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] $\rightarrow$ Largest = 9, Second Largest = 5, Output: 5
- **Test Case 2**: Input = [10, 10, 10] $\rightarrow$ Only one distinct element, Output: -1
- **Test Case 3**: Input = [5, 3] $\rightarrow$ Largest = 5, Second Largest = 3, Output: 3
- **Test Case 4**: Input = [-1, -5, -3] $\rightarrow$ 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: $k = k \% \text{len}(\text{arr})$.
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] $\rightarrow$ Max subarray [4, -1, 2, 1] = 6
- **Test Case 2**: Input = [1] $\rightarrow$ Output: 1
- **Test Case 3**: Input = [-1, -2, -3] $\rightarrow$ Output: -1

- **Test Case 4**: Input = [5, 4, -1, 7, 8] $\rightarrow$ 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 $\rightarrow$ arr1: [1, 2, 3, 4, 5, 6, 7]
- **Test Case 2**: arr1 = [1], m = 1; arr2 = [], n = 0 $\rightarrow$ arr1: [1]
- **Test Case 3**: arr1 = [0, 0], m = 0; arr2 = [2, 3], n = 2 $\rightarrow$ 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

```
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 ≤ 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):
```

```python
    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

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

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

```python
        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] $\rightarrow$ Water = 6
- **Test Case 2**: Input = [4, 2, 0, 3, 2, 5] $\rightarrow$ Water = 9
- **Test Case 3**: Input = [1, 2, 3] $\rightarrow$ Water = 0
- **Test Case 4**: Input = [0, 0] $\rightarrow$ 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] $\geq$ left_max, update left_max.
     - Else, add (left_max - arr[left]) to water.
     - left++
   - Else:
     - If arr[right] $\geq$ 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] $\rightarrow$ Buy at 1, sell at 6, Profit = 5
- **Test Case 2**: Input = [7, 6, 4, 3, 1] $\rightarrow$ No profit, Output: 0
- **Test Case 3**: Input = [1] $\rightarrow$ Output: 0
- **Test Case 4**: Input = [2, 4, 1] $\rightarrow$ 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] → Max area = 49
- **Test Case 2**: Input = [1, 1] → Area = 1
- **Test Case 3**: Input = [4, 3, 2, 1, 4] → Area = 16
- **Test Case 4**: Input = [1] → 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

```python
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]] $\rightarrow$ Max area = 2
- **Test Case 2**: Input = [[0,0,0],[0,0,0]] $\rightarrow$ Max area = 0
- **Test Case 3**: Input = [[1]] $\rightarrow$ Max area = 1
- **Test Case 4**: Input = [[1,1],[1,1]] $\rightarrow$ 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 $\rightarrow$ True
- **Test Case 2**: matrix = [[1,3],[2,4]], target = 5 $\rightarrow$ False
- **Test Case 3**: matrix = [[1]], target = 1 $\rightarrow$ True
- **Test Case 4**: matrix = [], target = 1 $\rightarrow$ 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]] $\rightarrow$ Output: [[1,6],[8,10],[15,18]]
- **Test Case 2**: Input = [[1,4],[4,5]] $\rightarrow$ Output: [[1,5]]
- **Test Case 3**: Input = [[1,4]] $\rightarrow$ Output: [[1,4]]
- **Test Case 4**: Input = [] $\rightarrow$ 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 → Output: 2 ([4,3])
- **Test Case 2**: arr = [1,4,4], target = 4 → Output: 1
- **Test Case 3**: arr = [1,1,1], target = 5 → Output: 0
- **Test Case 4**: arr = [1], target = 1 → 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]
```