memoization and tabulation approaches where applicable.

Contents

1 Pacific Atlantic Water Flow

1.1 Problem Statement

Given an m x n matrix of heights, find all cells where water can flow to both Pacific and Atlantic oceans.

1.2 Dry Run on Test Cases

- **Test Case 1**: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]] → Output: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]
- **Test Case 2**: heights = [[1]] → Output: [[0,0]]
- **Test Case 3**: heights = [[1,2],[2,1]] → Output: [[0,0],[0,1],[1,0],[1,1]]
- **Test Case 4**: heights = [] → Output: []

1.3 Algorithm

1. Use DFS from Pacific (top/left) and Atlantic (bottom/right) borders.

2. Mark reachable cells for each ocean.

3. Return cells reachable by both.

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

1.4 Python Solution

```python
def pacific_atlantic(heights):
    if not heights or not heights[0]:
        return []

    rows, cols = len(heights), len(heights[0])
    pacific = set()
    atlantic = set()

    def dfs(i, j, visited, prev_height):
        if i < 0 or i >= rows or j < 0 or j >= cols or (i, j) in
           visited or heights[i][j] < prev_height:
            return
        visited.add((i, j))
        for di, dj in [(1,0), (-1,0), (0,1), (0,-1)]:
            dfs(i + di, j + dj, visited, heights[i][j])

    # Pacific: top and left borders
    for j in range(cols):
        dfs(0, j, pacific, float('-inf'))
    for i in range(rows):
        dfs(i, 0, pacific, float('-inf'))

    # Atlantic: bottom and right borders
    for j in range(cols):
        dfs(rows-1, j, atlantic, float('-inf'))
    for i in range(rows):
        dfs(i, cols-1, atlantic, float('-inf'))

    return list(pacific & atlantic)
```

2 Surrounded Regions

2.1 Problem Statement

Given a 2D board of 'X' and 'O', flip all 'O's surrounded by 'X' to 'X'.

2.2 Dry Run on Test Cases

- **Test Case 1**: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]] → Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]

- **Test Case 2**: board = [["O"]] → Output: [["O"]]

- **Test Case 3**: board = [] → Output: []
- **Test Case 4**: board = [["X"]] → Output: [["X"]]

2.3 Algorithm

1. Use DFS to mark 'O's connected to border as safe.
2. Flip unmarked 'O's to 'X'.
3. Restore safe 'O's.

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

2.4 Python Solution

```python
def solve(board):
    if not board or not board[0]:
        return

    rows, cols = len(board), len(board[0])

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols or board[i][j
           ] != 'O':
            return
        board[i][j] = '#'
        for di, dj in [(1,0), (-1,0), (0,1), (0,-1)]:
            dfs(i + di, j + dj)

    # Mark border-connected 'O's
    for i in range(rows):
        if board[i][0] == 'O':
            dfs(i, 0)
        if board[i][cols-1] == 'O':
            dfs(i, cols-1)
    for j in range(cols):
        if board[0][j] == 'O':
            dfs(0, j)
        if board[rows-1][j] == 'O':
            dfs(rows-1, j)

    # Flip 'O' to 'X', '#' to 'O'
    for i in range(rows):
        for j in range(cols):
            if board[i][j] == 'O':
                board[i][j] = 'X'
            elif board[i][j] == '#':
                board[i][j] = 'O'
```

3 Graph Valid Tree

3.1 Problem Statement

Given n nodes and edges, check if they form a valid tree (connected, no cycles).

3.2 Dry Run on Test Cases

- **Test Case 1**: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]] → Output: True
- **Test Case 2**: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]] → Output: False
- **Test Case 3**: n = 1, edges = [] → Output: True
- **Test Case 4**: n = 2, edges = [] → Output: False

3.3 Algorithm

1. Use DFS to detect cycles and check connectivity.
2. Build adjacency list.
3. Ensure all nodes visited and no cycles.

Time Complexity: $O(V + E)$ **Space Complexity**: $O(V + E)$

3.4 Python Solution

```python
from collections import defaultdict

def valid_tree(n, edges):
    if len(edges) != n - 1:
        return False

    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = set()

    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if not dfs(neighbor, node):
                    return False
            elif neighbor != parent:
                return False
        return True

```

```
    return dfs(0, -1) and len(visited) == n
```

4 Minimum Spanning Tree (Kruskal's Algorithm)

4.1 Problem Statement

Given a weighted undirected graph, find the minimum spanning tree weight using Kruskal's algorithm.

4.2 Dry Run on Test Cases

- **Test Case 1**: n = 4, edges = [[0,1,1],[0,2,2],[1,2,5],[1,3,1],[2,3,4]] → Output: 4
- **Test Case 2**: n = 1, edges = [] → Output: 0
- **Test Case 3**: n = 2, edges = [[0,1,1]] → Output: 1
- **Test Case 4**: n = 3, edges = [[0,1,2],[1,2,2]] → Output: 4

4.3 Algorithm

1. Sort edges by weight.
2. Use Union-Find to avoid cycles.
3. Add edges to MST if they connect different components.

Time Complexity: $O(E \log E)$ **Space Complexity**: $O(V)$

4.4 Python Solution

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

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

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
```

```
        return True

def minimum_spanning_tree(n, edges):
    edges.sort(key=lambda x: x[2])
    uf = UnionFind(n)
    total_weight = 0
    edges_used = 0

    for u, v, weight in edges:
        if uf.union(u, v):
            total_weight += weight
            edges_used += 1

    return total_weight if edges_used == n - 1 else -1
```

5 Dijkstra's Algorithm

5.1 Problem Statement

Given a weighted directed graph and source, find shortest paths to all vertices.

5.2 Dry Run on Test Cases

- **Test Case 1**: n = 4, edges = [[0,1,1],[0,2,4],[1,2,2],[1,3,6],[2,3,3]], source = 0 → Output: [0,1,3,6]
- **Test Case 2**: n = 1, edges = [], source = 0 → Output: [0]
- **Test Case 3**: n = 2, edges = [[0,1,1]], source = 0 → Output: [0,1]
- **Test Case 4**: n = 3, edges = [[0,1,2],[1,2,2]], source = 0 → Output: [0,2,4]

5.3 Algorithm

1. Use min-heap to store (distance, node).
2. Update distances to neighbors if shorter path found.
3. Return distance array.

Time Complexity: $O((V + E) \log V)$ **Space Complexity**: $O(V + E)$

5.4 Python Solution

```
import heapq
from collections import defaultdict

def dijkstra(n, edges, source):
    graph = defaultdict(list)
    for u, v, w in edges:
```

```
        graph[u].append((v, w))

    distances = [float('inf')] * n
    distances[source] = 0
    heap = [(0, source)]

    while heap:
        dist, node = heapq.heappop(heap)
        if dist > distances[node]:
            continue
        for neighbor, weight in graph[node]:
            if dist + weight < distances[neighbor]:
                distances[neighbor] = dist + weight
                heapq.heappush(heap, (dist + weight,
                    neighbor))

    return [d if d != float('inf') else -1 for d in
        distances]
```

6 Climbing Stairs

6.1 Problem Statement

Given n stairs, find the number of ways to climb (1 or 2 steps at a time).

6.2 Dry Run on Test Cases

* **Test Case 1**: n = 2 → Output: 2 ([1,1], [2])
* **Test Case 2**: n = 3 → Output: 3 ([1,1,1], [1,2], [2,1])
* **Test Case 3**: n = 1 → Output: 1 ([1])
* **Test Case 4**: n = 4 → Output: 5

6.3 Algorithm (Memoization)

1. Use recursive function with memoization.
2. Base cases: n=0 (1 way), n<0 (0 ways).
3. Return sum of ways for n-1 and n-2.

6.4 Algorithm (Tabulation)

1. Use DP array where dp[i] = ways to climb i stairs.
2. Initialize dp[0]=1, dp[1]=1.
3. dp[i] = dp[i-1] + dp[i-2] for i from 2 to n.

Time Complexity: $O(n)$ **Space Complexity**: $O(n)$ (memoization), $O(1)$ (tabulation)

6.5 Python Solution (Memoization)

```python
def climb_stairs_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 0:
        return 1 if n == 0 else 0
    memo[n] = climb_stairs_memo(n-1, memo) +
       climb_stairs_memo(n-2, memo)
    return memo[n]
```

6.6 Python Solution (Tabulation)

```python
def climb_stairs(n):
    if n <= 1:
        return 1
    dp = [0] * (n + 1)
    dp[0] = dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

7 Min Cost Climbing Stairs

7.1 Problem Statement

Given a cost array for climbing stairs, find minimum cost to reach the top.

7.2 Dry Run on Test Cases

* **Test Case 1**: cost = [10,15,20] → Output: 15
* **Test Case 2**: cost = [1,100,1,1,1,100,1,1,100,1] → Output: 6
* **Test Case 3**: cost = [0,1] → Output: 0
* **Test Case 4**: cost = [1] → Output: 0

7.3 Algorithm (Memoization)

1. Recurse with memoization for min cost from index i.
2. Base case: i >= len(cost) returns 0.
3. Min cost = cost[i] + min(cost(i+1), cost(i+2)).

7.4 Algorithm (Tabulation)

1. dp[i] = min cost to reach top from i.
2. dp[n] = 0, dp[n-1] = cost[n-1].
3. dp[i] = cost[i] + min(dp[i+1], dp[i+2]).

Time Complexity: $O(n)$ **Space Complexity**: $O(n)$ (memoization), $O(1)$ (tabulation)

7.5 Python Solution (Memoization)

```python
def min_cost_climbing_stairs_memo(cost):
    memo = {}
    def dp(i):
        if i >= len(cost):
            return 0
        if i in memo:
            return memo[i]
        memo[i] = cost[i] + min(dp(i+1), dp(i+2))
        return memo[i]
    return min(dp(0), dp(1))
```

7.6 Python Solution (Tabulation)

```python
def min_cost_climbing_stairs(cost):
    n = len(cost)
    dp = [0] * (n + 1)
    for i in range(n-1, -1, -1):
        dp[i] = cost[i] + min(dp[i+1], dp[i+2])
    return min(dp[0], dp[1])
```

8 House Robber

8.1 Problem Statement

Given an array of house values, find max amount to rob without robbing adjacent houses.

8.2 Dry Run on Test Cases

* **Test Case 1**: nums = [1,2,3,1] → Output: 4 (1+3)
* **Test Case 2**: nums = [2,7,9,3,1] → Output: 12 (2+9+1)
* **Test Case 3**: nums = [1] → Output: 1
* **Test Case 4**: nums = [] → Output: 0

8.3 Algorithm (Memoization)

1. Recurse with memoization for max from index i.
2. Base case: i >= len(nums) returns 0.
3. Max = max(nums[i] + dp(i+2), dp(i+1)).

8.4 Algorithm (Tabulation)

1. dp[i] = max amount from index i to end.
2. dp[n] = 0, dp[n-1] = nums[n-1].
3. dp[i] = max(nums[i] + dp[i+2], dp[i+1]).

Time Complexity: $O(n)$ **Space Complexity**: $O(n)$ (memoization), $O(1)$ (tabulation)

8.5 Python Solution (Memoization)

```python
def rob_memo(nums):
    memo = {}
    def dp(i):
        if i >= len(nums):
            return 0
        if i in memo:
            return memo[i]
        memo[i] = max(nums[i] + dp(i+2), dp(i+1))
        return memo[i]
    return dp(0)
```

8.6 Python Solution (Tabulation)

```python
def rob(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    dp = [0] * (len(nums) + 1)
    dp[-2] = nums[-1]
    for i in range(len(nums)-2, -1, -1):
        dp[i] = max(nums[i] + dp[i+2], dp[i+1])
    return dp[0]
```

9 House Robber II

9.1 Problem Statement

Given a circular array of house values, find max amount to rob without robbing adjacent houses.

9.2 Dry Run on Test Cases

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

9.3 Algorithm (Memoization)

1. Solve House Robber for nums[0:n-1] and nums[1:n].
2. Return max of both results.

9.4 Algorithm (Tabulation)

1. Use tabulation for two cases: exclude first and exclude last house.
2. dp[i] = max(nums[i] + dp[i+2], dp[i+1]).

Time Complexity: $O(n)$ **Space Complexity**: $O(n)$ (memoization), $O(1)$ (tabulation)

9.5 Python Solution (Tabulation)

```
def rob_circular(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    def rob_linear(arr):
        if not arr:
            return 0
        if len(arr) == 1:
            return arr[0]
        dp = [0] * (len(arr) + 1)
        dp[-2] = arr[-1]
        for i in range(len(arr)-2, -1, -1):
            dp[i] = max(arr[i] + dp[i+2], dp[i+1])
        return dp[0]

    return max(rob_linear(nums[:-1]), rob_linear(
       nums[1:]))
```

10 Coin Change

10.1 Problem Statement

Given coins and an amount, find minimum coins needed to make the amount.

10.2 Dry Run on Test Cases

- **Test Case 1**: coins = [1,2,5], amount = 11 → Output: 3 (5+5+1)
- **Test Case 2**: coins = [2], amount = 3 → Output: -1
- **Test Case 3**: coins = [1], amount = 0 → Output: 0
- **Test Case 4**: coins = [1,2], amount = 1 → Output: 1

10.3 Algorithm (Memoization)

1. Recurse with memoization for min coins for amount.
2. Base case: amount=0 returns 0, amount<0 returns inf.
3. Try each coin and take minimum.

10.4 Algorithm (Tabulation)

1. dp[i] = min coins for amount i.
2. Initialize dp[0]=0, others=inf.
3. For each amount, try each coin, update dp.

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

10.5 Python Solution (Memoization)

```
def coin_change_memo(coins, amount):
    memo = {}
    def dp(amt):
        if amt in memo:
            return memo[amt]
        if amt == 0:
            return 0
        if amt < 0:
            return float('inf')
        min_coins = float('inf')
        for coin in coins:
            min_coins = min(min_coins, dp(amt -
                coin) + 1)
        memo[amt] = min_coins
```

```
        return memo[amt]
    result = dp(amount)
    return result if result != float('inf') else -1
```

10.6 Python Solution (Tabulation)

```
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for i in range(1, amount + 1):
        for coin in coins:
            if i >= coin:
                dp[i] = min(dp[i], dp[i - coin] +
                    1)
    return dp[amount] if dp[amount] != float('inf')
        else -1
```

11 Coin Change II

11.1 Problem Statement

Given coins and an amount, find number of ways to make the amount.

11.2 Dry Run on Test Cases

- **Test Case 1**: amount = 5, coins = [1,2,5] → Output: 4 ([1,1,1,1,1], [1,1,1,2], [1,2,2], [5])
- **Test Case 2**: amount = 3, coins = [2] → Output: 0
- **Test Case 3**: amount = 10, coins = [10] → Output: 1
- **Test Case 4**: amount = 0, coins = [1] → Output: 1

11.3 Algorithm (Memoization)

1. Recurse with memoization for ways with index i and amount.
2. Base case: amount=0 returns 1, amount<0 or i>=len(coins) returns 0.
3. Sum ways by including or excluding current coin.

11.4 Algorithm (Tabulation)

1. dp[i] = ways to make amount i.
2. Initialize dp[0]=1.

3. For each coin, update dp[j] += dp[j-coin].

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

11.5 Python Solution (Memoization)

```python
def change_memo(amount, coins):
    memo = {}
    def dp(i, amt):
        if (i, amt) in memo:
            return memo[(i, amt)]
        if amt == 0:
            return 1
        if amt < 0 or i >= len(coins):
            return 0
        memo[(i, amt)] = dp(i, amt - coins[i]) + dp
            (i + 1, amt)
        return memo[(i, amt)]
    return dp(0, amount)
```

11.6 Python Solution (Tabulation)

```python
def change(amount, coins):
    dp = [0] * (amount + 1)
    dp[0] = 1
    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] += dp[i - coin]
    return dp[amount]
```

12 Longest Increasing Subsequence

12.1 Problem Statement

Given an array, find the length of the longest increasing subsequence.

12.2 Dry Run on Test Cases

- **Test Case 1**: nums = [10,9,2,5,3,7,101,18] → Output: 4 ([2,3,7,101])
- **Test Case 2**: nums = [0,1,0,3,2,3] → Output: 4
- **Test Case 3**: nums = [7,7,7,7] → Output: 1
- **Test Case 4**: nums = [] → Output: 0

12.3 Algorithm (Memoization)

1. Recurse with memoization for LIS ending at i with prev value.

2. Base case: i >= len(nums) returns 0.

3. Include nums[i] if greater than prev, else skip.

12.4 Algorithm (Tabulation)

1. dp[i] = length of LIS ending at i.

2. Initialize dp[i]=1.

3. For each i, check j<i where nums[j]<nums[i], update dp[i].

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

12.5 Python Solution (Memoization)

```python
def length_of_lis_memo(nums):
    memo = {}
    def dp(i, prev):
        if i >= len(nums):
            return 0
        if (i, prev) in memo:
            return memo[(i, prev)]
        take = 0
        if prev == -1 or nums[i] > nums[prev]:
            take = 1 + dp(i + 1, i)
        skip = dp(i + 1, prev)
        memo[(i, prev)] = max(take, skip)
        return memo[(i, prev)]
    return dp(0, -1)
```

12.6 Python Solution (Tabulation)

```python
def length_of_lis(nums):
    if not nums:
        return 0
    dp = [1] * len(nums)
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

13 Longest Common Subsequence

13.1 Problem Statement

Given two strings, find the length of their longest common subsequence.

13.2 Dry Run on Test Cases

- **Test Case 1**: text1 = "abcde", text2 = "ace" → Output: 3 ("ace")
- **Test Case 2**: text1 = "abc", text2 = "abc" → Output: 3
- **Test Case 3**: text1 = "abc", text2 = "def" → Output: 0
- **Test Case 4**: text1 = "", text2 = "a" → Output: 0

13.3 Algorithm (Memoization)

1. Recurse with memoization for LCS of prefixes i, j.
2. If characters match, add 1 and recurse.
3. Else, take max of skipping one character.

13.4 Algorithm (Tabulation)

1. dp[i][j] = LCS length for prefixes i, j.
2. If characters match, dp[i][j] = dp[i-1][j-1] + 1.
3. Else, dp[i][j] = max(dp[i-1][j], dp[i][j-1]).

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

13.5 Python Solution (Memoization)

```
def longest_common_subsequence_memo(text1, text2):
    memo = {}
    def dp(i, j):
        if i >= len(text1) or j >= len(text2):
            return 0
        if (i, j) in memo:
            return memo[(i, j)]
        if text1[i] == text2[j]:
            memo[(i, j)] = 1 + dp(i + 1, j + 1)
        else:
            memo[(i, j)] = max(dp(i + 1, j), dp(i,
                j + 1))
        return memo[(i, j)]
    return dp(0, 0)
```

13.6 Python Solution (Tabulation)

```
def longest_common_subsequence(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
```

```
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j
                    -1])
    return dp[m][n]
\end Prefeitura

% Problem 124
\section{Edit Distance}
\subsection{Problem Statement}
Given two strings, find minimum operations (insert,
    delete, replace) to convert one to the other.

\subsection{Dry Run on Test Cases}
\begin{itemize}
    \item \textbf{Test Case 1}: word1 = "horse",
       word2 = "ros" $\rightarrow$ Output: 3
    \item \textbf{Test Case 2}: word1 = "intention"
       , word2 = "execution" $\rightarrow$ Output:
       5
    \item \textbf{Test Case 3}: word1 = "", word2 =
        "a" $\rightarrow$ Output: 1
    \item \textbf{Test Case 4}: word1 = "abc",
       word2 = "abc" $\rightarrow$ Output: 0
\end{itemize]

\subsection{Algorithm (Memoization)}
\begin{enumerate}
    \item Recurse with memoization for min
       operations for prefixes i, j.
    \item If characters match, no operation needed.
    \item Else, try insert, delete, replace, take
       minimum.
\end{enumerate}
\subsection{Algorithm (Tabulation)}
\begin{enumerate}
    \item dp[i][j] = min operations for prefixes i,
        j.
    \item Initialize dp[0][j] = j, dp[i][0] = i.
    \item If match, dp[i][j] = dp[i-1][j-1].
    \item Else, dp[i][j] = min(insert, delete,
       replace) + 1.
\end{enumerate}
\textbf{Time Complexity}: $O(m \cdot n)$ \quad \
   textbf{Space Complexity}: $O(m \cdot n)$

\subsection{Python Solution (Memoization)}
\begin{lstlisting}
def min_distance_memo(word1, word2):
    memo = {}
```

```
    def dp(i, j):
        if i >= len(word1):
            return len(word2) - j
        if j >= len(word2):
            return len(word1) - i
        if (i, j) in memo:
            return memo[(i, j)]
        if word1[i] == word2[j]:
            memo[(i, j)] = dp(i + 1, j + 1)
        else:
            memo[(i, j)] = min(dp(i, j + 1), dp(i +
                1, j), dp(i + 1, j + 1)) + 1
        return memo[(i, j)]
    return dp(0, 0)
```

13.7 Python Solution (Tabulation)

```
def min_distance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for j in range(n + 1):
        dp[0][j] = j
    for i in range(m + 1):
        dp[i][0] = i
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i-1] == word2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = min(dp[i-1][j], dp[i][j
                   -1], dp[i-1][j-1]) + 1
    return dp[m][n]
```

14 Unique Paths

14.1 Problem Statement

Given an m x n grid, find number of unique paths from top-left to bottom-right (right/down moves).

14.2 Dry Run on Test Cases

- **Test Case 1**: m = 3, n = 7 → Output: 28
- **Test Case 2**: m = 3, n = 2 → Output: 3
- **Test Case 3**: m = 1, n = 1 → Output: 1
- **Test Case 4**: m = 2, n = 2 → Output: 2

14.3 Algorithm (Memoization)

1. Recurse with memoization for paths to (i,j).
2. Base case: (0,0) returns 1, out of bounds returns 0.
3. Paths = sum of paths from above and left.

14.4 Algorithm (Tabulation)

1. dp[i][j] = paths to (i,j).
2. Initialize first row and column to 1.
3. dp[i][j] = dp[i-1][j] + dp[i][j-1].

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

14.5 Python Solution (Memoization)

```python
def unique_paths_memo(m, n):
    memo = {}
    def dp(i, j):
        if i < 0 or j < 0:
            return 0
        if i == 0 and j == 0:
            return 1
        if (i, j) in memo:
            return memo[(i, j)]
        memo[(i, j)] = dp(i-1, j) + dp(i, j-1)
        return memo[(i, j)]
    return dp(m-1, n-1)
```

14.6 Python Solution (Tabulation)

```python
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[m-1][n-1]
```

15 Unique Paths II

15.1 Problem Statement

Given an m x n grid with obstacles, find number of unique paths to bottom-right.

15.2 Dry Run on Test Cases

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

15.3 Algorithm (Memoization)

1. Recurse with memoization for paths to (i,j).
2. Base case: obstacle or out of bounds returns 0, (0,0) returns 1 if no obstacle.
3. Paths = sum of paths from above and left.

15.4 Algorithm (Tabulation)

1. dp[i][j] = paths to (i,j).
2. Initialize dp[0][0] based on obstacle.
3. For each cell, dp[i][j] = dp[i-1][j] + dp[i][j-1] if no obstacle.

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

15.5 Python Solution (Memoization)

```
def unique_paths_with_obstacles_memo(obstacleGrid):
    if not obstacleGrid or obstacleGrid[0][0] == 1:
        return 0
    m, n = len(obstacleGrid), len(obstacleGrid[0])
    memo = {}
    def dp(i, j):
        if i < 0 or j < 0 or obstacleGrid[i][j] ==
           1:
            return 0
        if i == 0 and j == 0:
            return 1
        if (i, j) in memo:
            return memo[(i, j)]
        memo[(i, j)] = dp(i-1, j) + dp(i, j-1)
        return memo[(i, j)]
    return dp(m-1, n-1)
```

15.6 Python Solution (Tabulation)

```
def unique_paths_with_obstacles(obstacleGrid):
    if not obstacleGrid or obstacleGrid[0][0] == 1:
```

```
        return 0
    m, n = len(obstacleGrid), len(obstacleGrid[0])
    dp = [[0] * n for _ in range(m)]
    dp[0][0] = 1
    for i in range(m):
        for j in range(n):
            if obstacleGrid[i][j] == 1:
                continue
            if i > 0:
                dp[i][j] += dp[i-1][j]
            if j > 0:
                dp[i][j] += dp[i][j-1]
    return dp[m-1][n-1]
```

16 Minimum Path Sum

16.1 Problem Statement

Given an m x n grid of non-negative numbers, find minimum path sum from top-left to bottom-right.

16.2 Dry Run on Test Cases

- **Test Case 1**: grid = [[1,3,1],[1,5,1],[4,2,1]] → Output: 7 (1->3->1->1->1)
- **Test Case 2**: grid = [[1,2],[3,4]] → Output: 7
- **Test Case 3**: grid = [[1]] → Output: 1
- **Test Case 4**: grid = [] → Output: 0

16.3 Algorithm (Memoization)

1. Recurse with memoization for min sum to (i,j).
2. Base case: (0,0) returns grid[0][0], out of bounds returns inf.
3. Min sum = grid[i][j] + min(above, left).

16.4 Algorithm (Tabulation)

1. dp[i][j] = min sum to (i,j).
2. Initialize first row and column.
3. dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1]).

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

16.5 Python Solution (Memoization)

```python
def min_path_sum_memo(grid):
    if not grid:
        return 0
    m, n = len(grid), len(grid[0])
    memo = {}
    def dp(i, j):
        if i < 0 or j < 0:
            return float('inf')
        if i == 0 and j == 0:
            return grid[0][0]
        if (i, j) in memo:
            return memo[(i, j)]
        memo[(i, j)] = grid[i][j] + min(dp(i-1, j),
            dp(i, j-1))
        return memo[(i, j)]
    return dp(m-1, n-1)
```

16.6 Python Solution (Tabulation)

```python
def min_path_sum(grid):
    if not grid:
        return 0
    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]
    dp[0][0] = grid[0][0]
    for i in range(1, m):
        dp[i][0] = dp[i-1][0] + grid[i][0]
    for j in range(1, n):
        dp[0][j] = dp[0][j-1] + grid[0][j]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = grid[i][j] + min(dp[i-1][j],
                dp[i][j-1])
    return dp[m-1][n-1]
```

17 Decode Ways

17.1 Problem Statement

Given a string of digits, find number of ways to decode it into letters (1='A', 2='B', ..., 26='Z').

17.2 Dry Run on Test Cases

- **Test Case 1**: s = "12" → Output: 2 ("AB" or "L")
- **Test Case 2**: s = "226" → Output: 3 ("BBF", "VF", "BF")

- **Test Case 3**: s = "06" → Output: 0
- **Test Case 4**: s = "" → Output: 0

17.3 Algorithm (Memoization)

1. Recurse with memoization for ways from index i.
2. Base case: i=len(s) returns 1, invalid digit returns 0.
3. Add ways for single and double digit decodes.

17.4 Algorithm (Tabulation)

1. dp[i] = ways to decode prefix of length i.
2. Initialize dp[0]=1 if valid.
3. dp[i] += dp[i-1] if single digit valid, dp[i-2] if double digit valid.

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

17.5 Python Solution (Memoization)

```python
def num_decodings_memo(s):
    memo = {}
    def dp(i):
        if i == len(s):
            return 1
        if i in memo:
            return memo[i]
        if s[i] == '0':
            return 0
        result = dp(i + 1)
        if i + 1 < len(s) and (s[i] == '1' or (s[i]
            == '2' and s[i+1] <= '6')):
            result += dp(i + 2)
        memo[i] = result
        return result
    return 0 if not s or s[0] == '0' else dp(0)
```

17.6 Python Solution (Tabulation)

```python
def num_decodings(s):
    if not s or s[0] == '0':
        return 0
    dp = [0] * (len(s) + 1)
    dp[0] = 1
    dp[1] = 1
    for i in range(2, len(s) + 1):
        if s[i-1] != '0':
```

```
            dp[i] += dp[i-1]
        if s[i-2] == '1' or (s[i-2] == '2' and s[i
            -1] <= '6'):
            dp[i] += dp[i-2]
    return dp[len(s)]
```

18 Maximal Square

18.1 Problem Statement

Given a 2D binary matrix, find the size of the largest square of 1s.

18.2 Dry Run on Test Cases

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

18.3 Algorithm (Tabulation)

1. dp[i][j] = side length of max square ending at (i,j).
2. If matrix[i][j] = '1', dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1.
3. Track max side length.

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

18.4 Python Solution (Tabulation)

```
def maximal_square(matrix):
    if not matrix or not matrix[0]:
        return 0
    m, n = len(matrix), len(matrix[0])
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    max_side = 0
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if matrix[i-1][j-1] == '1':
                dp[i][j] = min(dp[i-1][j], dp[i][j
                    -1], dp[i-1][j-1]) + 1
                max_side = max(max_side, dp[i][j])
    return max_side * max_side
```

19 Palindromic Substrings

19.1 Problem Statement

Given a string, count all palindromic substrings.

19.2 Dry Run on Test Cases

- **Test Case 1**: s = "abc" → Output: 3 ("a","b","c")
- **Test Case 2**: s = "aaa" → Output: 6 ("a","a","a","aa","aa","aaa")
- **Test Case 3**: s = "" → Output: 0
- **Test Case 4**: s = "a" → Output: 1

19.3 Algorithm (Tabulation)

1. dp[i][j] = True if substring i to j is palindrome.
2. Single characters are palindromes.
3. Check pairs, then longer substrings using dp[i+1][j-1].

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

19.4 Python Solution (Tabulation)

```python
def count_substrings(s):
    n = len(s)
    dp = [[False] * n for _ in range(n)]
    count = 0

    # Single characters
    for i in range(n):
        dp[i][i] = True
        count += 1

    # Pairs
    for i in range(n-1):
        if s[i] == s[i+1]:
            dp[i][i+1] = True
            count += 1

    # Longer substrings
    for length in range(3, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j] and dp[i+1][j-1]:
                dp[i][j] = True
                count += 1
```

```
24
25      return count
```

20 Longest Palindromic Substring

20.1 Problem Statement

Given a string, find the longest palindromic substring.

20.2 Dry Run on Test Cases

- **Test Case 1**: s = "babad" → Output: "bab" or "aba"
- **Test Case 2**: s = "cbbd" → Output: "bb"
- **Test Case 3**: s = "" → Output: ""
- **Test Case 4**: s = "a" → Output: "a"

20.3 Algorithm (Tabulation)

1. dp[i][j] = True if substring i to j is palindrome.
2. Track longest palindrome with start index and length.
3. Check single, pairs, then longer substrings.

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

20.4 Python Solution (Tabulation)

```python
def longest_palindromic_substring(s):
    n = len(s)
    if n == 0:
        return ""
    dp = [[False] * n for _ in range(n)]
    start, max_len = 0, 1

    # Single characters
    for i in range(n):
        dp[i][i] = True

    # Pairs
    for i in range(n-1):
        if s[i] == s[i+1]:
            dp[i][i+1] = True
            start = i
            max_len = 2

    # Longer substrings
```

```
    for length in range(3, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j] and dp[i+1][j-1]:
                dp[i][j] = True
                if length > max_len:
                    start = i
                    max_len = length

    return s[start:start + max_len]
```

21 Number of 1 Bits

21.1 Problem Statement

Given an unsigned integer, return the number of 1 bits (Hamming weight).

21.2 Dry Run on Test Cases

- **Test Case 1**: n = 11 (1011) → Output: 3
- **Test Case 2**: n = 128 (10000000) → Output: 1
- **Test Case 3**: n = 0 → Output: 0
- **Test Case 4**: n = 4294967295 (111...1) → Output: 32

21.3 Algorithm

1. Use bitwise AND with 1 to check least significant bit.
2. Right shift number and count 1s.

Time Complexity: $O(1)$ (32 bits) **Space Complexity**: $O(1)$

21.4 Python Solution

```
def hamming_weight(n):
    count = 0
    while n:
        count += n & 1
        n >>= 1
    return count
```

22 Sum of Two Integers

22.1 Problem Statement

Given two integers, calculate their sum without using + or - operators.

22.2 Dry Run on Test Cases

- **Test Case 1**: a = 1, b = 2 → Output: 3
- **Test Case 2**: a = 2, b = 3 → Output: 5
- **Test Case 3**: a = 0, b = 0 → Output: 0
- **Test Case 4**: a = -2, b = 3 → Output: 1

22.3 Algorithm

1. Use XOR for sum without carry.
2. Use AND and left shift for carry.
3. Repeat until carry is 0, mask to 32 bits.

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

22.4 Python Solution

```python
def get_sum(a, b):
    mask = 0xffffffff
    while (b & mask) > 0:
        carry = ((a & b) << 1) & mask
        a = (a ^ b) & mask
        b = carry
    if (a >> 31) & 1:  # Handle negative numbers
        return ~(a ^ mask)
    return a
```

23 Single Number

23.1 Problem Statement

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

23.2 Dry Run on Test Cases

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

- **Test Case 4**: nums = [] → Output: 0

23.3 Algorithm

1. Use XOR of all numbers.
2. Paired numbers cancel out, leaving single number.

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

23.4 Python Solution

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

24 Missing Number

24.1 Problem Statement

Given an array of [0,n] with one number missing, find the missing number.

24.2 Dry Run on Test Cases

- **Test Case 1**: nums = [3,0,1] → Output: 2
- **Test Case 2**: nums = [0,1] → Output: 2
- **Test Case 3**: nums = [9,6,4,2,3,5,7,0,1] → Output: 8
- **Test Case 4**: nums = [0] → Output: 1

24.3 Algorithm

1. Use XOR of all indices and numbers.
2. XOR with n to account for array length.
3. Result is the missing number.

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

24.4 Python Solution

```
def missing_number(nums):
    result = len(nums)
    for i, num in enumerate(nums):
```

```
        result ^= i ^ num
    return result
```

25 Bitwise AND of Numbers Range

25.1 Problem Statement

Given a range [left, right], find the bitwise AND of all numbers.

25.2 Dry Run on Test Cases

- **Test Case 1**: left = 5, right = 7 → Output: 4 (5 6 7 = 4)
- **Test Case 2**: left = 0, right = 0 → Output: 0
- **Test Case 3**: left = 1, right = 2147483647 → Output: 0
- **Test Case 4**: left = 2, right = 3 → Output: 2

25.3 Algorithm

1. Shift left and right until they are equal.
2. Track shifts, shift result back.
3. Common prefix is the AND result.

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

25.4 Python Solution

```
def range_bitwise_and(left, right):
    shift = 0
    while left != right:
        left >>= 1
        right >>= 1
        shift += 1
    return left << shift
```

26 Counting Bits

26.1 Problem Statement

Given an integer n, return an array of number of 1 bits for numbers 0 to n.

26.2 Dry Run on Test Cases

- **Test Case 1**: n = 2 → Output: [0,1,1]
- **Test Case 2**: n = 5 → Output: [0,1,1,2,1,2]
- **Test Case 3**: n = 0 → Output: [0]
- **Test Case 4**: n = 1 → Output: [0,1]

26.3 Algorithm

1. For each i, use i (i-1) to remove least significant 1.
2. dp[i] = dp[i (i-1)] + 1.

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

26.4 Python Solution

```
def count_bits(n):
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        dp[i] = dp[i & (i-1)] + 1
    return dp
```

27 Reverse Bits

27.1 Problem Statement

Given a 32-bit unsigned integer, reverse its bits.

27.2 Dry Run on Test Cases

- **Test Case 1**: n = 43261596 (00000010100101000001111010011100) → Output: 964176192 (00111001011110000010100101000000)
- **Test Case 2**: n = 4294967295 (11111111111111111111111111111111) → Output: 4294967295
- **Test Case 3**: n = 0 → Output: 0
- **Test Case 4**: n = 1 → Output: 2147483648

27.3 Algorithm

1. Initialize result = 0.
2. For 32 bits, shift result left, add n's least significant bit, shift n right.

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

27.4 Python Solution

```python
def reverse_bits(n):
    result = 0
    for _ in range(32):
        result = (result << 1) | (n & 1)
        n >>= 1
    return result
```

28 Power of Two

28.1 Problem Statement

Given an integer, determine if it is a power of 2.

28.2 Dry Run on Test Cases

- **Test Case 1**: n = 1 → Output: True
- **Test Case 2**: n = 16 → Output: True
- **Test Case 3**: n = 3 → Output: False
- **Test Case 4**: n = 0 → Output: False

28.3 Algorithm

1. Check if n > 0 and n (n-1) == 0.
2. Powers of 2 have exactly one 1 bit.

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

28.4 Python Solution

```python
def is_power_of_two(n):
    return n > 0 and n & (n - 1) == 0
```

29 Power of Three

29.1 Problem Statement

Given an integer, determine if it is a power of 3.

29.2 Dry Run on Test Cases

- **Test Case 1**: n = 27 → Output: True
- **Test Case 2**: n = 0 → Output: False
- **Test Case 3**: n = 9 → Output: True
- **Test Case 4**: n = 45 → Output: False

29.3 Algorithm

1. Check if n > 0 and 3^19 **Time Complexity**: $O(1)$ **Space Complexity**: $O(1)$

29.4 Python Solution

```
def is_power_of_three(n):
    return n > 0 and 1162261467 % n == 0  # 3^19 = 1162261467
```

30 Power of Four

30.1 Problem Statement

Given an integer, determine if it is a power of 4.

30.2 Dry Run on Test Cases

- **Test Case 1**: n = 16 → Output: True
- **Test Case 2**: n = 5 → Output: False
- **Test Case 3**: n = 1 → Output: True
- **Test Case 4**: n = 0 → Output: False

30.3 Algorithm

1. Check if n > 0 and n (n-1) == 0 (power of 2).
2. Check if 1 bits are in odd positions (n 0x55555555 != 0).

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

30.4 Python Solution

```
def is_power_of_four(n):
    return n > 0 and n & (n - 1) == 0 and n & 0x55555555 != 0
```

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

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

```python
def permute(nums):
    result = []
    def backtrack(curr, used):
        if len(curr) == len(nums):
            result.append(curr[:])
            return
        for i in range(len(nums)):
            if not used[i]:
                used[i] = True
                curr.append(nums[i])
                backtrack(curr, used)
                curr.pop()
                used[i] = False
    backtrack([], [False] * len(nums))
    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**: nums = [1,1,2] → Output: [[1,1,2],[1,2,1],[2,1,1]]

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

```python
def permute_unique(nums):
    nums.sort()
    result = []
    def backtrack(curr, used):
        if len(curr) == len(nums):
            result.append(curr[:])
            return
        for i in range(len(nums)):
            if used[i] or (i > 0 and nums[i] == nums[i-1] and not
                used[i-1]):
                continue
            used[i] = True
            curr.append(nums[i])
            backtrack(curr, used)
            curr.pop()
            used[i] = False
    backtrack([], [False] * len(nums))
    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**: nums = [1,2,3] → Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
- **Test Case 2**: nums = [0] → Output: [[],[0]]
- **Test Case 3**: nums = [] → Output: [[]]

- **Test Case 4**: nums = [1,2] → 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

```python
def subsets(nums):
    result = []
    def backtrack(curr, i):
        result.append(curr[:])
        for j in range(i, len(nums)):
            curr.append(nums[j])
            backtrack(curr, j + 1)
            curr.pop()
    backtrack([], 0)
    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**: nums = [1,2,2] → Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]
- **Test Case 2**: nums = [0] → Output: [[],[0]]
- **Test Case 3**: nums = [] → Output: [[]]
- **Test Case 4**: nums = [1,1] → 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

```python
def subsets_with_dup(nums):
    nums.sort()
    result = []
    def backtrack(curr, i):
        result.append(curr[:])
        for j in range(i, len(nums)):
            if j > i and nums[j] == nums[j-1]:
                continue
            curr.append(nums[j])
            backtrack(curr, j + 1)
            curr.pop()
    backtrack([], 0)
    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

```python
def combination_sum(candidates, target):
    result = []
    def backtrack(curr, i, total):
        if total == target:
            result.append(curr[:])
            return
```

```
        if total > target:
            return
        for j in range(i, len(candidates)):
            curr.append(candidates[j])
            backtrack(curr, j, total + candidates[j])
            curr.pop()
    backtrack([], 0, 0)
    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

```
def combination_sum2(candidates, target):
    candidates.sort()
    result = []
    def backtrack(curr, i, total):
        if total == target:
            result.append(curr[:])
            return
        if total > target:
            return
        for j in range(i, len(candidates)):
```

```
            if j > i and candidates[j] ==
                candidates[j-1]:
                continue
            curr.append(candidates[j])
            backtrack(curr, j + 1, total +
                candidates[j])
            curr.pop()
    backtrack([], 0, 0)
    return result
```

7 N-Queens

7.1 Problem Statement

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

7.2 Dry Run on Test Cases

- **Test Case 1**: n = 4 → Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
- **Test Case 2**: n = 1 → Output: [["Q"]]
- **Test Case 3**: n = 2 → Output: []
- **Test Case 4**: n = 3 → 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

```
def solve_n_queens(n):
    def create_board(positions):
        board = [['.'] * n for _ in range(n)]
        for r, c in enumerate(positions):
            board[r][c] = 'Q'
        return [''.join(row) for row in board]

    def is_safe(row, col, queens):
        for r, c in enumerate(queens[:row]):
            if c == col or r - c == row - col or r
                + c == row + col:
```

```
                return False
        return True

    def backtrack(row, queens):
        if row == n:
            result.append(create_board(queens))
            return
        for col in range(n):
            if is_safe(row, col, queens):
                queens[row] = col
                backtrack(row + 1, queens)

    result = []
    backtrack(0, [0] * n)
    return result
```

8 N-Queens II

8.1 Problem Statement

Given an n x n chessboard, return the number of distinct N-Queens solutions.

8.2 Dry Run on Test Cases

- **Test Case 1**: n = 4 → Output: 2
- **Test Case 2**: n = 1 → Output: 1
- **Test Case 3**: n = 2 → Output: 0
- **Test Case 4**: n = 3 → 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

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

```
                return False
        return True

    def backtrack(row, queens):
        if row == n:
            return 1
        count = 0
        for col in range(n):
            if is_safe(row, col, queens):
                queens[row] = col
                count += backtrack(row + 1, queens)
        return count

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

```
def generate_parenthesis(n):
    result = []
    def backtrack(curr, open_count, close_count):
        if len(curr) == 2 * n:
            result.append(curr)
            return
```

```
        if open_count < n:
            backtrack(curr + '(', open_count + 1,
                close_count)
        if close_count < open_count:
            backtrack(curr + ')', open_count,
                close_count + 1)
    backtrack('', 0, 0)
    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

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

```
            return
        for c in mapping[digits[i]]:
            backtrack(curr + c, i + 1)
    backtrack('', 0)
    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

```
def exist(board, word):
    if not board or not board[0]:
        return False
    rows, cols = len(board), len(board[0])

    def backtrack(i, j, k):
        if k == len(word):
            return True
        if i < 0 or i >= rows or j < 0 or j >= cols
            or board[i][j] != word[k]:
            return False
        temp = board[i][j]
```

```
        board[i][j] = '#'
        result = (backtrack(i+1, j, k+1) or
           backtrack(i-1, j, k+1) or
                  backtrack(i, j+1, k+1) or
                     backtrack(i, j-1, k+1))
        board[i][j] = temp
        return result

    for i in range(rows):
        for j in range(cols):
            if backtrack(i, j, 0):
                return True
    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

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

```

```python
def find_words(board, words):
    if not board or not board[0]:
        return []

    # Build trie
    root = TrieNode()
    for word in words:
        node = root
        for c in word:
            if c not in node.children:
                node.children[c] = TrieNode()
            node = node.children[c]
        node.word = word

    rows, cols = len(board), len(board[0])
    result = []

    def backtrack(i, j, node):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or board[i][j] not in node.children:
            return
        c = board[i][j]
        node = node.children[c]
        if node.word:
            result.append(node.word)
            node.word = None
        temp = board[i][j]
        board[i][j] = '#'
        for di, dj in [(1,0), (-1,0), (0,1), (0,-1)
           ]:
            backtrack(i + di, j + dj, node)
        board[i][j] = temp

    for i in range(rows):
        for j in range(cols):
            backtrack(i, j, root)
    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","delete [[],[1],[3],[1,2],[1],[1],[1]] → 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

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

class MyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def get(self, index):
        if index < 0 or index >= self.size:
            return -1
        curr = self.head
        for _ in range(index):
            curr = curr.next
        return curr.val

    def addAtHead(self, val):
        self.head = ListNode(val, self.head)
        self.size += 1

    def addAtTail(self, val):
        if not self.head:
            self.head = ListNode(val)
        else:
            curr = self.head
            while curr.next:
                curr = curr.next
            curr.next = ListNode(val)
        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

```python
class ListNode:
    def __init__(self, key=-1, val=-1, next=None):
        self.key = key
        self.val = val
        self.next = next

class MyHashMap:
    def __init__(self):
        self.size = 1000
        self.buckets = [None] * self.size

    def _hash(self, key):
        return key % self.size

    def put(self, key, value):
        index = self._hash(key)
        if not self.buckets[index]:
            self.buckets[index] = ListNode(key,
               value)
        else:
            curr = self.buckets[index]
            while True:
                if curr.key == key:
                    curr.val = value
                    return
                if not curr.next:
                    break
                curr = curr.next
            curr.next = ListNode(key, value)

    def get(self, key):
        index = self._hash(key)
        curr = self.buckets[index]
        while curr:
            if curr.key == key:
                return curr.val
            curr = curr.next
        return -1

    def remove(self, key):
        index = self._hash(key)
        curr = self.buckets[index]
        if not curr:
            return
        if curr.key == key:
```

```
            self.buckets[index] = curr.next
            return
        while curr.next:
            if curr.next.key == key:
                curr.next = curr.next.next
                return
            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

```
from collections import deque

class MyStack:
    def __init__(self):
        self.q = deque()

    def push(self, x):
        self.q.append(x)
```

```
        for _ in range(len(self.q) - 1):
            self.q.append(self.q.popleft())

    def pop(self):
        return self.q.popleft()

    def top(self):
        return self.q[0]

    def empty(self):
        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

```
class MyQueue:
    def __init__(self):
        self.input = []
        self.output = []

    def push(self, x):
```

```
        self.input.append(x)

    def pop(self):
        self.peek()
        return self.output.pop()

    def peek(self):
        if not self.output:
            while self.input:
                self.output.append(self.input.pop()
                    )
        return self.output[-1]

    def empty(self):
        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","Rea [[3],[1],[2],[3],[4],[],[],[],[4],[]] → Output: [null,true,true,true,false,3,true,true,true,4]
- **Test Case 2**: ["MyCircularQueue","enQueue","Rear","Front"] [[1],[1],[],[]] → Output: [null,true,1,1]
- **Test Case 3**: ["MyCircularQueue","isEmpty"] [[1],[]] → Output: [null,true]
- **Test Case 4**: ["MyCircularQueue","enQueue","deQueue"] [[1],[1],[]] → 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

```python
class MyCircularQueue:
    def __init__(self, k):
        self.size = k
        self.queue = [None] * k
        self.front = -1  # Index of front element
        self.rear = -1   # Index of last element
        self.count = 0   # Number of elements

    def enQueue(self, value):
        if self.isFull():
            return False
        if self.isEmpty():
            self.front = 0
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = value
        self.count += 1
        return True

    def deQueue(self):
        if self.isEmpty():
            return False
        self.front = (self.front + 1) % self.size
        self.count -= 1
        if self.isEmpty():
            self.front = -1
            self.rear = -1
        return True

    def Front(self):
        return self.queue[self.front] if not self.
           isEmpty() else -1

    def Rear(self):
        return self.queue[self.rear] if not self.
           isEmpty() else -1

    def isEmpty(self):
        return self.count == 0

    def isFull(self):
        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

```python
class ListNode:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}
        self.head = ListNode()
        self.tail = ListNode()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_node(self, node):
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def _remove_node(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def get(self, key):
```

```
        if key in self.cache:
            node = self.cache[key]
            self._remove_node(node)
            self._add_node(node)
            return node.value
        return -1

    def put(self, key, value):
        if key in self.cache:
            self._remove_node(self.cache[key])
        node = ListNode(key, value)
        self._add_node(node)
        self.cache[key] = node
        if len(self.cache) > self.capacity:
            lru = self.tail.prev
            self._remove_node(lru)
            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

```python
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val):
        self.stack.append(val)
        if not self.min_stack or val <= self.
           min_stack[-1]:
            self.min_stack.append(val)

    def pop(self):
        if self.stack:
            val = self.stack.pop()
            if val == self.min_stack[-1]:
                self.min_stack.pop()

    def top(self):
        return self.stack[-1] if self.stack else
           None

    def getMin(self):
        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 = "abcabcbb" → Output: 3 ("abc")
- **Test Case 2**: s = "bbbbb" → Output: 1 ("b")
- **Test Case 3**: s = "pwwkew" → Output: 3 ("wke")
- **Test Case 4**: s = "" → 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

```
def length_of_longest_substring(s):
    char_index = {}
    max_len = 0
    start = 0
    for end, char in enumerate(s):
        if char in char_index and char_index[char]
           >= start:
            start = char_index[char] + 1
        char_index[char] = end
        max_len = max(max_len, end - start + 1)
    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 = "ABAB", k = 2 → Output: 4
- **Test Case 2**: s = "AABABBA", k = 1 → Output: 4
- **Test Case 3**: s = "", k = 1 → Output: 0
- **Test Case 4**: s = "AAAA", k = 0 → 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

```python
def character_replacement(s, k):
    freq = {}
    max_len = 0
    start = 0
    max_freq = 0
    for end, char in enumerate(s):
        freq[char] = freq.get(char, 0) + 1
        max_freq = max(max_freq, freq[char])
        if end - start + 1 - max_freq > k:
            freq[s[start]] -= 1
            start += 1
        max_len = max(max_len, end - start + 1)
    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 = "ADOBECODEBANC", t = "ABC" → Output: "BANC"
- **Test Case 2**: s = "a", t = "a" → Output: "a"
- **Test Case 3**: s = "a", t = "aa" → Output: ""
- **Test Case 4**: s = "", t = "a" → 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

```python
def min_window(s, t):
    if not s or not t:
        return ""
    t_count = {}
    for c in t:
```

```
        t_count[c] = t_count.get(c, 0) + 1
    required = len(t_count)
    formed = 0
    window_counts = {}

    start, min_len = 0, float('inf')
    min_window_substr = ""
    left = right = 0

    while right < len(s):
        c = s[right]
        window_counts[c] = window_counts.get(c, 0)
            + 1
        if c in t_count and window_counts[c] ==
            t_count[c]:
            formed += 1
        while left <= right and formed == required:
            c = s[left]
            if right - left + 1 < min_len:
                min_len = right - left + 1
                min_window_substr = s[left:right +
                    1]
            window_counts[c] -= 1
            if c in t_count and window_counts[c] <
                t_count[c]:
                formed -= 1
            left += 1
        right += 1
    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**: nums = [1,3,-1,-3,5,3,6,7], k = 3 → Output: [3,3,5,5,6,7]
- **Test Case 2**: nums = [1], k = 1 → Output: [1]
- **Test Case 3**: nums = [], k = 1 → Output: []
- **Test Case 4**: nums = [1,-1], k = 1 → 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

```python
from collections import deque

def max_sliding_window(nums, k):
    if not nums:
        return []
    result = []
    dq = deque()
    for i in range(len(nums)):
        while dq and dq[0] <= i - k:
            dq.popleft()
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()
        dq.append(i)
        if i >= k - 1:
            result.append(nums[dq[0]])
    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

```python
def merge(intervals):
    if not intervals:
        return []
    intervals.sort(key=lambda x: x[0])
    result = [intervals[0]]
    for start, end in intervals[1:]:
        if start <= result[-1][1]:
            result[-1][1] = max(result[-1][1], end)
        else:
            result.append([start, end])
    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

```python
def erase_overlap_intervals(intervals):
    if not intervals:
        return 0
    intervals.sort(key=lambda x: x[1])
    count = 0
    prev_end = intervals[0][1]
    for start, end in intervals[1:]:
        if start < prev_end:
```

```
            count += 1
        else:
            prev_end = end
    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

```
def insert(intervals, newInterval):
    result = []
    i = 0
    n = len(intervals)

    # Add non-overlapping intervals before
       newInterval
    while i < n and intervals[i][1] < newInterval
       [0]:
        result.append(intervals[i])
        i += 1

    # Merge overlapping intervals
```

```
    while i < n and intervals[i][0] <= newInterval
        [1]:
        newInterval[0] = min(newInterval[0],
            intervals[i][0])
        newInterval[1] = max(newInterval[1],
            intervals[i][1])
        i += 1
    result.append(newInterval)

    # Add remaining intervals
    while i < n:
        result.append(intervals[i])
        i += 1

    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

```
def can_attend_meetings(intervals):
    intervals.sort(key=lambda x: x[0])
    for i in range(1, len(intervals)):
        if intervals[i][0] < intervals[i-1][1]:
            return False
    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]] → Output: 2
- **Test Case 2**: intervals = [[7,10],[2,4]] → Output: 1
- **Test Case 3**: intervals = [] → Output: 0
- **Test Case 4**: intervals = [[1,5],[5,10]] → 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

```python
def min_meeting_rooms(intervals):
    if not intervals:
        return 0
    start = sorted(s for s, e in intervals)
    end = sorted(e for s, e in intervals)
    rooms = 0
    max_rooms = 0
    i = j = 0
    while i < len(intervals):
        if start[i] < end[j]:
            rooms += 1
            max_rooms = max(max_rooms, rooms)
            i += 1
        else:
            rooms -= 1
            j += 1
    return max_rooms
```

29 Spiral Matrix

29.1 Problem Statement

Given an m x 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]] → 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]] → Output: [1,2,3,4,8,12,11,10,9,5,6,7]
- **Test Case 3**: matrix = [] → Output: []
- **Test Case 4**: matrix = [[1]] → 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

```python
def spiral_order(matrix):
    if not matrix or not matrix[0]:
        return []
    result = []
    top, bottom = 0, len(matrix) - 1
    left, right = 0, len(matrix[0]) - 1

    while top <= bottom and left <= right:
        # Traverse right
        for j in range(left, right + 1):
            result.append(matrix[top][j])
        top += 1
        # Traverse down
        for i in range(top, bottom + 1):
            result.append(matrix[i][right])
        right -= 1
        if top <= bottom:
            # Traverse left
            for j in range(right, left - 1, -1):
                result.append(matrix[bottom][j])
            bottom -= 1
        if left <= right:
            # Traverse up
            for i in range(bottom, top - 1, -1):
                result.append(matrix[i][left])
```

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

Solutions to DSA Questions 171-200 (Matrix, Math, Geometry) For 1-2 Years Experience Roles at EPAM

Compiled on September 27, 2025

Introduction

This document provides detailed solutions for 30 Data Structures and Algorithms (DSA) problems (questions 171 to 200) from the Matrix, Math, and Geometry 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. Solutions are optimized for readability and efficiency, suitable for interview preparation.

Contents

1 Rotate Image

1.1 Problem Statement

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

1.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[1,2,3],[4,5,6],[7,8,9]] → Output: [[7,4,1],[8,5,2],[9,6,3]]
- **Test Case 2**: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]] → Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
- **Test Case 3**: matrix = [[1]] → Output: [[1]]
- **Test Case 4**: matrix = [] → Output: []

1.3 Algorithm

1. Transpose matrix (swap elements across diagonal).
2. Reverse each row.

Time Complexity: $O(n^2)$ **Space Complexity**: $O(1)$

1.4 Python Solution

```
def rotate(matrix):
    n = len(matrix)
    # Transpose
    for i in range(n):
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][
                j]
    # Reverse each row
    for i in range(n):
        matrix[i].reverse()
```

2 Set Matrix Zeroes

2.1 Problem Statement

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

2.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[1,1,1],[1,0,1],[1,1,1]] → Output: [[1,0,1],[0,0,0],[1,0,1]]
- **Test Case 2**: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]] → Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

- **Test Case 3**: matrix = [[1]] → Output: [[1]]
- **Test Case 4**: matrix = [] → Output: []

2.3 Algorithm

1. Use first row and column as markers.
2. Mark rows/columns to be zeroed.
3. Set zeros using markers, handle first row/column separately.

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

2.4 Python Solution

```python
def set_zeroes(matrix):
    if not matrix or not matrix[0]:
        return
    rows, cols = len(matrix), len(matrix[0])
    first_row_zero = any(matrix[0][j] == 0 for j in range(cols))
    first_col_zero = any(matrix[i][0] == 0 for i in range(rows))

    # Mark zeros in first row/column
    for i in range(1, rows):
        for j in range(1, cols):
            if matrix[i][j] == 0:
                matrix[i][0] = matrix[0][j] = 0

    # Set zeros based on markers
    for i in range(1, rows):
        if matrix[i][0] == 0:
            for j in range(1, cols):
                matrix[i][j] = 0
    for j in range(1, cols):
        if matrix[0][j] == 0:
            for i in range(1, rows):
                matrix[i][j] = 0

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

    # Handle first column
    if first_col_zero:
        for i in range(rows):
            matrix[i][0] = 0
```

3 Search a 2D Matrix

3.1 Problem Statement

Given a sorted m x n matrix (rows and first column sorted), search for a target.

3.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3 → Output: True
- **Test Case 2**: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13 → Output: False
- **Test Case 3**: matrix = [[1]], target = 1 → Output: True
- **Test Case 4**: matrix = [], target = 1 → Output: False

3.3 Algorithm

1. Start from top-right corner.
2. If target matches, return True.
3. If target smaller, move left; if larger, move down.

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

3.4 Python Solution

```python
def search_matrix(matrix, target):
    if not matrix or not matrix[0]:
        return False
    rows, cols = len(matrix), len(matrix[0])
    i, j = 0, cols - 1
    while i < rows and j >= 0:
        if matrix[i][j] == target:
            return True
        elif matrix[i][j] > target:
            j -= 1
        else:
            i += 1
    return False
```

4 Search a 2D Matrix II

4.1 Problem Statement

Given a sorted m x n matrix (rows and columns sorted), search for a target.

4.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 5 → Output: True
- **Test Case 2**: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 20 → Output: False
- **Test Case 3**: matrix = [[1]], target = 1 → Output: True
- **Test Case 4**: matrix = [], target = 1 → Output: False

4.3 Algorithm

1. Start from top-right corner.
2. If target matches, return True.
3. If target smaller, move left; if larger, move down.

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

4.4 Python Solution

```python
def search_matrix_ii(matrix, target):
    if not matrix or not matrix[0]:
        return False
    rows, cols = len(matrix), len(matrix[0])
    i, j = 0, cols - 1
    while i < rows and j >= 0:
        if matrix[i][j] == target:
            return True
        elif matrix[i][j] > target:
            j -= 1
        else:
            i += 1
    return False
```

5 Valid Sudoku

5.1 Problem Statement

Given a 9x9 Sudoku board, determine if it is valid (rows, columns, 3x3 sub-boxes).

5.2 Dry Run on Test Cases

– **Test Case 1**: board = [["5","3",".",".","7",".",".",".",".",".],["6",".",".","1","9","5",".",".",".",".],[".","9","8
→ Output: True

– **Test Case 2**: board = [["8","3",".",".","7",".",".",".",".","."],["6",".",".","1","9","5",".",".",".","."],[".","9","8
→ Output: False

– **Test Case 3**: board = [[".",".","."],[".",".","."],[".",".","."]] → Output: True

– **Test Case 4**: board = [] → Output: True

5.3 Algorithm

1. Use sets to track numbers in each row, column, and 3x3 box.
2. Check each cell; if number exists in respective set, return False.
3. Add valid numbers to sets.

Time Complexity: $O(1)$ (fixed 9x9) **Space Complexity**: $O(1)$

5.4 Python Solution

```python
def is_valid_sudoku(board):
    if not board or not board[0]:
        return True
    rows = [set() for _ in range(9)]
    cols = [set() for _ in range(9)]
    boxes = [set() for _ in range(9)]

    for i in range(9):
        for j in range(9):
            if board[i][j] == '.':
                continue
            num = board[i][j]
            if num in rows[i] or num in cols[j] or num in
                boxes[(i // 3) * 3 + j // 3]:
                return False
            rows[i].add(num)
            cols[j].add(num)
            boxes[(i // 3) * 3 + j // 3].add(num)
    return True
```

6 Factorial Trailing Zeroes

6.1 Problem Statement

Given an integer n, return the number of trailing zeros in n!.

6.2 Dry Run on Test Cases

* **Test Case 1**: n = 3 → Output: 0
* **Test Case 2**: n = 5 → Output: 1

* **Test Case 3**: n = 0 → Output: 0
* **Test Case 4**: n = 10 → Output: 2

6.3 Algorithm

1. Count factors of 5 in n! (since 2s are abundant).
2. Sum n//5, n//25, n//125, etc., until $n//5^k is 0$. **Time Complexity**: $O(\log n)$ **Space Complexity**: $O(1)$

6.4 Python Solution

```
def trailing_zeroes(n):
    count = 0
    while n > 0:
        n //= 5
        count += n
    return count
```

7 Power of N

7.1 Problem Statement

Given x and n, compute x^n.

7.2 Dry Run on Test Cases

- **Test Case 1**: x = 2.00000, n = 10 → Output: 1024.00000
- **Test Case 2**: x = 2.10000, n = 3 → Output: 9.26100
- **Test Case 3**: x = 2.00000, n = -2 → Output: 0.25000
- **Test Case 4**: x = 1.00000, n = 0 → Output: 1.00000

7.3 Algorithm

1. Use binary exponentiation.
2. If n is negative, compute $1/x^{|n|}.Squarebase, halveexponent, multiplyresultifexpone$ **Time Complexity**: $O(\log n)$ **Space Complexity**: $O(1)$

7.4 Python Solution

```
def my_pow(x, n):
    if n == 0:
        return 1.0
    if n < 0:
        x = 1 / x
```

```
        n = -n
    result = 1.0
    while n > 0:
        if n % 2 == 1:
            result *= x
        x *= x
        n //= 2
    return result
```

8 Sqrt(x)

8.1 Problem Statement

Given a non-negative integer x, return the square root of x (integer part).

8.2 Dry Run on Test Cases

3. **Test Case 1**: x = 4 → Output: 2

- **Test Case 2**: x = 8 → Output: 2
- **Test Case 3**: x = 0 → Output: 0
- **Test Case 4**: x = 1 → Output: 1

8.3 Algorithm

1. Use binary search to find largest integer whose square <= x.
2. Search range [0, x].
3. Adjust range based on mid * mid vs x.

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

8.4 Python Solution

```
def my_sqrt(x):
    if x == 0:
        return 0
    left, right = 1, x
    while left <= right:
        mid = (left + right) // 2
        if mid * mid == x:
            return mid
        elif mid * mid < x:
            left = mid + 1
        else:
            right = mid - 1
```

```
    return right
```

9 Divide Two Integers

9.1 Problem Statement

Given two integers dividend and divisor, return quotient without using * or /.

9.2 Dry Run on Test Cases

- **Test Case 1**: dividend = 10, divisor = 3 → Output: 3
- **Test Case 2**: dividend = 7, divisor = -3 → Output: -2
- **Test Case 3**: dividend = 0, divisor = 1 → Output: 0
- **Test Case 4**: dividend = 1, divisor = 1 → Output: 1

9.3 Algorithm

1. Handle signs and convert to positive numbers.
2. Use bit manipulation to subtract doubled divisors.
3. Handle 32-bit integer overflow.

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

9.4 Python Solution

```python
def divide(dividend, divisor):
    MAX_INT = 2**31 - 1
    MIN_INT = -2**31
    sign = -1 if (dividend < 0) ^ (divisor < 0)
       else 1
    dividend, divisor = abs(dividend), abs(divisor)

    quotient = 0
    while dividend >= divisor:
        temp, count = divisor, 1
        while dividend >= (temp << 1):
            temp <<= 1
            count <<= 1
        dividend -= temp
        quotient += count

    result = sign * quotient
    return min(max(result, MIN_INT), MAX_INT)
```

10 Fraction to Recurring Decimal

10.1 Problem Statement

Given numerator and denominator, return fraction as a string with recurring decimal.

10.2 Dry Run on Test Cases

- **Test Case 1**: numerator = 1, denominator = 2 → Output: "0.5"
- **Test Case 2**: numerator = 2, denominator = 1 → Output: "2"
- **Test Case 3**: numerator = 4, denominator = 333 → Output: "0.(012)"
- **Test Case 4**: numerator = -1, denominator = -2147483648 → Output: "0.0000000004656612873077392578125"

10.3 Algorithm

1. Handle sign and compute integer part.
2. For decimal part, use hashmap to detect repeating digits.
3. Add parentheses for repeating sequence.

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

10.4 Python Solution

```python
def fraction_to_decimal(numerator, denominator):
    if numerator == 0:
        return "0"
    result = []
    sign = -1 if (numerator < 0) ^ (denominator < 0) else 1
    numerator, denominator = abs(numerator), abs(denominator)

    # Integer part
    integer = numerator // denominator
    result.append(str(integer))

    # Decimal part
    remainder = numerator % denominator
    if remainder == 0:
        return ("-" if sign == -1 else "") + result[0]

    result.append(".")
    seen = {}
```

```
    while remainder:
        if remainder in seen:
            result.insert(seen[remainder], "(")
            result.append(")")
            break
        seen[remainder] = len(result)
        remainder *= 10
        result.append(str(remainder // denominator)
            )
        remainder %= denominator

    return ("-" if sign == -1 else "") + "".join(
        result)
```

11 Happy Number

11.1 Problem Statement

Given a number, determine if it is happy (sum of squares of digits leads to 1).

11.2 Dry Run on Test Cases

- **Test Case 1**: n = 19 → Output: True ($1^2 + 9^2 = 82, 8^2 + 2^2 = 68, 6^2 + 8^2 = 100, 1^2 + 0^2 + 0^2 = 1$)**Test Case 2** : $n = 2$→ Output: False (cycles)
- **Test Case 3**: n = 1 → Output: True
- **Test Case 4**: n = 7 → Output: True

11.3 Algorithm

1. Use set to detect cycles.
2. Compute sum of squares of digits.
3. Return True if sum is 1, False if cycle detected.

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

11.4 Python Solution

```
def is_happy(n):
    seen = set()
    while n != 1:
        if n in seen:
            return False
        seen.add(n)
        n = sum(int(d) ** 2 for d in str(n))
```

```
    return True
```

12 Plus One

12.1 Problem Statement

Given a non-negative integer as an array of digits, add one to it.

12.2 Dry Run on Test Cases

- **Test Case 1**: digits = [1,2,3] → Output: [1,2,4]
- **Test Case 2**: digits = [4,3,2,1] → Output: [4,3,2,2]
- **Test Case 3**: digits = [9] → Output: [1,0]
- **Test Case 4**: digits = [9,9] → Output: [1,0,0]

12.3 Algorithm

1. Iterate digits from right to left.
2. Add 1, handle carry.
3. If carry remains, prepend 1.

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

12.4 Python Solution

```
def plus_one(digits):
    n = len(digits)
    for i in range(n - 1, -1, -1):
        if digits[i] < 9:
            digits[i] += 1
            return digits
        digits[i] = 0
    return [1] + [0] * n
```

13 Climbing Stairs (Math Approach)

13.1 Problem Statement

Given n stairs, find number of ways to climb (1 or 2 steps) using a mathematical approach.

13.2 Dry Run on Test Cases

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

13.3 Algorithm

1. Recognize as Fibonacci sequence: ways(n) = ways(n-1) + ways(n-2).
2. Use iterative Fibonacci to avoid recursion.

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

13.4 Python Solution

```python
def climb_stairs(n):
    if n <= 1:
        return 1
    a, b = 1, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

14 Pow(x, n) (Alternative)

14.1 Problem Statement

Given x and n, compute $x^n using an alternative approach.$

14.2 Dry Run on Test Cases

- **Test Case 1**: x = 2.00000, n = 10 → Output: 1024.00000
- **Test Case 2**: x = 2.10000, n = 3 → Output: 9.26100
- **Test Case 3**: x = 2.00000, n = -2 → Output: 0.25000
- **Test Case 4**: x = 1.00000, n = 0 → Output: 1.00000

14.3 Algorithm

1. Use recursive binary exponentiation.
2. If n is odd, multiply by x; if even, square base.

3. Handle negative n by computing $1/x^{|n|}$. **Time Complexity**: $O(\log n)$ **Space Complexity**: $O(\log n)$

14.4 Python Solution

```python
def my_pow(x, n):
    def pow_positive(x, n):
        if n == 0:
            return 1.0
        half = pow_positive(x, n // 2)
        if n % 2 == 0:
            return half * half
        return half * half * x

    if n < 0:
        x = 1 / x
        n = -n
    return pow_positive(x, n)
```

15 Max Points on a Line

15.1 Problem Statement

Given a list of points, find the maximum number of points on a single line.

15.2 Dry Run on Test Cases

- **Test Case 1**: points = [[1,1],[2,2],[3,3]] → Output: 3
- **Test Case 2**: points = [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]] → Output: 4
- **Test Case 3**: points = [[1,1]] → Output: 1
- **Test Case 4**: points = [] → Output: 0

15.3 Algorithm

1. For each point, compute slopes with other points.
2. Use hashmap to count points with same slope.
3. Handle duplicates and vertical lines.

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

15.4 Python Solution

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

```

```python
def max_points(points):
    if not points:
        return 0
    max_points = 1
    for i in range(len(points)):
        slopes = defaultdict(int)
        duplicates = 0
        for j in range(i + 1, len(points)):
            if points[i] == points[j]:
                duplicates += 1
                continue
            dx = points[j][0] - points[i][0]
            dy = points[j][1] - points[i][1]
            if dx == 0:
                slope = 'inf'
            else:
                g = gcd(dx, dy)
                slope = (dy // g, dx // g)
            slopes[slope] += 1
        max_points = max(max_points, max(slopes.
           values(), default=0) + duplicates + 1)
    return max_points
```

16 Valid Number

16.1 Problem Statement

Given a string, determine if it is a valid number (integer, decimal, scientific).

16.2 Dry Run on Test Cases

- **Test Case 1**: s = "0" → Output: True
- **Test Case 2**: s = "e" → Output: False
- **Test Case 3**: s = "2e10" → Output: True
- **Test Case 4**: s = "abc" → Output: False

16.3 Algorithm

1. Split string on 'e' or 'E' for scientific notation.
2. Check if base and exponent (if present) are valid.
3. Base: optional sign, digits, optional decimal with digits.

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

16.4 Python Solution

```python
def is_number(s):
    def is_decimal(s):
        if not s:
            return False
        if s[0] in '+-':
            s = s[1:]
        if not s or s == '.':
            return False
        dot_seen = False
        for i, c in enumerate(s):
            if c == '.':
                if dot_seen:
                    return False
                dot_seen = True
            elif not c.isdigit():
                return False
        return True

    s = s.strip()
    if not s:
        return False
    parts = s.split('e') if 'e' in s else s.split('E')
    if len(parts) > 2:
        return False
    if len(parts) == 2:
        return is_decimal(parts[0]) and parts[1] and is_decimal(parts[1].replace('+', '-', 1))
    return is_decimal(parts[0])
```

17 Reverse Integer

17.1 Problem Statement

Given a 32-bit signed integer, reverse its digits.

17.2 Dry Run on Test Cases

- **Test Case 1**: x = 123 → Output: 321
- **Test Case 2**: x = -123 → Output: -321
- **Test Case 3**: x = 120 → Output: 21
- **Test Case 4**: x = 0 → Output: 0

17.3 Algorithm

1. Handle sign and convert to positive.
2. Reverse digits using modulo and division.
3. Check for 32-bit integer overflow.

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

17.4 Python Solution

```python
def reverse(x):
    MAX_INT = 2**31 - 1
    MIN_INT = -2**31
    sign = -1 if x < 0 else 1
    x = abs(x)
    result = 0
    while x:
        digit = x % 10
        if result > MAX_INT // 10 or (result ==
           MAX_INT // 10 and digit > MAX_INT % 10):
            return 0
        result = result * 10 + digit
        x //= 10
    return sign * result if MIN_INT <= sign *
       result <= MAX_INT else 0
```

18 Palindrome Number

18.1 Problem Statement

Given an integer, determine if it is a palindrome.

18.2 Dry Run on Test Cases

- **Test Case 1**: x = 121 → Output: True
- **Test Case 2**: x = -121 → Output: False
- **Test Case 3**: x = 10 → Output: False
- **Test Case 4**: x = 0 → Output: True

18.3 Algorithm

1. If negative, return False.
2. Reverse number and compare with original.

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

18.4 Python Solution

```python
def is_palindrome(x):
    if x < 0:
        return False
    original = x
    reversed_num = 0
    while x:
        reversed_num = reversed_num * 10 + x % 10
        x //= 10
    return original == reversed_num
```

19 Roman to Integer

19.1 Problem Statement

Given a Roman numeral string, convert it to an integer.

19.2 Dry Run on Test Cases

- **Test Case 1**: s = "III" → Output: 3
- **Test Case 2**: s = "IV" → Output: 4
- **Test Case 3**: s = "MCMXCIV" → Output: 1994
- **Test Case 4**: s = "LVIII" → Output: 58

19.3 Algorithm

1. Map Roman symbols to values.
2. Iterate string; if current value $<$ next, subtract, else add.

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

19.4 Python Solution

```python
def roman_to_int(s):
    roman = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C':
        100, 'D': 500, 'M': 1000}
    result = 0
    for i in range(len(s)):
        if i < len(s) - 1 and roman[s[i]] < roman[s
            [i + 1]]:
            result -= roman[s[i]]
        else:
            result += roman[s[i]]
    return result
```

20 Integer to Roman

20.1 Problem Statement

Given an integer, convert it to a Roman numeral string.

20.2 Dry Run on Test Cases

- **Test Case 1**: num = 3 → Output: "III"
- **Test Case 2**: num = 4 → Output: "IV"
- **Test Case 3**: num = 1994 → Output: "MCMXCIV"
- **Test Case 4**: num = 58 → Output: "LVIII"

20.3 Algorithm

1. Define Roman numeral values and symbols in descending order.
2. Greedily select largest possible value, append symbol, subtract.

Time Complexity: $O(1)$ (fixed range 1-3999) **Space Complexity**: $O(1)$

20.4 Python Solution

```python
def int_to_roman(num):
    values = [1000, 900, 500, 400, 100, 90, 50, 40,
        10, 9, 5, 4, 1]
    symbols = ["M", "CM", "D", "CD", "C", "XC", "L"
       , "XL", "X", "IX", "V", "IV", "I"]
    result = ""
    for v, s in zip(values, symbols):
        while num >= v:
            result += s
            num -= v
    return result
```

21 Count Primes

21.1 Problem Statement

Given an integer n, return the number of prime numbers less than n.

21.2 Dry Run on Test Cases

- **Test Case 1**: n = 10 → Output: 4 (2,3,5,7)
- **Test Case 2**: n = 0 → Output: 0

- **Test Case 3**: n = 1 → Output: 0
- **Test Case 4**: n = 2 → Output: 0

21.3 Algorithm

1. Use Sieve of Eratosthenes.
2. Mark multiples of each prime as non-prime.
3. Count remaining primes.

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

21.4 Python Solution

```python
def count_primes(n):
    if n < 2:
        return 0
    is_prime = [True] * n
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(n ** 0.5) + 1):
        if is_prime[i]:
            for j in range(i * i, n, i):
                is_prime[j] = False
    return sum(is_prime)
```

22 Ugly Number

22.1 Problem Statement

Given an integer, determine if it is an ugly number (prime factors only 2, 3, 5).

22.2 Dry Run on Test Cases

- **Test Case 1**: n = 6 → Output: True (2*3)
- **Test Case 2**: n = 8 → Output: True (2^3)**Test Case 3** : $n = 14$→ Output: False (2*7)
- **Test Case 4**: n = 1 → Output: True

22.3 Algorithm

1. Repeatedly divide n by 2, 3, 5 as long as possible.
2. If final n == 1, it is an ugly number.

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

22.4 Python Solution

```python
def is_ugly(n):
    if n <= 0:
        return False
    for prime in [2, 3, 5]:
        while n % prime == 0:
            n //= prime
    return n == 1
```

23 Ugly Number II

23.1 Problem Statement

Given an integer n, return the nth ugly number.

23.2 Dry Run on Test Cases

- **Test Case 1**: n = 10 → Output: 12 (1,2,3,4,5,6,8,9,10,12)
- **Test Case 2**: n = 1 → Output: 1
- **Test Case 3**: n = 7 → Output: 8
- **Test Case 4**: n = 4 → Output: 4

23.3 Algorithm

1. Use min-heap to generate ugly numbers.
2. Track seen numbers to avoid duplicates.
3. Pop n times to get nth ugly number.

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

23.4 Python Solution

```python
import heapq

def nth_ugly_number(n):
    if n == 1:
        return 1
    heap = [1]
    seen = {1}
    primes = [2, 3, 5]
    for _ in range(n - 1):
        curr = heapq.heappop(heap)
        for prime in primes:
            next_num = curr * prime
            if next_num not in seen:
```

```
                    seen.add(next_num)
                    heapq.heappush(heap, next_num)
    return heap[0]
```

24 Perfect Squares

24.1 Problem Statement

Given an integer n, return the least number of perfect squares that sum to n.

24.2 Dry Run on Test Cases

- **Test Case 1**: n = 12 → Output: 3 (4+4+4)
- **Test Case 2**: n = 13 → Output: 2 (9+4)
- **Test Case 3**: n = 1 → Output: 1
- **Test Case 4**: n = 4 → Output: 1

24.3 Algorithm

1. Use dynamic programming: dp[i] = min squares to sum to i.
2. For each i, try all perfect squares <= i, take minimum.

Time Complexity: $O(n \cdot \sqrt{n})$ **Space Complexity**: $O(n)$

24.4 Python Solution

```
def num_squares(n):
    dp = [float('inf')] * (n + 1)
    dp[0] = 0
    for i in range(1, n + 1):
        j = 1
        while j * j <= i:
            dp[i] = min(dp[i], dp[i - j * j] + 1)
            j += 1
    return dp[n]
```

25 Nth Digit

25.1 Problem Statement

Find the nth digit in the infinite sequence 1, 2, 3, ..., 10, 11, ...

25.2 Dry Run on Test Cases

- **Test Case 1**: n = 3 → Output: 3
- **Test Case 2**: n = 11 → Output: 0 (10)
- **Test Case 3**: n = 1 → Output: 1
- **Test Case 4**: n = 15 → Output: 1 (12)

25.3 Algorithm

1. Find range of numbers (1-digit, 2-digit, etc.) containing nth digit.
2. Compute which number and which digit within that number.

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

25.4 Python Solution

```python
def find_nth_digit(n):
    length = 0
    count = 9
    start = 1
    while n > length * count:
        n -= length * count
        length += 1
        count *= 10
        start *= 10
    start += (n - 1) // length
    return int(str(start)[(n - 1) % length])
```

26 Valid Perfect Square

26.1 Problem Statement

Given a positive integer, determine if it is a perfect square.

26.2 Dry Run on Test Cases

- **Test Case 1**: num = 16 → Output: True
- **Test Case 2**: num = 14 → Output: False
- **Test Case 3**: num = 1 → Output: True
- **Test Case 4**: num = 25 → Output: True

26.3 Algorithm

1. Use binary search to find if num = i * i.

2. Search range [1, num].

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

26.4 Python Solution

```python
def is_perfect_square(num):
    if num < 0:
        return False
    left, right = 0, num
    while left <= right:
        mid = (left + right) // 2
        square = mid * mid
        if square == num:
            return True
        elif square < num:
            left = mid + 1
        else:
            right = mid - 1
    return False
```

27 Arranging Coins

27.1 Problem Statement

Given n coins, find number of complete rows in a staircase (1, 2, 3, ... coins per row).

27.2 Dry Run on Test Cases

- **Test Case 1**: n = 5 → Output: 2 (1+2=3, 5-3=2)
- **Test Case 2**: n = 8 → Output: 3 (1+2+3=6, 8-6=2)
- **Test Case 3**: n = 0 → Output: 0
- **Test Case 4**: n = 1 → Output: 1

27.3 Algorithm

1. Use quadratic formula to solve k*(k+1)/2 <= n.
2. Return floor of solution.

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

27.4 Python Solution

```python
def arrange_coins(n):
    return int((-1 + (1 + 8 * n) ** 0.5) // 2)
```

28 Sum of Square Numbers

28.1 Problem Statement

Given a non-negative integer c, determine if it can be expressed as sum of two squares.

28.2 Dry Run on Test Cases

- **Test Case 1**: c = 5 → Output: True $(1^2 + 2^2)$**Test Case 2** : $c = 3$→ Output: False
- **Test Case 3**: c = 4 → Output: True $(2^2 + 0^2)$**Test Case 4** : $c = 0$→ Output: True

28.3 Algorithm

- 1. Use two pointers (i, j) where $i^2 + j^2 = c.Adjustpointersbasedonsumvsc.$ **Time Complexity**: $O(\sqrt{c})$ **Space Complexity**: $O(1)$

28.4 Python Solution

```python
def judge_square_sum(c):
    left, right = 0, int(c ** 0.5)
    while left <= right:
        curr_sum = left * left + right * right
        if curr_sum == c:
            return True
        elif curr_sum < c:
            left += 1
        else:
            right -= 1
    return False
```

29 Max Area of Island

29.1 Problem Statement

Given a binary grid, find the maximum area of an island (connected 1s).

29.2 Dry Run on Test Cases

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

29.3 Algorithm

1. Use DFS to explore each island.
2. Mark visited cells, count area.
3. Track maximum area across all islands.

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

29.4 Python Solution

```python
def max_area_of_island(grid):
    if not grid or not grid[0]:
        return 0
    rows, cols = len(grid), len(grid[0])

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or grid[i][j] != 1:
            return 0
        grid[i][j] = 0
        return 1 + dfs(i+1, j) + dfs(i-1, j) + dfs(
           i, j+1) + dfs(i, j-1)

    max_area = 0
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
                max_area = max(max_area, dfs(i, j))
    return max_area
```

30 Number of Islands

30.1 Problem Statement

Given a binary grid, count the number of islands (connected 1s).

30.2 Dry Run on Test Cases

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

30.3 Algorithm

1. Use DFS to mark all cells in an island as visited.
2. Count each new island encountered.

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

30.4 Python Solution

```python
def num_islands(grid):
    if not grid or not grid[0]:
        return 0
    rows, cols = len(grid), len(grid[0])

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or grid[i][j] != "1":
            return
        grid[i][j] = "0"
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    count = 0
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == "1":
                dfs(i, j)
                count += 1
    return count
```