

# Flipkart SDE Intern Prep



Congratulations on the interview! Let's make sure you're fully prepared. Flipkart typically focuses on these areas for SDE interviews:

## 1. DSA (Data Structures and Algorithms):

- **Key topics:**
  - Arrays, Strings, Linked Lists
  - Stack, Queue, and Deque
  - Trees (Binary Trees, BSTs)
  - Graphs (DFS, BFS, Dijkstra's Algorithm)
  - Dynamic Programming
  - Hashing and Hash Maps
  - Sorting and Searching
- **Practice Problems:**
  - Solve problems on platforms like LeetCode, Codeforces, and GeeksforGeeks.
  - Focus on medium-to-hard-level problems.

## 2. Core CS Fundamentals:

- **Operating Systems:**
  - Processes and Threads, Synchronization
  - Memory Management
- **DBMS:**
  - SQL queries, Normalization, ACID properties
- **OOPs:**
  - Polymorphism, Inheritance, Abstraction, and Encapsulation

## 3. System Design (for Intern Level):

- Basics of designing scalable systems (e.g., URL shortener, library management system)
- Understanding of APIs, load balancers, and database selection.

## 4. Behavioral Questions:

- STAR framework (Situation, Task, Action, Result) for describing projects and experiences.
- Be ready to discuss your Flipkart Grid experience.

## Action Plan:

1. **Daily DSA Practice:**
  - Solve 3-5 problems daily, covering different topics.
2. **Mock Interviews:**
  - Practice with peers or use platforms like Pramp, InterviewBit, or HackerRank for mocks.
3. **CS Concepts Review:**
  - Brush up on key fundamentals using notes or resources.
4. **System Design Basics:**

- Learn how to approach system design questions at a high level.

## 5. Behavioral Prep:

- Reflect on your experiences and prepare concise responses.

Let's start with a practice problem. Would you like to focus on DSA, system design, or behavioral questions first?



Let's focus on Data Structures and Algorithms! I'll start with a problem, and you can solve it step by step. If needed, I'll guide you through.

## Problem: Two Sum

### Description:

Given an array of integers `nums` and an integer `target`, return the indices of the two numbers such that they add up to the target.

### Example:

plaintext

```
Input: nums = [2, 7, 11, 15], target = 9
Output: [0, 1]
Explanation: Because nums[0] + nums[1] = 2 + 7 = 9, we return [0, 1].
```

### Constraints:

1. Each input will have **exactly one solution**.
2. You may not use the same element twice.
3. The order of the output indices does not matter.

### Task:

1. Solve the problem with a brute-force approach first.
2. Optimize your solution using a hash map for  $O(n)$  complexity.

Let me know your brute-force solution first, and we'll refine it!



Your implementation is a good attempt, but there are a few key issues to address:

### Observations:

#### 1. Duplicate Elements Issue:

- If there are duplicate elements in `nums`, `nums.index()` will always return the first occurrence of the element, which may not be correct.

## 2. Efficiency:

- Your solution is close to a brute-force approach and can still be optimized to  $O(n)$  using a hash map.

## Fixed Version (Brute-Force):

Here's how you can fix the duplicate issue in a brute-force manner:

python

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                if nums[i] + nums[j] == target:
                    return [i, j]
```

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

## Optimized Solution (Using Hash Map):

We can reduce the time complexity to  $O(n)$  by using a hash map to store the indices of the elements as we iterate through the array.

python

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        num_map = {} # Dictionary to store value-to-index mapping
        for i, num in enumerate(nums):
            complement = target - num
            if complement in num_map:
                return [num_map[complement], i]
            num_map[num] = i
```

## Explanation:

1. Use a dictionary (`num\_map`) to store the indices of numbers as we iterate.
2. For each number, calculate its complement (`target - num`).
3. Check if the complement exists in the dictionary:
  - If yes, return the indices.
  - If no, add the current number and its index to the dictionary.

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

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

## Test Cases:

You should test the function with these scenarios:

## 1. Standard Case:

python

```
nums = [2, 7, 11, 15]
target = 9
# Expected Output: [0, 1]
```

## 2. Negative Numbers:

python

```
nums = [-3, 4, 3, 90]
target = 0
# Expected Output: [0, 2]
```

## 3. Edge Case with Large Input:

python

```
nums = [3, 3]
target = 6
# Expected Output: [0, 1]
```

Would you like to proceed with testing, or should we dive into another problem?



Here's your original code fixed to handle the issues:

python

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        for i in range(len(nums)):
            complement = target - nums[i]
            # Ensure the complement exists and it's not the same index
            if complement in nums[i + 1:]:
                return [i, nums.index(complement, i + 1)]
```

## Changes Made:

### 1. Sliced Search for `index()`:

- To ensure we don't pick the same index twice, I adjusted `nums.index()` to search only in the subarray after the current index (`nums.index(complement, i + 1)`).

### 2. Improved Logic:

- Calculate the complement (`target - nums[i]`) for clarity and maintainability.

## Explanation:

1. **Outer Loop:** Iterates through each element in the array.
2. **Complement Check:** Checks if the complement exists in the subarray (`nums[i+1:]`) to ensure no index duplication.
3. **Index Lookup:** Uses `nums.index(complement, i + 1)` to find the correct index of the complement.

## Example:

For input:

```
python

nums = [3, 2, 4]
target = 6
```

Execution steps:

- At `i = 0`:
  - `complement = 6 - 3 = 3` (not found in `[2, 4]`).
- At `i = 1`:
  - `complement = 6 - 2 = 4` (found in `[4]` at index `2`).
- Returns `[1, 2]`.

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

(Still a brute-force solution since slicing creates overhead.)

If you'd like to further optimize this solution to  $O(n)$ , let me know!



Let's move to the next problem!

## Problem: Longest Substring Without Repeating Characters

### Description:

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

### Example:

```
plaintext

Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with a length of 3.
```

### Constraints:

- $0 \leq s.length \leq 5 \times 10^4$

- `s` consists of English letters, digits, symbols, and spaces.

## Approach:

### 1. Brute Force:

- Check all substrings and find the one with unique characters.
- **Time Complexity:**  $O(n^3)$  (inefficient for larger inputs).

### 2. Optimized Approach (Sliding Window + Hash Set):

- Use two pointers to create a sliding window.
- Maintain a set to store characters in the current window.
- Adjust the window to ensure all characters remain unique.
- **Time Complexity:**  $O(n)$ .

## Task:

1. Try writing a brute-force solution first.
2. If you're stuck, I'll guide you to optimize it.

Let me know your approach!



Here's the next problem for you:

## Problem: Merge Intervals

### Description:

Given an array of intervals where `intervals[i] = [start_i, end_i]`, merge all overlapping intervals and return an array of the non-overlapping intervals that cover all the intervals in the input.

### Example:

```
plaintext
```

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Intervals [1,3] and [2,6] overlap, so they are merged into [1,6].
```

### Constraints:

1.  $1 \leq \text{intervals.length} \leq 10^4$
2.  $\text{intervals}[i].\text{length} = 2$
3.  $0 \leq \text{start}_i \leq \text{end}_i \leq 10^4$

## Approach:

### 1. Sort Intervals:

- Sort the intervals by their starting points.

### 2. Iterate and Merge:

- Use a result array to track merged intervals.
- Compare the current interval with the last one in the result:
  - If they overlap, merge them.
  - If not, add the current interval to the result.

## Task:

1. Try implementing this problem in Python.
2. Let me know if you need help with the logic or code!



Here's how you can solve the **Merge Intervals** problem step-by-step:

## Steps to Solve:

### 1. Sort the Intervals:

- Sort the intervals by the starting value to ensure they are in order.

### 2. Merge Overlapping Intervals:

- Use a result list.
- Iterate through the sorted intervals:
  - If the current interval overlaps with the last interval in the result, merge them.
  - Otherwise, add the current interval to the result as a new interval.

## Python Code:

python

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        # Step 1: Sort intervals by starting time
        intervals.sort(key=lambda x: x[0])

        # Step 2: Initialize result list
        merged = []

        for interval in intervals:
            # If merged is empty or there's no overlap, add the interval
            if not merged or merged[-1][1] < interval[0]:
                merged.append(interval)
            else:
                # Overlap exists, merge intervals
                merged[-1][1] = max(merged[-1][1], interval[1])

        return merged
```

## Explanation:

### 1. Sorting:

```
python

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

- Sorts intervals by their start time for proper ordering.

### 2. Merging Logic:

- If the result list is empty, or the current interval does **not** overlap with the last interval in the result:

```
python

if not merged or merged[-1][1] < interval[0]:
    merged.append(interval)
```

- If there **is** an overlap, update the end of the last interval in the result:

```
python

merged[-1][1] = max(merged[-1][1], interval[1])
```

---

## Test Cases:

### 1. Example Input:

```
python

intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
print(Solution().merge(intervals)) # Output: [[1, 6], [8, 10], [15, 18]]
```

### 2. Edge Case - Single Interval:

```
python

intervals = [[1, 4]]
print(Solution().merge(intervals)) # Output: [[1, 4]]
```

### 3. Fully Overlapping Intervals:

```
python

intervals = [[1, 4], [2, 3], [3, 5]]
```



```
print(Solution().merge(intervals)) # Output: [[1, 5]]
```

#### 4. Non-Overlapping Intervals:

python

```
intervals = [[1, 2], [3, 4], [5, 6]]
print(Solution().merge(intervals)) # Output: [[1, 2], [3, 4], [5, 6]]
```

Let me know if you'd like to practice more problems or further explanation!



Here's the next problem for you:

### Problem: Find First and Last Position of Element in Sorted Array

#### Description:

Given an array of integers `nums` sorted in non-decreasing order, find the starting and ending position of a given `target` value.

If the target is not found in the array, return `[-1, -1]`.

#### Example:

plaintext

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```

#### Constraints:

1.  $0 \leq \text{nums.length} \leq 10^5$
2.  $-10^9 \leq \text{nums}[i], \text{target} \leq 10^9$
3. `nums` is sorted in non-decreasing order.

#### Approach:

##### 1. Binary Search:

- Use binary search to efficiently find the first occurrence of the target.
- Similarly, use binary search to find the last occurrence of the target.

##### 2. Time Complexity:

- Binary search runs in  $O(\log n)$  for each occurrence, making this solution  $O(\log n)$ .

## Python Code:

```
python
```

```
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        def findFirst(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] >= target:
                    right = mid - 1
                else:
                    left = mid + 1
            return left

        def findLast(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] <= target:
                    left = mid + 1
                else:
                    right = mid - 1
            return right

        # Find first and last occurrences
        first = findFirst(nums, target)
        last = findLast(nums, target)

        # Check if the target exists in nums
        if first <= last and first < len(nums) and nums[first] == target:
            return [first, last]
        return [-1, -1]
```

## Explanation:

### 1. findFirst Function:

- Performs binary search to locate the first occurrence of the target.
- If `nums[mid] >= target`, move `right` to `mid - 1` to search the left side.

### 2. findLast Function:

- Performs binary search to locate the last occurrence of the target.
- If `nums[mid] <= target`, move `left` to `mid + 1` to search the right side.

### 3. Final Check:

- Ensures `first` and `last` indices are valid and the target exists.

## Test Cases:

### 1. Standard Case:

```
python
```

```
nums = [5, 7, 7, 8, 8, 10]
target = 8
# Expected Output: [3, 4]
```

## 2. Target Not Found:

```
python

nums = [5, 7, 7, 8, 8, 10]
target = 6
# Expected Output: [-1, -1]
```

## 3. Edge Case - Empty Array:

```
python

nums = []
target = 0
# Expected Output: [-1, -1]
```

## 4. All Elements Match Target:

```
python

nums = [2, 2, 2, 2]
target = 2
# Expected Output: [0, 3]
```

Let me know if you want to tackle this or move to another problem!



Your code works for some cases but has a few issues and inefficiencies:

---

## Issues in Your Code:

### 1. Inefficient Target Search:

- Checking `if target not in nums` iterates through the entire list, which is  $O(n)$ . This is inefficient, especially when the input list is large.

### 2. Reverse List Search:

- `nums[::-1]` creates a reversed copy of the list, which is  $O(n)$  in both time and space.
- The `index` method on the reversed list needs adjustment to compute the correct index in the original list.

### 3. Edge Case Handling:

- You don't need to check `nums is None` since an empty list will handle itself with no matches.
- 

## Suggested Fix:

Instead of reversing the list or using linear searches, use **binary search** for both the first and last positions. This approach ensures optimal efficiency with  $O(\log n)$ .

---

## Fixed Code:

python

```
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        def findFirst(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] >= target:
                    right = mid - 1
                else:
                    left = mid + 1
            return left

        def findLast(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] <= target:
                    left = mid + 1
                else:
                    right = mid - 1
            return right

        # Perform binary searches
        first = findFirst(nums, target)
        last = findLast(nums, target)

        # Validate the result
        if first <= last and first < len(nums) and nums[first] == target:
            return [first, last]
        return [-1, -1]
```

## Key Points of Improvement:

### 1. Binary Search for First and Last Occurrences:

- Efficient and tailored to the sorted nature of the array.
- Avoids unnecessary list reversals or full iterations.

### 2. Validation Step:

- Ensures the computed indices are valid and the target actually exists in the array.

## Time Complexity:

- $O(\log n)$ , since each binary search runs in logarithmic time.

## Test the Fixed Code:

### 1. Example Input:

python

```
nums = [5, 7, 7, 8, 8, 10]
target = 8
```

```
print(Solution().searchRange(nums, target)) # Output: [3, 4]
```

## 2. Edge Case - Target Not Found:

python

```
nums = [1, 3, 5, 7]
target = 2
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

## 3. Edge Case - Empty List:

python

```
nums = []
target = 1
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

Let me know if you'd like more examples or further explanations!



To address your request for a function like `rightIndex` and improve the code structure, here's a clear implementation that uses separate functions to find the **left index** and **right index** of the target in the list:

## Python Code:

python

```
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        # Helper function to find the leftmost (first) index
        def leftIndex(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] < target:
                    left = mid + 1
                else:
                    right = mid - 1
            return left

        # Helper function to find the rightmost (last) index
        def rightIndex(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] <= target:
                    left = mid + 1
                else:
                    right = mid - 1
            return right

        # Perform binary searches
        left = leftIndex(nums, target)
        right = rightIndex(nums, target)
```

```
# Check if target exists
if left <= right and left < len(nums) and nums[left] == target:
    return [left, right]
return [-1, -1]
```

## Explanation:

### 1. leftIndex Function:

- Finds the smallest index where `nums[index] == target`.
- If `nums[mid] < target`, the search moves to the right half.
- Otherwise, the search moves to the left half



To use a **pre-made function** like `rightIndex`, I assume you are referring to a more structured approach where we use Python's built-in libraries or efficient algorithms.

We can implement this by leveraging `bisect\_left` and `bisect\_right` from Python's `bisect` module to find the first and last occurrences of the target efficiently.

- `bisect\_left(nums, target)` gives the leftmost position where `target` can be inserted while maintaining the sorted order (effectively the first occurrence).
- `bisect\_right(nums, target)` gives the rightmost position where `target` can be inserted, which corresponds to the position just beyond the last occurrence of the `target`.

## Python Code Using `bisect` Module:

python

```
import bisect

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        # Find the leftmost index (first occurrence)
        left = bisect.bisect_left(nums, target)

        # Find the rightmost index (last occurrence)
        right = bisect.bisect_right(nums, target) - 1

        # Check if the target exists in the array
        if left <= right and left < len(nums) and nums[left] == target:
            return [left, right]
        return [-1, -1]
```

## Explanation:

### 1. `bisect\_left(nums, target)`:

- Returns the index where `target` should be inserted to maintain sorted order (first occurrence of `target`).

### 2. `bisect\_right(nums, target)`:

- Returns the index where `target` should be inserted to maintain sorted order, just beyond the last occurrence. Subtracting 1 gives the last occurrence of `target`.

### 3. Validation:

- If the `target` is not present in the array (i.e., `nums[left] != target`), return `[-1, -1]`.

## Time Complexity:

- Both `bisect\_left` and `bisect\_right` run in  $O(\log n)$ , so the overall time complexity is  $O(\log n)$ .

## Test Cases:

### 1. Example Input:

```
python

nums = [5, 7, 7, 8, 8, 10]
target = 8
print(Solution().searchRange(nums, target)) # Output: [3, 4]
```

### 2. Target Not Found:

```
python

nums = [1, 3, 5, 7]
target = 2
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

### 3. Edge Case - Empty List:

```
python

nums = []
target = 1
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

### 4. All Elements Match Target:

```
python

nums = [2, 2, 2, 2]
target = 2
print(Solution().searchRange(nums, target)) # Output: [0, 3]
```

This solution utilizes the pre-made `bisect` functions to efficiently solve the problem. Let me know if you need further clarifications or additional practice problems!



Your solution has some issues that need to be addressed for it to work correctly and efficiently:

## Issues:

1. **Inefficient Check for Target:** The ``if target not in nums`` check will take  $O(n)$ , which is inefficient especially for large inputs.
2. **Reversing the List:** Reversing the list with ``nums[::-1]`` is not optimal and introduces an unnecessary  $O(n)$  complexity.
3. **``rindex`` is not a built-in method:** There is no ``rindex`` method for lists in Python. You could use ``nums[::-1].index(target)`` to find the index from the reversed list, but this is inefficient.

## Optimized Solution:

Instead of reversing the list, we can use binary search to find the first and last occurrence of the target in a more efficient manner. Here's how you can do it:

1. **Binary Search for First Occurrence:** Use a binary search to find the first occurrence of the target.
2. **Binary Search for Last Occurrence:** Use binary search to find the last occurrence of the target.

## Solution Using Binary Search:

python

```
import bisect

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        # Helper function to find the leftmost (first) index
        def leftIndex(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] < target:
                    left = mid + 1
                else:
                    right = mid - 1
            return left

        # Helper function to find the rightmost (last) index
        def rightIndex(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] <= target:
                    left = mid + 1
                else:
                    right = mid - 1
            return right

        # Perform binary searches
        left = leftIndex(nums, target)
        right = rightIndex(nums, target)

        # Check if target exists
        if left <= right and left < len(nums) and nums[left] == target:
            return [left, right]
        return [-1, -1]
```

## Explanation:

### 1. Binary Search Functions:

- ``leftIndex(nums, target)``: Finds the index of the first occurrence of the target.



- `rightIndex(nums, target)`: Finds the index of the last occurrence of the target.

## 2. Time Complexity:

- Both `leftIndex` and `rightIndex` are binary searches, so each runs in  $O(\log n)$ , making the overall time complexity  $O(\log n)$ .

## Test Cases:

### 1. Example Input:

```
python

nums = [5, 7, 7, 8, 8, 10]
target = 8
print(Solution().searchRange(nums, target)) # Output: [3, 4]
```

### 2. Target Not Found:

```
python

nums = [1, 3, 5, 7]
target = 2
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

### 3. Edge Case - Empty List:

```
python

nums = []
target = 1
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

### 4. All Elements Match Target:

```
python

nums = [2, 2, 2, 2]
target = 2
print(Solution().searchRange(nums, target)) # Output: [0, 3]
```

This solution is efficient and works in  $O(\log n)$  time complexity, making it suitable for large arrays. Let me know if you'd like further clarification or adjustments!



Here's another efficient solution using `bisect_left` and `bisect_right` from Python's `bisect` module, which can simplify the implementation for finding the left and right indices of the target.

## Optimized Solution Using `bisect`:

python

```
import bisect
from typing import List

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        # Find the leftmost index (first occurrence)
        left = bisect.bisect_left(nums, target)
        # Find the rightmost index (last occurrence)
        right = bisect.bisect_right(nums, target) - 1

        # Check if the target exists in the array
        if left <= right and left < len(nums) and nums[left] == target:
            return [left, right]
        return [-1, -1]
```

## Explanation:

1. `bisect_left(nums, target)`:
  - Finds the smallest index where `target` can be inserted to maintain sorted order. This corresponds to the first occurrence of the target.
2. `bisect_right(nums, target)`:
  - Finds the smallest index where `target + 1` can be inserted to maintain sorted order. Subtracting 1 gives the last occurrence of the target.
3. **Validation:**
  - Check if the target exists by ensuring `left <= right` and `nums[left] == target`.

## Time Complexity:

- $O(\log n)$ : Both `bisect_left` and `bisect_right` use binary search, making the solution very efficient.

## Test Cases:

### 1. Target Exists:

python

```
nums = [5, 7, 7, 8, 8, 10]
target = 8
print(Solution().searchRange(nums, target)) # Output: [3, 4]
```

### 2. Target Does Not Exist:

python

```
nums = [5, 7, 7, 8, 8, 10]
target = 6
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

### 3. Empty List:

```
python

nums = []
target = 1
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

### 4. Single Element List:

```
python

nums = [5]
target = 5
print(Solution().searchRange(nums, target)) # Output: [0, 0]
```

### 5. All Elements Match Target:

```
python

nums = [2, 2, 2, 2, 2]
target = 2
print(Solution().searchRange(nums, target)) # Output: [0, 4]
```

### 6. Target Greater Than All Elements:

```
python

nums = [1, 2, 3, 4, 5]
target = 10
print(Solution().searchRange(nums, target)) # Output: [-1, -1]
```

This approach is clean and leverages Python's powerful `bisect` module for maximum efficiency. Let me know if you have more scenarios to test or questions!



Sure! Here's a classic **binary search** problem for practice:

## Problem: Find Peak Element

### Problem Statement:

A peak element is an element that is strictly greater than its neighbors.

Given an integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index of **any of the peaks**.

You may assume that `nums[-1] = -∞` and `nums[n] = -∞`.

## Example:

**Input:** `nums = [1, 2, 3, 1]`

**Output:** `2`

**Explanation:** `nums[2] = 3` is a peak element.

**Input:** `nums = [1, 2, 1, 3, 5, 6, 4]`

**Output:** `5`

**Explanation:** `nums[5] = 6` is a peak element.

## Constraints:

1.  $1 \leq \text{nums.length} \leq 10^4$
2.  $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

## Follow-Up:

Can you implement a solution with  $O(\log n)$  complexity?

Would you like to try this problem yourself, or should I guide you through the solution? 😊



Your approach to maintaining a variable to store the maximum value and its index is a valid **linear search** solution. It works in  $O(n)$  time complexity, which is acceptable for small arrays. Here's how you can implement it:

## Linear Search Solution:

```
python

from typing import List

class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        max_value = float('-inf') # Initialize to negative infinity
        max_index = -1 # Initialize index

        # Iterate through the array
        for i in range(len(nums)):
            if nums[i] > max_value: # Check if the current element is greater than the max
                max_value = nums[i] # Update max value
                max_index = i # Update max index

        return max_index # Return the index of the peak element
```

## Test Cases:

### 1. Single Peak:

```
python
```

```
nums = [1, 2, 3, 1]
print(Solution().findPeakElement(nums)) # Output: 2
```

## 2. Multiple Peaks:

```
python
```

```
nums = [1, 2, 1, 3, 5, 6, 4]
print(Solution().findPeakElement(nums)) # Output: 5
```

## 3. All Elements Equal:

```
python
```

```
nums = [3, 3, 3, 3]
print(Solution().findPeakElement(nums)) # Output: 0
```

## 4. Descending Array:

```
python
```

```
nums = [10, 8, 6, 4, 2]
print(Solution().findPeakElement(nums)) # Output: 0
```

---

## Optimization with Binary Search ( $O(\log n)$ ):

For larger arrays, we can solve the problem using **binary search** to achieve  $O(\log n)$ . Here's the optimized solution:

---

## Binary Search Solution:

```
python
```

```
class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1

        while left < right:
            mid = (left + right) // 2
            if nums[mid] > nums[mid + 1]: # Descending part
                right = mid
            else: # Ascending part
                left = mid + 1

        return left # `left` will point to the peak
```

---

## Explanation:

### 1. Divide and Conquer:

- Check if `nums[mid] > nums[mid + 1]`. If true, the peak is in the left half.
- Otherwise, the peak is in the right half.

### 2. Why It Works:

- At least one peak exists in any array (even at the edges).
- The binary search always moves towards a peak.

## Test Cases for Binary Search:

The binary search implementation should provide the same results as the linear search for all test cases, but it will be much faster for large arrays.

Let me know if you'd like to discuss or test further! 😊



Here's your next DSA question to practice:

## Problem: Rotate Array

### Problem Statement:

Given an array, rotate the array to the right by  $k$  steps, where  $k$  is non-negative.

### Examples:

**Input:** `nums = [1, 2, 3, 4, 5, 6, 7], k = 3`

**Output:** `[5, 6, 7, 1, 2, 3, 4]`

**Explanation:** Rotate right by 3 steps:

`[1, 2, 3, 4, 5, 6, 7] → [5, 6, 7, 1, 2, 3, 4]`

**Input:** `nums = [-1, -100, 3, 99], k = 2`

**Output:** `[3, 99, -1, -100]`

**Explanation:** Rotate right by 2 steps:

`[-1, -100, 3, 99] → [3, 99, -1, -100]`

### Constraints:

1.  $1 \leq \text{nums.length} \leq 10^5$
2.  $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
3.  $0 \leq k \leq 10^5$

### Follow-Up:

Try to solve the problem in  $O(n)$  time complexity using  $O(1)$  extra space.

Would you like to attempt this problem, or should I guide you through different solutions? 😊



The expression `list(zip(matrix[::-1]))` does not work as intended because `zip(matrix)` is not a valid operation. The `zip` function requires multiple iterable arguments, and passing a single iterable (like `matrix`) results in a `TypeError`.

If you're trying to **rotate a 2D matrix clockwise by 90 degrees**, the correct usage involves the following steps:

## Correct Code for 90-Degree Clockwise Rotation:

python

```
def rotate_matrix_90(matrix):
    return list(zip(*matrix[::-1]))
```

## Explanation:

### 1. Reverse the Matrix Rows:

- `matrix[::-1]`: Reverses the order of rows in the matrix.

### 2. Transpose the Matrix:

- `zip(*matrix)`: Transposes the rows and columns of the matrix. The `*` operator unpacks the rows into arguments for `zip`.

### 3. Convert Back to List:

- `list(zip(...))`: Converts the result from `zip` (which is a tuple of tuples) into a list of tuples.

## Example:

### Input Matrix:

python

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

### Rotate Clockwise by 90:

python

```
rotated_matrix = rotate_matrix_90(matrix)
```

```
print(rotated_matrix)
```

## Output:

```
python

[
    (7, 4, 1),
    (8, 5, 2),
    (9, 6, 3)
]
```

If you'd like the output as a **list of lists** instead of tuples:

```
python

rotated_matrix = [list(row) for row in zip(*matrix[::-1])]
print(rotated_matrix)
```

## Output:

```
python

[
    [7, 4, 1],
    [8, 5, 2],
    [9, 6, 3]
]
```

Would you like me to walk you through additional matrix manipulation techniques? 😊



Great! Let's dive into some common **matrix manipulation techniques** that are frequently used in programming interviews and competitive programming. These techniques include:

## 1. Transposing a Matrix

**Transpose:** Convert rows into columns and columns into rows.

### Code:

```
python

def transpose(matrix):
    return [list(row) for row in zip(*matrix)]
```



## Example:

```
python

matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

transposed = transpose(matrix)
print(transposed)
```

## Output:

```
python

[
    [1, 4, 7],
    [2, 5, 8],
    [3, 6, 9]
]
```

## 2. Rotate a Matrix by 90 Degrees

### Clockwise Rotation:

```
python

def rotate_clockwise(matrix):
    return [list(row) for row in zip(*matrix[::-1])]
```

### Counterclockwise Rotation:

```
python

def rotate_counterclockwise(matrix):
    return [list(row) for row in zip(*matrix)][::-1]
```

## 3. Flip a Matrix Horizontally

**Horizontal Flip:** Reverse each row.

### Code:

```
python
```

```
def flip_horizontal(matrix):
    return [row[::-1] for row in matrix]
```

### Example:

```
python
```

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

flipped = flip_horizontal(matrix)
print(flipped)
```

### Output:

```
python
```

```
[
    [3, 2, 1],
    [6, 5, 4],
    [9, 8, 7]
]
```

## 4. Flip a Matrix Vertically

**Vertical Flip:** Reverse the order of rows.

### Code:

```
python
```

```
def flip_vertical(matrix):
    return matrix[::-1]
```

### Example:

```
python
```

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

```
flipped = flip_vertical(matrix)
print(flipped)
```

## Output:

python

```
[
    [7, 8, 9],
    [4, 5, 6],
    [1, 2, 3]
]
```

## 5. Spiral Order Traversal

**Spiral Traversal:** Traverse the matrix in a spiral order.

### Code:

python

```
def spiral_order(matrix):
    result = []
    while matrix:
        result += matrix.pop(0) # Take the first row
        if matrix and matrix[0]: # Check if the matrix is not empty
            for row in matrix:
                result.append(row.pop()) # Take the last element of each remaining row
        if matrix:
            result += matrix.pop()[::-1] # Take the last row in reverse order
        if matrix and matrix[0]:
            for row in matrix[::-1]:
                result.append(row.pop(0)) # Take the first element of each remaining row
    return result
```

### Example:

python

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print(spiral_order(matrix))
```

## Output:

```
python
```

```
[1, 2, 3, 6, 9, 8, 7, 4, 5]
```

## 6. Diagonal Traversal

**Diagonal Traversal:** Traverse the matrix diagonally from top-left to bottom-right.

**Code:**

```
python
```

```
def diagonal_traversal(matrix):
    result = []
    for d in range(len(matrix) + len(matrix[0]) - 1):
        diagonal = []
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if i + j == d:
                    diagonal.append(matrix[i][j])
        result.append(diagonal)
    return result
```

**Example:**

```
python
```

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print(diagonal_traversal(matrix))
```

**Output:**

```
python
```

```
[
    [1],
    [2, 4],
    [3, 5, 7],
    [6, 8],
    [9]
]
```

These are some common matrix manipulations. Let me know which ones you'd like to explore in more depth or if you want to solve related problems. 😊



Here's your next DSA question to practice:

---

## Problem: Set Matrix Zeroes

### Problem Statement:

Given an  $m \times n$  matrix, if an element is 0, set its entire row and column to 0. Do it in-place.

---

### Examples:

#### Input:

```
python

matrix = [
    [1, 1, 1],
    [1, 0, 1],
    [1, 1, 1]
]
```

#### Output:

```
python

[
    [1, 0, 1],
    [0, 0, 0],
    [1, 0, 1]
]
```

---

#### Input:

```
python

matrix = [
    [0, 1, 2, 0],
    [3, 4, 5, 2],
    [1, 3, 1, 5]
]
```

#### Output:

```
python

[
    [0, 0, 0, 0],
    [0, 4, 5, 0],
    [0, 3, 1, 0]
]
```

---

## Constraints:

1.  $m == \text{matrix.length}, n == \text{matrix}[0].\text{length}$
2.  $1 \leq m, n \leq 200$
3.  $-2^{31} \leq \text{matrix}[i][j] \leq 2^{31} - 1$

## Follow-Up:

1. Try to solve it in  $O(m \times n)$  time complexity.
2. Solve it using  $O(1)$  additional space.

Would you like to attempt it, or should I guide you through the solution? 😊



Your idea is valid but can be improved for efficiency. Let's implement it step by step.

## Approach

1. Use two sets (`marked\_row` and `marked\_col`) instead of lists to store the indices of rows and columns containing `0`. This ensures  $O(1)$  lookups when checking for marked indices.
2. Traverse the matrix and record the rows and columns that need to be set to zero.
3. Traverse the matrix again and update elements to `0` if their row or column index is in the `marked\_row` or `marked\_col`.

Here's the Python implementation:

## Code Implementation:

python

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        # Step 1: Initialize sets to store rows and columns to be zeroed
        marked_row = set()
        marked_col = set()

        # Step 2: Traverse the matrix and mark rows and columns
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if matrix[i][j] == 0:
                    marked_row.add(i)
                    marked_col.add(j)

        # Step 3: Traverse the matrix again and set elements to 0
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if i in marked_row or j in marked_col:
                    matrix[i][j] = 0
```

## Example Walkthrough:

## Input:

```
python

matrix = [
    [1, 1, 1],
    [1, 0, 1],
    [1, 1, 1]
]
```

## Execution:

1. Traverse the matrix:
  - Found `0` at `(1, 1)`, so:
    - `marked\_row = {1}`
    - `marked\_col = {1}`
2. Modify the matrix:
  - Row 1: All elements set to `0`.
  - Column 1: All elements set to `0`.

## Output:

```
python

[
    [1, 0, 1],
    [0, 0, 0],
    [1, 0, 1]
]
```

## Optimization for Space: $O(1)$ Approach

If you want to solve it with constant space:

1. Use the first row and column of the matrix itself as markers.
2. Modify the matrix based on those markers.

Would you like me to explain the  $O(1)$  approach as well? 😊



To solve the problem with  $O(1)$  **extra space** (i.e., without using any additional data structures like sets or lists for row and column markers), we can modify the input matrix itself to store the state of which rows and columns should be set to `0`.

## Approach:

1. **Use the first row and first column** of the matrix to act as markers to store which rows and columns should be set to `0`. Specifically:

- If a row needs to be set to `0`, we mark the first column of that row.
  - If a column needs to be set to `0`, we mark the first row of that column.
2. After marking, traverse the matrix from bottom-right to top-left to modify the elements based on the markers stored in the first row and first column.
  3. Finally, handle the special case of the first row and first column, which are used as markers.

## Steps:

### 1. Check the first row and column:

- If the first row or first column contains `0`, we need to track it, so we use two additional variables (`first\_row\_has\_zero` and `first\_col\_has\_zero`) to store if the first row or first column contains a zero.

### 2. Marking rows and columns:

- Traverse the matrix from `matrix[1][1]` to `matrix[m-1][n-1]`. If a `0` is found, mark the corresponding row and column by setting the first element of the respective row and column to `0`.

### 3. Modify the matrix:

- Traverse the matrix again. If the first element of a row or column is `0`, set the entire row or column to `0`.

### 4. Handle the first row and column:

- Based on the flags `first\_row\_has\_zero` and `first\_col\_has\_zero`, set the entire first row and/or first column to `0` if needed.

## Code Implementation:

python

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        # Step 1: Initialize flags for the first row and first column
        first_row_has_zero = any(matrix[0][j] == 0 for j in range(len(matrix[0])))
        first_col_has_zero = any(matrix[i][0] == 0 for i in range(len(matrix)))

        # Step 2: Use first row and first column as markers
        for i in range(1, len(matrix)):
            for j in range(1, len(matrix[0])):
                if matrix[i][j] == 0:
                    matrix[i][0] = 0 # Mark the start of the row
                    matrix[0][j] = 0 # Mark the start of the column

        # Step 3: Set elements to 0 based on the markers in the first row and column
        for i in range(1, len(matrix)):
            for j in range(1, len(matrix[0])):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0

        # Step 4: Set the first row to 0 if needed
        if first_row_has_zero:
            for j in range(len(matrix[0])):
                matrix[0][j] = 0

        # Step 5: Set the first column to 0 if needed
        if first_col_has_zero:
            for i in range(len(matrix)):
                matrix[i][0] = 0
```



## Explanation:

- Step 1:** We check if the first row or first column contains any `0`. These will be flagged by the variables `first\_row\_has\_zero` and `first\_col\_has\_zero`.
- Step 2:** We traverse the matrix from `matrix[1][1]` onward. If a `0` is found at `matrix[i][j]`, we mark the first cell of that row (`matrix[i][0]`) and the first cell of that column (`matrix[0][j]`) to `0`.
- Step 3:** We traverse the matrix again, and if any element in the first row or column is `0`, we set the corresponding element in the matrix to `0`.
- Step 4 and Step 5:** Finally, we handle the special case of the first row and first column based on the flags `first\_row\_has\_zero` and `first\_col\_has\_zero`.

## Example Walkthrough:

### Input:

```
python

matrix = [
    [1, 1, 1],
    [1, 0, 1],
    [1, 1, 1]
]
```

- Step 1:** The first row and column don't have `0`s, so `first\_row\_has\_zero = False` and `first\_col\_has\_zero = False`.
- Step 2:** Mark row 1 and column 1 as `0` because `matrix[1][1] == 0`.
- Step 3:** Set elements in the matrix based on the markers.
- Step 4:** The first row doesn't need to be set to `0`, so we skip.
- Step 5:** The first column doesn't need to be set to `0`, so we skip.

### Output:

```
python

[
    [1, 0, 1],
    [0, 0, 0],
    [1, 0, 1]
]
```

## Time and Space Complexity:

- Time Complexity:**  $O(m \times n)$ , where `m` is the number of rows and `n` is the number of columns.
- Space Complexity:**  $O(1)$  (ignoring the input matrix), as we are using only a few extra variables.

This approach minimizes the space usage while ensuring we can still achieve the result in  $O(m \times n)$  time. Would you like to try another problem or need clarification on this one? 😊



Here's your next DSA question to practice:

---

## Problem: Rotate Image (90 Degrees)

### Problem Statement:

You are given an  $n \times n$  2D matrix representing an image. Rotate the image by **90 degrees (clockwise)**, **in-place**.

---

### Example 1:

#### Input:

```
python

matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

#### Output:

```
python

[
    [7, 4, 1],
    [8, 5, 2],
    [9, 6, 3]
]
```

---

### Example 2:

#### Input:

```
python

matrix = [
    [5, 1, 9, 11],
    [2, 4, 8, 10],
    [13, 3, 6, 12],
    [14, 16, 7, 15]
]
```

#### Output:

python

```
[
    [14, 13, 2, 5],
    [16, 3, 4, 1],
    [7, 6, 8, 9],
    [15, 12, 10, 11]
]
```

### Constraints:

1.  $n == \text{matrix.length} == \text{matrix}[i].\text{length}$
2.  $1 \leq n \leq 20$
3. All elements of the matrix are integers.

### Steps for Rotation:

1. **Transpose** the matrix: Swap elements over the diagonal (i.e., `matrix[i][j]` with `matrix[j][i]`).
2. **Reverse** each row: Once transposed, reverse the order of elements in each row to achieve a 90-degree clockwise rotation.

Would you like to attempt it or need help with the solution? 😊



Yes, using `list(row) for row in zip(*matrix[::-1])` is a concise and elegant way to rotate a matrix 90 degrees clockwise in Python.

Here's an explanation of how it works:

### Explanation:

1. `matrix[::-1]`:
  - This reverses the rows of the matrix. In Python, slicing with `[::-1]` reverses the order of the list.
2. `zip(*matrix[::-1])`:
  - The `zip()` function takes multiple iterables and aggregates them into tuples, where each tuple contains elements from the same index across all input iterables.
  - By using `*matrix[::-1]`, we are unpacking the reversed matrix row by row. This effectively transposes the matrix while reversing the rows, which is equivalent to rotating the matrix 90 degrees clockwise.
3. `list(row) for row in zip(*matrix[::-1])`:
  - This constructs a new matrix by converting each tuple returned by `zip()` into a list. This gives us the final rotated matrix.

### Example:

Let's say we have the matrix:

python

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

- First, reverse the rows: `matrix[::-1]` results in:

python

```
[
    [7, 8, 9],
    [4, 5, 6],
    [1, 2, 3]
]
```

- Then, apply `zip(*matrix[::-1])` to transpose and reverse the rows:

python

```
zip(*matrix[::-1]) = [
    (7, 4, 1),
    (8, 5, 2),
    (9, 6, 3)
]
```

- Finally, convert the tuples into lists:

python

```
[
    [7, 4, 1],
    [8, 5, 2],
    [9, 6, 3]
]
```

This is the rotated matrix (90 degrees clockwise).

## Final Code:

python

```
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        matrix[:] = [list(row) for row in zip(*matrix[::-1])]
```

This solution rotates the matrix **in-place** by modifying the original matrix.

## Time and Space Complexity:

- **Time Complexity:**  $O(n^2)$ , as we traverse the entire matrix.
- **Space Complexity:**  $O(n^2)$ , because a new matrix is created during the transformation.

This approach is efficient for the problem at hand. Would you like to try another question or need further clarifications? 😊



Here's your next question to practice:

## Problem: Word Search

### Problem Statement:

Given an **m x n** grid of characters and a string `word`, return **true** if `word` exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where **adjacency** means cells that are horizontally or vertically neighboring. The same letter cell may not be used more than once.

### Example 1:

#### Input:

```
python

board = [
    ['A', 'B', 'C', 'E'],
    ['S', 'F', 'C', 'S'],
    ['A', 'D', 'E', 'E']
]
word = "ABCCED"
```

#### Output:

```
python

True
```

### Example 2:

#### Input:

```
python

board = [
    ['A', 'B', 'C', 'E'],
    ['S', 'F', 'C', 'S'],
    ['A', 'D', 'E', 'E']
]
word = "SEE"
```

## Output:

```
python
```

```
True
```

## Example 3:

### Input:

```
python
```

```
board = [
    ['A', 'B', 'C', 'E'],
    ['S', 'F', 'C', 'S'],
    ['A', 'D', 'E', 'E']
]
word = "ABCB"
```

## Output:

```
python
```

```
False
```

## Constraints:

1.  $m == \text{board.length}$
2.  $n == \text{board}[i].\text{length}$
3.  $1 \leq m, n \leq 6$
4.  $1 \leq \text{word.length} \leq 15$
5. **board** and **word** consist of lowercase and uppercase English letters.

## Approach:

### 1. Backtracking:

- Start from every cell on the board, and try to find the first letter of the word.
- If found, recursively check the adjacent cells for the next letter.
- Keep track of visited cells to avoid reusing the same cell.
- If we can construct the word, return **True**, otherwise continue with other paths.

### 2. Pruning:

- If at any point the current path does not match the expected character or the word cannot be formed, backtrack.

Would you like to try solving it or need further assistance with the approach? 😊



Your approach seems to suggest starting with the first letter and using flood fill to check for the remaining characters. Here's how you can implement this strategy effectively:

## Approach Breakdown:

### 1. Find First Letter:

- Iterate through each cell in the grid and check if the first letter of the word exists.

### 2. Flood Fill for Remaining Letters:

- Once the first letter is found, perform a flood fill or DFS (Depth First Search) to explore all possible adjacent cells (up, down, left, right) for the next letters in the word.
- If a letter is found, continue the search in adjacent cells.
- Mark cells as visited to prevent revisiting the same cell.

### 3. Backtrack if Necessary:

- If at any point the search fails to find the word, backtrack and try the next possible starting cell.

### 4. Repeat for All Starting Points:

- Repeat the process for every cell in the grid until the word is found or all possibilities are exhausted.

## Solution Code:

python

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        def dfs(i, j, index):
            # Base case: If the current index matches the length of the word, we've found the word
            if index == len(word):
                return True

            # Check boundaries and if the current cell matches the current character in word
            if i < 0 or j < 0 or i >= len(board) or j >= len(board[0]) or board[i][j] != word[index]:
                return False

            # Temporarily mark the cell as visited
            temp = board[i][j]
            board[i][j] = '#'

            # Explore all 4 possible directions (up, down, left, right)
            directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
            for di, dj in directions:
                ni, nj = i + di, j + dj
                if dfs(ni, nj, index + 1):
                    return True

            # Backtrack: restore the original value
            board[i][j] = temp
            return False

        # Iterate over every cell in the grid
        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] == word[0]: # First letter matches
                    if dfs(i, j, 0):
                        return True
        return False
```

## Explanation:

- **DFS (Depth-First Search)** is used to explore all possible paths starting from each cell that matches the first character of the word.
- **Base Case:** When `index == len(word)`, we have successfully found the word.
- **Boundary Check:** Ensure the next cell is within bounds and the character matches the current index in the word.
- **Visited Cells:** We mark a cell with `#` to indicate it's visited. After checking all adjacent cells, we backtrack and restore the original value.
- **Direction Array:** The four possible directions (right, down, left, up) are represented in the `directions` array and explored recursively.

## Time Complexity:

- **Worst Case:** In the worst case, we might need to check each cell multiple times, so the time complexity is  $O(m \times n \times 4^k)$ , where:
  - $m \times n$  is the number of cells in the grid.
  - $k$  is the length of the word.
- **Space Complexity:** The space complexity is  $O(k)$ , where  $k$  is the length of the word due to the recursive call stack.

---

Would you like to try implementing this, or do you need help understanding a part of the code? 😊



Here's your next DSA question to practice:

---

## Problem: Merge Intervals

### Problem Statement:

Given a collection of intervals, merge all overlapping intervals.

### Example 1:

#### Input:

```
python

intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
```

#### Output:

```
python

[[1, 6], [8, 10], [15, 18]]
```

## Explanation:



- The intervals [1, 3] and [2, 6] overlap, so they are merged into [1, 6].
- The other intervals [8, 10] and [15, 18] do not overlap and remain the same.

## Example 2:

### Input:

```
python

intervals = [[1, 4], [4, 5]]
```

### Output:

```
python

[[1, 5]]
```

### Constraints:

- $1 \leq \text{intervals.length} \leq 1000$
- $0 \leq \text{intervals}[i][0] \leq \text{intervals}[i][1] \leq 10^4$

## Approach:

1. **Sort the intervals** by the starting value of each interval.
2. **Merge** the intervals:
  - Initialize an empty list to hold the merged intervals.
  - Iterate through the sorted intervals:
    - If the current interval overlaps with the last merged interval, merge them.
    - If they do not overlap, simply add the current interval to the merged list.

Would you like to attempt solving it, or would you prefer me to help with the solution? 😊



Here's your next question:

## Problem: Spiral Matrix

### Problem Statement:

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

### Example 1:

### Input:

```
python
```

```
matrix = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]]
```

### Output:

```
python
```

```
[1, 2, 3, 6, 9, 8, 7, 4, 5]
```

### Example 2:

#### Input:

```
python
```

```
matrix = [[1, 2, 3, 4],
           [5, 6, 7, 8],
           [9, 10, 11, 12]]
```

### Output:

```
python
```

```
[1, 2, 3, 4, 8, 12, 11, 10, 9, 5, 6, 7]
```

### Constraints:

1.  $m == \text{matrix.length}$
2.  $n == \text{matrix}[i].\text{length}$
3.  $1 \leq m, n \leq 10$
4.  $-100 \leq \text{matrix}[i][j] \leq 100$

### Approach:

#### 1. Initialize Boundaries:

- Top, bottom, left, and right boundaries to track the current layer of the spiral.

#### 2. Traverse the Matrix in Spiral Order:

- Move from **left to right** across the top boundary.
- Move from **top to bottom** along the right boundary.
- Move from **right to left** along the bottom boundary.
- Move from **bottom to top** along the left boundary.

#### 3. Update Boundaries:

- Shrink the boundaries as you complete each layer of the spiral.

#### 4. Continue Until All Elements Are Visited:

- Stop once the boundaries overlap.

---

Would you like to solve this on your own, or should I provide a code implementation? 😊



Your solution is a clever and concise implementation to solve the **Spiral Matrix** problem by repeatedly processing the first row of the matrix and rotating the remaining part. Let me explain and refine it into a function:

---

### Solution Code:

python

```
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        result = []
        while matrix:
            # Add the first row to the result
            result += matrix.pop(0)
            # Rotate the remaining matrix counterclockwise
            matrix = list(zip(*matrix))[::-1]
        return result
```

### Explanation:

#### 1. Pop the First Row:

- The first row is always part of the spiral traversal. Add it to the result list using ``matrix.pop(0)``.

#### 2. Rotate Remaining Matrix:

- Use ``zip(*matrix)`` to transpose the matrix.
- Reverse the rows with ``[::-1]`` to simulate a counterclockwise rotation.

#### 3. Repeat Until Matrix is Empty:

- Continue the process until all rows are added to the result list.
- 

### Complexity Analysis:

- **Time Complexity:**  $O(m \times n)$ 
    - Each element in the matrix is processed exactly once.
  - **Space Complexity:**  $O(m \times n)$ 
    - The transposition and reversal operations create a new matrix.
- 

### Example Execution:

Input:

```
python
```

```
matrix = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]]
```

### Execution:

- **Step 1:** Add `[1, 2, 3]` (first row), rotate to:

```
lua
```

```
[[6, 9],
 [5, 8],
 [4, 7]]
```

- **Step 2:** Add `[6, 9]`, rotate to:

```
lua
```

```
[[8, 7],
 [5, 4]]
```

- **Step 3:** Add `[8, 7]`, rotate to:

```
lua
```

```
[[4],
 [5]]
```

- **Step 4:** Add `[4, 5]`, matrix is empty.

### Output:

```
python
```

```
[1, 2, 3, 6, 9, 8, 7, 4, 5]
```

Your approach works perfectly for this problem. Would you like to move on to a harder matrix-related question, or should we try a different topic? 😊



Here's your next matrix-related DSA question:

### Problem: Rotate Image (90 Degrees Clockwise)

## Problem Statement:

You are given an  $n \times n$  2D matrix representing an image. Rotate the image by **90 degrees clockwise**, in place.

## Example 1:

### Input:

```
python

matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

### Output:

```
python

[
    [7, 4, 1],
    [8, 5, 2],
    [9, 6, 3]
]
```

## Example 2:

### Input:

```
python

matrix = [
    [5, 1, 9, 11],
    [2, 4, 8, 10],
    [13, 3, 6, 7],
    [15, 14, 12, 16]
]
```

### Output:

```
python

[
    [15, 13, 2, 5],
    [14, 3, 4, 1],
    [12, 6, 8, 9],
    [16, 7, 10, 11]
]
```

## Constraints:

- $n == \text{matrix.length} == \text{matrix}[i].\text{length}$
- $(1 \leq$



Understood! Let's focus on questions that are relevant for **Flipkart's SDE Internship**, prioritizing **DSA**, **problem-solving**, and **system design basics**, as well as the key areas Flipkart evaluates: **scalability**, **optimization**, and **real-world problem-solving**.

Here's a structured preparation plan:

---

## DSA Topics and Questions

### Arrays and Strings

#### 1. Find First and Last Position of Element in Sorted Array

Problem: Given a sorted array, find the first and last index of a given number.

Topics: Binary Search, Two-Pointer Technique.

#### 2. Subarray with Given Sum

Problem: Given an array, find if there is a subarray with a sum equal to a target.

Topics: Sliding Window, HashMap.

#### 3. Rotate Image (90 Degrees)

Problem: Rotate an  $n \times n$  matrix 90 degrees clockwise in place.

Topics: Matrix Transformation.

#### 4. Longest Substring Without Repeating Characters

Problem: Find the longest substring in a string without repeating characters.

Topics: Sliding Window, HashSet.

---

### Linked Lists

#### 1. Reverse Nodes in k-Group

Problem: Reverse the nodes of a linked list in groups of  $k$ .

Topics: Recursion, Iterative Linked List Manipulation.

#### 2. Detect Cycle in a Linked List

Problem: Check if a linked list contains a cycle and return the start node of the cycle.

Topics: Floyd's Tortoise and Hare Algorithm.

#### 3. Merge k Sorted Lists

Problem: Merge  $k$  sorted linked lists into a single sorted linked list.

Topics: Divide and Conquer, Min-Heap.

---

### Trees and Graphs

#### 1. Lowest Common Ancestor in a Binary Tree

Problem: Find the lowest common ancestor of two nodes in a binary tree.

Topics: DFS, Recursion.

## 2. Word Ladder

Problem: Find the shortest transformation sequence from a start word to an end word, changing one letter at a time.

Topics: BFS, Backtracking.

## 3. Course Schedule

Problem: Given  $n$  courses and prerequisites, determine if all courses can be completed.

Topics: Topological Sort, Graphs.

---

## Dynamic Programming

### 1. Longest Increasing Subsequence

Problem: Find the length of the longest increasing subsequence in an array.

Topics: DP, Binary Search.

### 2. Knapsack Problem

Problem: Given a set of items with weights and values, find the maximum value achievable with a given weight limit.

Topics: DP, Optimization.

### 3. Edit Distance

Problem: Given two strings, find the minimum number of operations (insert, delete, replace) to convert one into the other.

Topics: DP, String Manipulation.

---

## Sorting and Searching

### 1. Search in Rotated Sorted Array

Problem: Find a target element in a rotated sorted array.

Topics: Binary Search.

### 2. Median of Two Sorted Arrays

Problem: Find the median of two sorted arrays in  $O(\log(m + n))$ .

Topics: Binary Search, Divide and Conquer.

### 3. Kth Largest Element in an Array

Problem: Find the  $k$ th largest element in an array.

Topics: Quickselect, Heap.

---

## System Design Basics

### 1. Design a URL Shortener

Problem: Design a system like TinyURL to shorten URLs and retrieve the original URL.

Topics: Hashing, Databases.

### 2. Design a Cache System

Problem: Implement an LRU (Least Recently Used) cache.

Topics: Linked HashMap, Optimization.

### 3. Design Flipkart's Search Bar

Problem: Implement a search bar that provides suggestions while typing.

Topics: Trie, Auto-Complete.

---

### Behavioral Questions

1. Explain a challenging project you worked on. How did you handle roadblocks?
  2. Describe a time when you optimized a solution. What was the impact?
  3. How do you prioritize tasks when working on multiple deadlines?
- 

Would you like detailed solutions or mock interview sessions for these topics? Let me know! 😊