Arrays: Comprehensive Guide for SDE Interviews

Arrays are one of the most fundamental data structures in computer science. A strong understanding of arrays is crucial for SDE interviews as they form the basis for many other complex algorithms and data structures.

1. Essential Topics to Know in Arrays

Before diving into problem patterns, ensure you have a solid grasp of these core array concepts:

- **Definition and Properties**: Understand what an array is (a collection of elements stored at contiguous memory locations), its fixed-size nature (in most static array implementations), and how elements are accessed using indices.
- **Basic Operations**:
  - **Access**: Retrieving an element at a specific index (O(1)).
  - **Insertion**: Adding an element (can be O(N) for shifting elements in the middle, O(1) at end of dynamic array).
  - **Deletion**: Removing an element (can be O(N) for shifting elements).
  - **Update**: Modifying an element at a specific index (O(1)).
  - **Traversal**: Iterating through all elements (O(N)).
- **Searching Algorithms**:
  - **Linear Search**: Searching for an element sequentially (O(N)).
  - **Binary Search**: Efficiently searching in sorted arrays (O(logN)).
- **Sorting Algorithms**: While you might not implement all of them from scratch in an interview, understand the core principles, time/space complexities, and best use cases for common algorithms like:
  - Bubble Sort, Selection Sort, Insertion Sort (O(N2) average)
  - Merge Sort, Quick Sort, Heap Sort (O(NlogN) average)
- **Multidimensional Arrays (Matrices)**: Understanding 2D and 3D arrays, their storage, and common traversal techniques (row-major, column-major, diagonal, spiral).
- **Dynamic Arrays (e.g., ArrayList in Java, Vector in C++, List in Python)**: How they handle resizing and manage memory dynamically.

- **Hashing with Arrays**: Using hash maps or frequency arrays for counting occurrences, checking presence, or quick lookups.
- **In-place Operations**: Performing modifications on an array without using significant extra space.
- **Edge Cases**: Handling empty arrays, single-element arrays, duplicate elements, sorted/reverse-sorted arrays, and large/small integer values.

2. Questions Categorized by Patterns

Here are the most common patterns encountered in array-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Two Pointers

This technique involves using two pointers that traverse the array (or two arrays) to solve problems efficiently. Pointers can move in the same direction, opposite directions, or at different speeds.

- **Easy**: **Reverse an Array In-place**
  - **Problem**: Given an array, reverse its elements in-place.
  - **Example**: Input: [1, 2, 3, 4, 5] -> Output: [5, 4, 3, 2, 1]
  - **Approach**: Use one pointer at the start and one at the end. Swap elements and move pointers towards the center until they meet or cross.
- **Medium**: **Find a Pair with a Given Sum in a Sorted Array**
  - **Problem**: Given a sorted array of integers and a target sum, find if there exists a pair of elements that sum up to the target. Return their indices or true/false.
  - **Example**: Input: arr = [2, 7, 11, 15], target = 9 -> Output: [0, 1] or true
  - **Approach**: Use two pointers, left at the beginning and right at the end. If arr[left] + arr[right] is less than target, increment left. If greater, decrement right. If equal, you found the pair.
- **Hard**: **3Sum**
  - **Problem**: Given an integer array nums, return all unique triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

    - **Example**: Input: nums = [-1, 0, 1, 2, -1, -4] -> Output: [[-1, -1, 2], [-1, 0, 1]]
    - **Approach**: Sort the array first. Iterate with one pointer (i). For each nums[i], use two pointers (left and right) on the remaining array to find pairs that sum to -nums[i], similar to the "Pair with a Given Sum" problem. Handle duplicates carefully.

Pattern 2: Sliding Window

This technique is used for problems that involve finding a subarray or substring that satisfies a certain condition. A "window" of elements "slides" over the array.

- **Easy**: **Maximum Sum Subarray of Size K**
    - **Problem**: Given an array of positive numbers and a positive integer k, find the maximum sum of any contiguous subarray of size k.
    - **Example**: Input: arr = [1, 4, 2, 10, 2, 3, 1, 0, 20], K = 3 -> Output: 21 (from subarray [1, 0, 20])
    - **Approach**: Calculate the sum of the first k elements. Then, slide the window: subtract the element leaving the window and add the new element entering the window. Keep track of the maximum sum found.
- **Medium**: **Longest Substring with K Distinct Characters**
    - **Problem**: Given a string (or character array) and an integer k, find the length of the longest substring in it that contains no more than k distinct characters.
    - **Example**: Input: str = "araaci", K = 2 -> Output: 4 (for "araa")
    - **Approach**: Use a sliding window and a hash map to keep track of character frequencies. Expand the window until it contains more than k distinct characters, then shrink from the left until the condition is met again.
- **Hard**: **Minimum Window Substring**
    - **Problem**: Given two strings s and t, find the minimum window substring of s that contains all the characters of t.
    - **Example**: Input: s = "ADOBECODEBANC", t = "ABC" -> Output: "BANC"
    - **Approach**: Use a sliding window and a frequency map for characters in t. Expand the window to include all characters from t, then try to shrink it from the left while maintaining the condition.

Pattern 3: Prefix Sum / Suffix Sum

This pattern involves precomputing sums of elements up to a certain index (prefix sum) or from the end to a certain index (suffix sum) to answer range queries or find subarrays with specific sums efficiently.

- **Easy**: **Range Sum Query - Immutable**
  - **Problem**: Given an integer array nums, handle multiple queries of the form sumRange(i, j), which returns the sum of elements between indices i and j (inclusive).
  - **Example**: Input: nums = [-2, 0, 3, -5, 2, -1], sumRange(0, 2) -> Output: 1 ((−2)+0+3=1)
  - **Approach**: Create a prefix sum array P where P[i] is the sum of nums[0] through nums[i-1]. Then sumRange(i, j) is simply P[j+1] - P[i].
- **Medium**: **Subarray Sum Equals K**
  - **Problem**: Given an array of integers nums and an integer k, return the total number of continuous subarrays whose sum equals k.
  - **Example**: Input: nums = [1, 1, 1], k = 2 -> Output: 2 (subarrays [1,1] from index 0 and [1,1] from index 1)
  - **Approach**: Use prefix sums and a hash map. Maintain a running sum. For each element, check if current_sum - k exists in the hash map. If it does, add its frequency to the count. Store current_sum and its frequency in the map.
- **Hard**: **Find Pivot Index / Equilibrium Index**
  - **Problem**: Given an array of integers nums, calculate the pivot index of this array. The pivot index is the index where the sum of all the numbers strictly to the left of the index is equal to the sum of all the numbers strictly to the right of the index.
  - **Example**: Input: nums = [1, 7, 3, 6, 5, 6] -> Output: 3 (At index 3, left sum 1+7+3=11, right sum 5+6=11)
  - **Approach**: Calculate the total sum of the array. Iterate through the array, maintaining a left_sum. For each element, check if left_sum equals total_sum - left_sum - current_element.

Pattern 4: Sorting & Searching

Many array problems can be simplified or solved efficiently by first sorting the array. Binary search is a key searching algorithm for sorted arrays.

- **Easy**: **Find Kth Largest Element**

    - **Problem**: Given an unsorted array nums and an integer k, return the kth largest element in the array.
    - **Example**: Input: nums = [3,2,1,5,6,4], k = 2 -> Output: 5
    - **Approach**: Sort the array and return the element at nums.length - k index. More efficient solutions involve using a min-heap or Quickselect.
- **Medium**: **Merge Intervals**
    - **Problem**: 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**: Input: intervals = [[1,3],[2,6],[8,10],[15,18]] -> Output: [[1,6],[8,10],[15,18]]
    - **Approach**: Sort the intervals based on their start times. Iterate through the sorted intervals, merging if there's an overlap (current_end >= next_start).
- **Hard**: **Median of Two Sorted Arrays**
    - **Problem**: Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log(m+n)).
    - **Example**: Input: nums1 = [1,3], nums2 = [2] -> Output: 2.0
    - **Approach**: This is a classic binary search problem on the smaller array. The goal is to partition both arrays such that elements on the left are less than or equal to elements on the right, and the number of elements in the left partition equals half the total.

Pattern 5: In-place Operations / Modifications

Problems where you need to modify the array directly without using extra space (or using O(1) auxiliary space).

- **Easy**: **Remove Duplicates from Sorted Array In-place**
    - **Problem**: Given a sorted array nums, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same.
    - **Example**: Input: nums = [1,1,2] -> Output: 2, with nums becoming [1,2,_] (underscores denote irrelevant values beyond the new length).
    - **Approach**: Use two pointers: one to iterate through the array, and another to keep track of the position of the next unique element.
- **Medium**: **Rotate Array**

    - **Problem**: Given an integer array nums, rotate the array to the right by k steps, where k is non-negative. Perform the rotation in-place.
    - **Example**: Input: nums = [1,2,3,4,5,6,7], k = 3 -> Output: [5,6,7,1,2,3,4]
    - **Approach**: Multiple ways:
        1. Using a temporary array (not in-place, but simple).
        2. Reversing portions of the array (reverse all, then reverse first k, then reverse remaining).
        3. Cyclic replacements.
- **Hard**: **First Missing Positive**
    - **Problem**: Given an unsorted integer array nums, return the smallest missing positive integer. You must implement an algorithm that runs in O(N) time and uses O(1) auxiliary space.
    - **Example**: Input: nums = [3,4,-1,1] -> Output: 2
    - **Approach**: Use the array itself as a hash map. Try to place each positive number x at index x-1. Iterate and find the first index i where nums[i] != i+1.

Pattern 6: Hashing / Frequency Counting

Using hash maps (dictionaries) or frequency arrays to store counts, indices, or check for presence of elements to achieve O(1) average time complexity for lookups.

- **Easy**: **Contains Duplicate**
    - **Problem**: Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.
    - **Example**: Input: nums = [1,2,3,1] -> Output: true
    - **Approach**: Use a hash set to store seen elements. Iterate through the array; if an element is already in the set, return true. Otherwise, add it.
- **Medium**: **Group Anagrams**
    - **Problem**: Given an array of strings strs, group the anagrams together. You can return the answer in any order. An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.
    - **Example**: Input: strs = ["eat","tea","tan","ate","nat","bat"] -> Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

    - **Approach**: For each string, create a canonical representation (e.g., sorted string or character count array). Use this representation as a key in a hash map, and store lists of anagrams as values.
- **Hard**: **Longest Consecutive Sequence**
    - **Problem**: Given an unsorted array of integers nums, return the length of the longest consecutive elements sequence. You must write an algorithm that runs in O(N) time.
    - **Example**: Input: nums = [100,4,200,1,3,2] -> Output: 4 (The longest consecutive sequence is [1, 2, 3, 4])
    - **Approach**: Store all numbers in a hash set for O(1) lookups. Iterate through the set. For each number, check if number - 1 exists. If not, it's a potential start of a sequence. Then count the consecutive numbers from there by checking number + 1, number + 2, etc.

Pattern 7: Kadane's Algorithm & Subarray Problems

Specifically for finding the maximum (or minimum) sum/product of a contiguous subarray. Kadane's algorithm is a classic dynamic programming approach for maximum subarray sum.

- **Easy**: **Maximum Subarray Sum** (Kadane's Algorithm)
    - **Problem**: Given an integer array nums, find the subarray with the largest sum, and return its sum. A subarray is a contiguous non-empty sequence of elements within an array.
    - **Example**: Input: nums = [-2,1,-3,4,-1,2,1,-5,4] -> Output: 6 (subarray [4,-1,2,1])
    - **Approach**: Maintain current_max (max sum ending at current position) and global_max (overall max sum). current_max = max(nums[i], nums[i] + current_max).
- **Medium**: **Maximum Product Subarray**
    - **Problem**: Given an integer array nums, find a contiguous non-empty subarray within the array that has the largest product, and return the product.
    - **Example**: Input: nums = [2,3,-2,4] -> Output: 6 (subarray [2,3])
    - **Approach**: Similar to Kadane's, but you need to track both max_so_far and min_so_far because a negative number multiplied by another negative number can turn into a large positive.
- **Hard**: **Maximum Sum Circular Subarray**
    - **Problem**: Given a circular integer array nums (the next element of nums[n-1] is nums[0]), return the maximum possible sum of a non-empty subarray.

    - **Example**: Input: nums = [1,-2,3,-2] -> Output: 3 (subarray [3])
    - **Approach**: The max sum subarray can be either a normal subarray (solved by Kadane's) or a circular subarray (which means total_sum - minimum_subarray_sum). Consider edge cases where all numbers are negative.

Pattern 8: Cyclic Sort

This pattern is applicable when the array contains numbers within a specific range (e.g., 1 to N, or 0 to N). It aims to place each number at its correct sorted position (nums[i]=i+1 or nums[i]=i).

- **Easy**: **Find the Missing Number**
    - **Problem**: Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.
    - **Example**: Input: nums = [3,0,1] -> Output: 2
    - **Approach**: Use Cyclic Sort to place nums[i] at index nums[i]. Then iterate and find the first index i where nums[i] != i. Alternatively, sum up numbers from 0 to n and subtract the sum of elements in the array.
- **Medium**: **Find All Duplicates in an Array**
    - **Problem**: Given an integer array nums of length n where all the integers of nums are in the range [1, n] and each integer appears once or twice, return an array of all the integers that appear twice. You must write an algorithm that runs in O(N) time and uses only constant extra space.
    - **Example**: Input: nums = [4,3,2,7,8,2,3,1] -> Output: [2,3]
    - **Approach**: Use Cyclic Sort. Iterate through the array. If nums[i] is not at its correct position (nums[i] != nums[nums[i]-1]), swap it. If it's a duplicate (i.e., nums[i] == nums[nums[i]-1]), move on. After placing elements, iterate again to find numbers not at their correct positions. A clever alternative is to mark visited numbers by negating elements at indices.
- **Hard**: **Find the Corrupt Pair (Set Mismatch)**
    - **Problem**: You have a set of integers s, which originally contained all the numbers from 1 to n. Unfortunately, due to an error, one of the numbers in s got duplicated to another number in the set, which results in repetition of one number and loss of another number. Given an integer array nums representing the data status of this set after the error, return the sum of the duplicate number and the missing number.
    - **Example**: Input: nums = [1,2,2,4] -> Output: [2,3] (2 is duplicate, 3 is missing)

    - **Approach**: Use Cyclic Sort to place each number x at index x-1. After the rearrangement, iterate through the array. The index i where nums[i] is not equal to i+1 indicates that nums[i] is the duplicate and i+1 is the missing number.

Pattern 9: Interval Problems

These problems involve working with intervals, often requiring sorting them by start or end times to identify overlaps, merges, or gaps.

- **Easy**: **Check for Overlapping Intervals**
    - **Problem**: Given a collection of intervals, determine if any two intervals overlap.
    - **Example**: Input: intervals = [[1,3],[4,6],[7,9]] -> Output: false; Input: intervals = [[1,3],[2,4]] -> Output: true
    - **Approach**: Sort intervals by their start times. Then, iterate and check if current_end is greater than or equal to next_start.
- **Medium**: **Insert Interval**
    - **Problem**: You are given an array of non-overlapping intervals intervals, sorted by their start times, and a newInterval. Insert newInterval into intervals such that intervals is still sorted in ascending order by start time and remains non-overlapping.
    - **Example**: Input: intervals = [[1,3],[6,9]], newInterval = [2,5] -> Output: [[1,5],[6,9]]
    - **Approach**: Iterate through existing intervals. Add intervals that come before newInterval or after newInterval without overlap. For overlapping intervals, merge them with newInterval.
- **Hard**: **Meeting Rooms II**
    - **Problem**: Given an array of meeting time intervals intervals where intervals[i] = [start_i, end_i], find the minimum number of conference rooms required.
    - **Example**: Input: intervals = [[0, 30],[5, 10],[15, 20]] -> Output: 2 (One room for [0,30], another for [5,10] and [15,20])
    - **Approach**: Sort start times and end times separately. Use two pointers to iterate through the sorted times. Increment room count for a start time, decrement for an end time. Keep track of the maximum concurrent meetings. (Alternatively, use a min-heap to store end times of ongoing meetings).

Pattern 10: Dynamic Programming on Arrays

Many optimization problems involving arrays can be solved using dynamic programming, where solutions to subproblems are stored to avoid recomputation.

- **Easy**: **Climbing Stairs**
  - **Problem**: You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top? (Often presented as a DP problem that uses an array to store subproblem solutions).
  - **Example**: Input: n = 3 -> Output: 3 (1+1+1, 1+2, 2+1)
  - **Approach**: This is a Fibonacci sequence. dp[i] = dp[i-1] + dp[i-2]. Base cases: dp[0]=1, dp[1]=1.
- **Medium**: **House Robber**
  - **Problem**: You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses are arranged in a circle. If two adjacent houses are robbed, it will automatically trigger an alarm. Determine the maximum amount of money you can rob tonight without alerting the police.
  - **Example**: Input: nums = [2,3,2] -> Output: 3 (You cannot rob nums[0] and nums[2] because they are adjacent if array is circular. If linear [2,3,2] -> max is 3. For circular, you either exclude the first house or the last house and apply standard house robber logic.)
  - **Approach**: For linear array: dp[i] = max(dp[i-1], dp[i-2] + nums[i]). For circular: apply linear solution twice, once excluding first house, once excluding last house, and take max.
- **Hard**: **Trapping Rain Water**
  - **Problem**: Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.
  - **Example**: Input: height = [0,1,0,2,1,0,1,3,2,1,2,1] -> Output: 6
  - **Approach**: Can be solved with DP by pre-calculating left_max and right_max arrays for each position. The water trapped at index i is min(left_max[i], right_max[i]) - height[i]. Sum these up. (Also solvable with two pointers).

This guide covers the most important topics and patterns for array-based problems in SDE interviews. Remember to

practice a variety of problems under each pattern to solidify your understanding. Good luck!

Strings: Comprehensive Guide for SDE Interviews

Strings are fundamental data structures in computer science, representing sequences of characters. Mastering string manipulation and algorithms is essential for SDE interviews, as they appear frequently in various forms.

1. Essential Topics to Know in Strings

Before diving into problem patterns, ensure you have a solid grasp of these core string concepts:

- **Definition and Properties**: Understand strings as immutable (in many languages like Java, Python) or mutable character arrays (like in C/C++). Know how they are stored in memory and basic character indexing (access).
- **Basic Operations**:
    - **Concatenation**: Combining strings.
    - **Substring**: Extracting a portion of a string.
    - **Length**: Determining the number of characters.
    - **Comparison**: Lexicographical comparison.
    - **Character Access**: Retrieving a character at a specific index.
    - **Conversion**: Between strings, character arrays, and numeric types.
- **Character Sets and Encoding**: Basic understanding of ASCII, Unicode, and UTF-8, and how they affect string length and character representation, especially for multi-byte characters.
- **String Matching Algorithms**: While you might not implement all of them from scratch, understand the concepts, time/space complexities, and general ideas behind:
    - Naive String Matching (O(MN))
    - Rabin-Karp (using hashing, O(M+N) average)
    - Knuth-Morris-Pratt (KMP) (using a prefix function, O(M+N))
    - Boyer-Moore (often faster in practice, O(M/N) best case)
- **Regular Expressions**: Basic usage for pattern matching, validation, and searching within strings.
- **Immutability vs. Mutability**: Understand the implications of string immutability in languages where strings cannot be changed after creation (modifications typically create new string objects).

- **Palindrome Check**: Efficiently determining if a string reads the same forwards and backward, considering various constraints (case, alphanumeric only).
- **Anagrams**: Identifying strings that are permutations of each other, often involving character frequency counts.
- **Edge Cases**: Handling empty strings, single-character strings, strings with special characters, case sensitivity, strings with leading/trailing spaces, and very long strings.

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in string-based SDE interview questions, along with example problems for varying difficulty levels. Many complex string problems are often combinations or variations of these fundamental patterns.

Pattern 1: Two Pointers

This technique involves using two pointers that traverse the string (or two strings) to perform in-place modifications, checks, or comparisons efficiently. Pointers often move towards each other from opposite ends or in the same direction at different speeds.

- **Easy**: **Valid Palindrome**
  - **Problem**: Given a string `s`, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.
  - **Example**: Input: `"A man, a plan, a canal: Panama"` -> Output: `true`
  - **Approach**: Initialize one pointer `left` at the beginning and `right` at the end. Increment `left` and decrement `right` simultaneously, skipping non-alphanumeric characters. Convert characters to lowercase and compare. If they don't match, return `false`. If pointers cross or meet, return `true`.
- **Medium**: **Reverse Vowels of a String**
  - **Problem**: Given a string `s`, reverse only all the vowels in the string and return it. Vowels are 'a', 'e', 'i', 'o', 'u', and they can appear in both lower and upper cases.
  - **Example**: Input: `"hello"` -> Output: `"holle"`
  - **Approach**: Convert the string to a character array. Use two pointers, `left` and `right`, starting at the

beginning and end. While `left < right`, move `left` forward until it points to a vowel, and `right` backward until it points to a vowel. Once both pointers are on vowels, swap them. Then increment `left` and decrement `right`.

- **Hard**: **Shortest Palindrome**
  - **Problem**: You are given a string `s`. You can convert `s` to a palindrome by adding characters in front of it. Find the shortest palindrome you can make by performing this transformation.
  - **Example**: Input: `"aacecaaa"` -> Output: `"aaacecaaa"`
  - **Approach**: The goal is to find the longest palindromic prefix of the string `s`. The characters in `s` after this longest palindromic prefix need to be reversed and prepended to `s`. This can be efficiently solved by using string hashing or by leveraging the Longest Prefix Suffix (LPS) array concept from the KMP algorithm to find the longest suffix of `s` that is also a prefix of `reverse(s)`.

Pattern 2: Sliding Window

This technique is highly effective for problems that involve finding a contiguous subarray or substring that satisfies a certain condition. A "window" of elements "slides" over the array/string, dynamically expanding or shrinking based on the condition.

- **Easy**: **Longest Substring Without Repeating Characters**
  - **Problem**: Given a string `s`, find the length of the longest substring without repeating characters.
  - **Example**: Input: `"abcabcbb"` -> Output: `3` (for "abc")
  - **Approach**: Use a sliding window defined by two pointers, `start` and `end`, and a hash set (or frequency map) to store characters within the current window. Expand the window by incrementing `end`. If `s[end]` is already in the set, it means a repeating character is encountered; then, shrink the window from the `start` until the duplicate is removed from the set. Keep track of the maximum window size (length of the longest substring).
- **Medium**: **Permutation in String**
  - **Problem**: Given two strings `s1` and `s2`, return `true` if `s2` contains a permutation of `s1`, or `false` otherwise. (In other words, check if `s2` contains any anagram of `s1` as a substring).
  - **Example**: Input: `s1 = "ab"`, `s2 = "eidbaooo"` -> Output: `true` (since "ba" is a permutation of "ab" and is a

substring of "eidbaooo")
    - **Approach**: This is a fixed-size sliding window problem. Calculate the frequency map of characters for `s1`. Then, use a sliding window of the same size as `s1` over `s2`. Maintain a frequency map for the current window in `s2`. In each step, expand the window by adding a character and shrink by removing a character. Compare the window's frequency map with `s1`'s frequency map.
- **Hard**: **Minimum Window Substring**
    - **Problem**: Given two strings `s` and `t`, find the minimum window substring of `s` that contains all the characters of `t`. If no such substring exists, return an empty string.
    - **Example**: Input: `s = "ADOBECODEBANC"`, `t = "ABC"` -> Output: `"BANC"`
    - **Approach**: Use a sliding window with two pointers, `left` and `right`. Maintain a frequency map for characters in `t` (target map) and a frequency map for characters within the current window (window map). Expand the window (`right++`). As characters are added, update the `window map` and count how many characters from `t` are "matched" (their count in `window map` meets or exceeds their count in `target map`). Once all characters from `t` are matched, try to shrink the window from the `left` (`left++`) while maintaining the "matched" condition, updating the minimum window found.

Pattern 3: Hashing / Frequency Counting

Hash maps (dictionaries) or frequency arrays are invaluable for problems involving character counts, unique characters, grouping strings (like anagrams), or quickly checking for the presence of characters.

- **Easy**: **Valid Anagram**
    - **Problem**: Given two strings `s` and `t`, return `true` if `t` is an anagram of `s`, and `false` otherwise.
    - **Example**: Input: `s = "anagram"`, `t = "nagaram"` -> Output: `true`
    - **Approach**: If the lengths are different, they can't be anagrams. Use a frequency array (e.g., of size 26 for lowercase English letters) or a hash map to count character occurrences in `s`. Then, iterate through `t`, decrementing counts. If a character in `t` is not found or its count becomes negative, return `false`. Finally, ensure all counts in the frequency map are zero.
- **Medium**: **Group Anagrams**

    - **Problem**: Given an array of strings `strs`, group the anagrams together. You can return the answer in any order.
    - **Example**: Input: `strs = ["eat","tea","tan","ate","nat","bat"]` -> Output: `[["bat"],["nat","tan"],["ate","eat","tea"]]`
    - **Approach**: For each string, create a canonical representation that is unique to all its anagrams. The most common canonical forms are a sorted version of the string (e.g., "eat" -> "aet") or a string representation of its character frequency count (e.g., "a1e1t1"). Use this canonical form as a key in a hash map, and append the original string to the list of strings associated with that key.
- **Hard**: **Longest Happy String**
    - **Problem**: A string `s` is called a "happy" string if it does not contain any of "aaa", "bbb", or "ccc" as a substring. Given three integers `a`, `b`, and `c`, return the longest possible happy string that can be formed using at most `a` occurrences of 'a', at most `b` occurrences of 'b', and at most `c` occurrences of 'c'.
    - **Example**: Input: `a = 1, b = 1, c = 7` -> Output: `"ccaccbcc"`
    - **Approach**: This is a greedy problem often solved with a max-heap (priority queue). Store the available counts of 'a', 'b', and 'c' along with their characters in the heap. In each step, extract the character with the highest count. Append it to the result string, ensuring it doesn't create "aaa", "bbb", or "ccc". If adding the highest count character would violate the condition, try the next highest count character. Re-insert counts into the heap after use.

Pattern 4: Dynamic Programming on Strings

Dynamic programming is crucial for optimization problems on strings, especially those involving subsequences, palindromes, or edit distances, where subproblems overlap.

- **Easy**: **Longest Common Prefix**
    - **Problem**: Write a function to find the longest common prefix string amongst an array of strings.
    - **Example**: Input: `strs = ["flower","flow","flight"]` -> Output: `"fl"`
    - **Approach**: This can be solved by iterating through the characters of the first string and comparing each character with the corresponding character in all other strings. Stop when a mismatch is found or a string ends. While not typically a complex DP problem, it shares the idea of building a solution from smaller

prefixes.

- **Medium**: **Longest Palindromic Substring**
  - **Problem**: Given a string `s`, return the longest palindromic substring in `s`.
  - **Example**: Input: `s = "babad"` -> Output: `"bab"` (or "aba")
  - **Approach**: A DP approach uses a 2D boolean array `dp[i][j]` where `dp[i][j]` is `true` if the substring `s[i...j]` is a palindrome. The recurrence relation is: `dp[i][j] = (s[i] == s[j]) && dp[i+1][j-1]`. Base cases are for single characters (`j=i`) and two characters (`j=i+1`). Iterate with increasing length of substrings. Keep track of the start and end of the longest palindrome found.
- **Hard**: **Edit Distance (Levenshtein Distance)**
  - **Problem**: Given two strings `word1` and `word2`, return the minimum number of operations required to convert `word1` to `word2`. The allowed operations are insert, delete, or replace a character.
  - **Example**: Input: `word1 = "horse"`, `word2 = "ros"` -> Output: `3`
  - **Approach**: Use a 2D DP array `dp[i][j]` representing the minimum operations to convert `word1[0...i-1]` to `word2[0...j-1]`.
    - If `word1[i-1] == word2[j-1]`: `dp[i][j] = dp[i-1][j-1]` (no operation needed for current characters).
    - Else: `dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])` (1 plus minimum of delete from `word1`, insert into `word2`, or replace).

Pattern 5: String Manipulation / Character Array Conversion

These problems involve direct manipulation of characters, often by converting the string to a mutable character array, performing operations, and then converting it back to a string. They test understanding of string immutability and efficient character-level operations.

- **Easy**: **Reverse String**
  - **Problem**: Write a function that reverses a string. The input string is given as a character array `char[]`. Do not allocate extra space for another array.
  - **Example**: Input: `s = ["h","e","l","l","o"]` -> Output: `["o","l","l","e","h"]`

    - **Approach**: Use the two-pointer approach described earlier: one pointer at the start and one at the end. Swap the characters pointed to by `left` and `right`, then move `left` forward and `right` backward until they meet or cross.
- **Medium**: **String to Integer (atoi)**
    - **Problem**: Implement the `myAtoi(string s)` function, which converts a string to a 32-bit signed integer. The function should handle various edge cases, including leading/trailing spaces, optional sign characters, non-digit characters, and integer overflow.
    - **Example**: Input: `" -42"` -> Output: `-42`
    - **Approach**: First, skip any leading whitespace. Then, check for an optional sign ('+' or '-'). Iterate through the remaining characters. If a character is a digit, convert it to its numeric value and append it to the accumulating result, carefully checking for potential integer overflow (e.g., before multiplying by 10 or adding the digit). Stop when a non-digit character is encountered.
- **Hard**: **Text Justification**
    - **Problem**: Given an array of strings `words` and a width `maxWidth`, format the text such that each line has exactly `maxWidth` characters and is fully (left and right) justified. You should pack your words in a greedy approach.
    - **Example**: Input: `words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16` -> Output: `["This is an", "example of text", "justification. "]`
    - **Approach**: Iterate through the words, greedily adding as many words as possible to the current line without exceeding `maxWidth`. Calculate the total length of words in the line and the number of spaces needed. Distribute these spaces evenly among the gaps between words. Handle the last line and single-word lines specially (left-justified with trailing spaces).

Pattern 6: Trie (Prefix Tree)

Tries are specialized tree-like data structures that are particularly efficient for storing and searching strings based on their prefixes. They are ideal for problems involving autocomplete, spell checking, or finding words with common prefixes.

- **Easy**: **Implement Trie (Prefix Tree)**
  - **Problem**: Implement a Trie (prefix tree) data structure with `insert`, `search`, and `startsWith` methods.
  - **Example**: `Trie trie = new Trie(); trie.insert("apple"); trie.search("apple"); // returns true; trie.startsWith("app"); // returns true; trie.search("app"); // returns false`
  - **Approach**: Each node in the Trie represents a character. A `TrieNode` typically contains an array or hash map of children nodes (one for each possible next character) and a boolean flag indicating if the node marks the end of a word. Implement `insert` by traversing and creating nodes as needed. `search` and `startsWith` involve traversing the Trie based on the input string.
- **Medium**: **Word Search II**
  - **Problem**: Given an `m x n` `board` of characters and a list of strings `words`, return all words on the `board` that can be formed by concatenating cells horizontally or vertically, where the same cell cannot be used more than once in a word.
  - **Example**: Input: `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"]`
  - **Approach**: Build a Trie from the given `words` list. Then, iterate through each cell of the `board` and start a Depth-First Search (DFS) from that cell. During the DFS, traverse the Trie simultaneously. If the current path of characters on the board matches a prefix in the Trie, continue the DFS. If a complete word is found in the Trie, add it to the results. To avoid duplicate words and re-using cells in a single word, mark visited cells during DFS and unmark them upon backtracking.
- **Hard**: **Concatenated Words**
  - **Problem**: Given a list of words, return all words that are a concatenation of at least two shorter words in the given list.
  - **Example**: Input: `words = ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]` -> Output: `["catsdogcats","dogcatsdog","ratcatdogcat"]`
  - **Approach**: This problem can be solved efficiently by first sorting the words by length. Store all words in a hash set for quick lookup. For each word, use dynamic programming (`dp[i]`) or a recursive helper function (with memoization) to determine if its prefix `s[0...i-1]` can be formed by concatenating other words from the set. The base case is `dp[0] = true` (empty string can be formed). Then, `dp[i]` is true if there exists some `j < i`

such that `dp[j]` is true AND the substring `s[j...i-1]` is present in the original word set.

Pattern 7: Recursion / Backtracking

This pattern is essential for problems that involve exploring all possible permutations, combinations, or paths within a string, often requiring recursive calls and managing state changes (like visited characters or remaining choices).

- **Easy**: **Permutations of a String**
  - **Problem**: Given a string `s`, print all possible permutations of the string.
  - **Example**: Input: `"abc"` -> Output: `"abc"`, `"acb"`, `"bac"`, `"bca"`, `"cab"`, `"cba"`
  - **Approach**: Use recursion. In each recursive call, iterate through the characters not yet used. Select a character, append it to the current permutation, mark it as used, and make a recursive call. Backtrack by unmarking the character and removing it from the current permutation.
- **Medium**: **Word Break**
  - **Problem**: Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.
  - **Example**: Input: `s = "leetcode"`, `wordDict = ["leet", "code"]` -> Output: `true`
  - **Approach**: This can be solved with recursion and memoization (top-down DP). For a given substring `s[i...n-1]`, try all possible splits `s[i...j]` where `s[i...j]` is in `wordDict`. If it is, recursively check if the remaining part `s[j+1...n-1]` can also be segmented. Memoize results for substrings to avoid re-computation.
- **Hard**: **Regular Expression Matching**
  - **Problem**: Given an input string `s` and a pattern `p`, implement regular expression matching with support for `.` (matches any single character) and `*` (matches zero or more of the preceding element).
  - **Example**: Input: `s = "aab"`, `p = "c*a*b"` -> Output: `true` (`c` can be repeated zero times, `a` can be repeated one time.)
  - **Approach**: This is a classic recursive (with memoization) or dynamic programming problem. Define `dp[i][j]` as `true` if `s[0...i-1]` matches `p[0...j-1]`. Handle two cases for `p[j-1]`:
    - If `p[j-1]` is a regular character or `.`: `dp[i][j] = dp[i-1][j-1]` if `s[i-1]` matches `p[j-1]`.
    - If `p[j-1]` is `*`:

- Zero occurrences: `dp[i][j] = dp[i][j-2]`
- One or more occurrences: `dp[i][j] = dp[i-1][j]` if `s[i-1]` matches `p[j-2]` (the character before `*`).

This revised guide provides a comprehensive overview of essential string topics and key interview patterns, along with illustrative examples for various difficulty levels. Consistent practice across these patterns will significantly improve your problem-solving skills for string-related questions.

Recursion, Backtracking & Dynamic Programming: Comprehensive Guide for SDE Interviews

Recursion, Backtracking, and Dynamic Programming are closely related and fundamental algorithmic paradigms. They are essential tools for solving a wide range of complex problems in software engineering interviews, particularly those involving combinatorial search, optimization, and decision-making. While distinct, they often build upon each other: recursion is the underlying mechanism, backtracking is a systematic way to explore possibilities using recursion, and dynamic programming optimizes recursive solutions with overlapping subproblems.

1. Recursion: The Foundation

Recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem. It's about a function calling itself.

Essential Concepts of Recursion

- **Definition**: A function is recursive if it calls itself directly or indirectly to solve a smaller version of its problem.
- **Base Case**: The most crucial part of a recursive function. It's the condition that stops the recursion and provides a direct answer without further recursive calls. Without a well-defined base case, recursion leads to an infinite loop (stack overflow).
- **Recursive Step**: The part of the function where the problem is broken down into one or more smaller, self-similar subproblems, and the function calls itself with these smaller inputs. It moves towards the base case.
- **Call Stack**: When a function calls itself, each call is pushed onto the call stack. Understanding how the stack works is vital for debugging recursive functions and preventing stack overflow errors (which occur when the stack runs out of memory).
- **Tail Recursion**: A special case where the recursive call is the last operation performed in the function. Some compilers can optimize tail-recursive calls, transforming them into iterative loops, which can prevent stack overflow issues. While not all languages or compilers perform this optimization, it's an important concept.

Patterns in Recursion

Pattern 1.1: Standard Recursive Traversal / Calculation

These problems involve computing a value or traversing a data structure where the solution for a given input relies directly on the solution(s) for smaller, related inputs. They often don't require complex state management beyond the function parameters.

- **Easy**: **Factorial of a Number**
  - **Problem**: Write a recursive function to calculate the factorial of a non-negative integer .
  - **Example**: Input: n=5 -> Output: 120 (5!=5×4×3×2×1)
  - **Approach**: The base case is when n=0 (or n=1), where 0!=1 (or 1!=1). The recursive step is n×(n−1)!.
    - Base Case: `if n == 0, return 1`
    - Recursive Step: `return n * factorial(n-1)`
- **Easy**: **Sum of N Natural Numbers**
  - **Problem**: Write a recursive function to find the sum of the first n natural numbers.
  - **Example**: Input: n=3 -> Output: 6 (1+2+3=6)
  - **Approach**: Base case: `if n == 0, return 0`. Recursive step: `return n + sum(n-1)`.
- **Medium**: **Binary Tree Traversal (Preorder/Inorder/Postorder)**
  - **Problem**: Given the root of a binary tree, return the node values in preorder (Root, Left, Right) / inorder (Left, Root, Right) / postorder (Left, Right, Root) traversal.
  - **Example**: Input: Binary Tree root with value 1, left child 2, right child 3 (Inorder: `[2, 1, 3]`)
  - **Approach**: Define a recursive helper function that takes the current `node` and a `result` list.
    - **Preorder**: Add `node.val` to `result`, then recursively call for `node.left`, then `node.right`.
    - **Inorder**: Recursively call for `node.left`, then add `node.val`, then recursively call for `node.right`.
    - **Postorder**: Recursively call for `node.left`, then `node.right`, then add `node.val`.
    - Base Case: `if node is null, return`.
- **Medium**: **Nth Fibonacci Number**
  - **Problem**: Compute the n-th Fibonacci number. The Fibonacci sequence is defined as F(0)=0, F(1)=1, and F(n)=F(n−1)+F(n−2) for n>1.

    - **Example**: Input: n=6 -> Output: 8 (0,1,1,2,3,5,8,...)
    - **Approach**:
        - Base Cases: `if n == 0, return 0`; `if n == 1, return 1`.
        - Recursive Step: `return fib(n-1) + fib(n-2)`. (Note: This naive recursive approach is inefficient due to overlapping subproblems and is better solved with memoization/DP, but it's a good basic recursive example).
- **Hard**: **Power(x, n)**
    - **Problem**: Implement `pow(x, n)`, which calculates x raised to the power n (xn). n can be a negative integer.
    - **Example**: Input: x=2.0, n=10 -> Output: 1024.0; Input: x=2.0, n=−2 -> Output: 0.25
    - **Approach**: Use recursion with exponentiation by squaring (divide and conquer). Handle negative n by inverting x and taking absolute value of n.
        - Base Cases: `if n == 0, return 1.0`; `if n == 1, return x`.
        - Recursive Step: Compute `temp = pow(x, n/2)`. If n is even, return `temp * temp`. If n is odd, return `x * temp * temp`.
- **Hard**: **Tower of Hanoi**
    - **Problem**: Solve the Tower of Hanoi puzzle for n disks, moving them from a `source` peg to a `destination` peg using an `auxiliary` peg, adhering to the rules.
    - **Example**: Print the steps for n=3 disks.
    - **Approach**:
        - Move n−1 disks from `source` to `auxiliary` using `destination` as temp.
        - Move the n-th disk from `source` to `destination`.
        - Move n−1 disks from `auxiliary` to `destination` using `source` as temp.
        - Base Case: If n=1, directly move the disk from source to destination.

2. Backtracking: Exploring Possibilities

Backtracking is a general algorithm for finding all (or some) solutions to computational problems that incrementally

builds candidates to the solutions. When a candidate fails to meet a constraint, it "backtracks" to a previous state and tries a different path. It's often used for problems involving permutations, combinations, and decision-making within a search space.

Essential Concepts of Backtracking

- **Definition**: A systematic way to try all possible solutions to a problem. It works by building a solution step-by-step. If a partial solution can't be completed, it abandons ("backtracks" from) it and tries another path.
- **State Management**: Crucial for backtracking. This involves saving the current state (e.g., current permutation, filled cells in a grid) and being able to restore it after a recursive call returns, allowing other paths to be explored cleanly. This might involve using auxiliary data structures (like boolean arrays for visited elements) or performing "undo" operations.
- **Decision Tree / Search Space**: Visualizing a backtracking algorithm as traversing a tree where each node represents a decision. Backtracking explores branches of this tree.
- **Pruning (Optimization)**: A key optimization technique in backtracking. It involves detecting early that a partial solution cannot lead to a valid (or optimal) complete solution and abandoning that branch of the search tree. This significantly reduces the search space.
- **Output**: Backtracking algorithms typically find *all* valid solutions, unlike some greedy or DP approaches which might find only one or an optimal one.

Patterns in Backtracking

Pattern 2.1: Permutations and Combinations

These problems involve generating all unique arrangements (permutations) or selections (combinations) of elements from a given set, often by exploring choices at each step and then reverting those choices.

- **Easy**: **Generate Parentheses**
    - **Problem**: Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.
    - **Example**: Input: n=3 -> Output: `["((()))", "(()())", "(())()", "()(())", "()()()"]`

    - **Approach**: Use a recursive backtracking function. Maintain `open_count` and `close_count` (number of open/close parentheses added so far) and the `current_string`.
        - If `open_count < n`, append `(` and recurse.
        - If `close_count < open_count`, append `)` and recurse.
        - Base case: If `open_count == n` and `close_count == n`, add `current_string` to results.
- **Easy**: **Letter Combinations of a Phone Number**
    - **Problem**: Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent.
    - **Example**: Input: `"23"` -> Output: `["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]`
    - **Approach**: Map digits to letters. Use a recursive backtracking function. For each digit in the input, iterate through its corresponding letters. Add a letter to the current combination and recurse for the next digit. Backtrack after the recursive call.
- **Medium**: **Permutations I (Unique Elements)**
    - **Problem**: Given an array `nums` of distinct integers, return all possible permutations.
    - **Example**: Input: `nums = [1,2,3]` -> Output: `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`
    - **Approach**: Use a backtracking function `find_permutations(current_permutation, visited_mask)`. Iterate through `nums`. If `nums[i]` is not `visited`, add it to `current_permutation`, mark it `visited`, and recurse. After the recursive call returns, `backtrack` by unmarking `visited` and removing `nums[i]` from `current_permutation`. Base case: `len(current_permutation) == len(nums)`.
- **Medium**: **Permutations II (With Duplicates)**
    - **Problem**: Given a collection of numbers `nums` that might contain duplicates, return all unique permutations.
    - **Example**: Input: `nums = [1,1,2]` -> Output: `[[1,1,2],[1,2,1],[2,1,1]]`
    - **Approach**: Similar to Permutations I, but first sort `nums`. When iterating and choosing a number, add a condition to skip duplicates: `if i > 0 and nums[i] == nums[i-1] and not visited[i-1]: continue`. This ensures unique permutations are generated.
- **Medium**: **Combinations**
    - **Problem**: Given two integers n and k, return all possible combinations of k numbers chosen from the

range [1,n].
    - **Example**: Input: n=4,k=2 -> Output: `[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]`
    - **Approach**: Use a recursive backtracking function. Maintain `current_combination` and `start_num`. Iterate from `start_num` to `n`. Add the current number to `current_combination` and recurse with `start_num + 1`. Backtrack after the call. Base case: `len(current_combination) == k`.
- **Hard**: **N-Queens**
    - **Problem**: The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other. Return all distinct solutions.
    - **Example**: Input: n=4 -> Output: `[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]`
    - **Approach**: Use backtracking. Place queens row by row. For each cell `(row, col)` in the current `row`, check if placing a queen there is valid (no other queen in the same column, or on either of the two diagonals). If valid, place the queen, update the board state (and possibly sets for occupied columns/diagonals), and recurse for the next row. If a solution is found (all n queens placed), add it to results. Backtrack by removing the queen and reverting the board state.
- **Hard**: **Sudoku Solver**
    - **Problem**: Write a program to solve a Sudoku puzzle by filling the empty cells. A valid Sudoku solution must satisfy all rules.
    - **Example**: Input: Partially filled 9x9 Sudoku board.
    - **Approach**: Use backtracking. Find the next empty cell on the board. Try placing digits from '1' to '9' into this empty cell. For each digit, check if it's valid to place in the current row, column, and 3x3 subgrid according to Sudoku rules. If valid, place the digit, then recursively call the solver for the next empty cell. If the recursive call returns `true` (meaning a solution was found), then this path is valid, return `true`. If not, backtrack by removing the digit from the current cell and trying the next digit. If no digit works, return `false`.

Pattern 2.2: Subset Sum / Partitioning

These problems involve finding all subsets or combinations of numbers that sum up to a specific target, or partitioning

a set into parts that meet certain conditions.

- **Easy**: **Subsets**
  - **Problem**: Given an integer array `nums` of unique elements, return all possible subsets (the power set).
  - **Example**: Input: `nums = [1,2,3]` -> Output: `[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`
  - **Approach**: Use a recursive backtracking function `generate_subsets(index, current_subset)`. For each element `nums[index]`, you have two choices:
    1. Include `nums[index]`: Add it to `current_subset` and recurse for `index + 1`.
    2. Exclude nums[index]: Do not add it, and recurse for index + 1.
       After each choice, ensure you backtrack by removing the element if it was included. Add a copy of current_subset to the final results at each recursive call (or when index reaches len(nums)).
- **Medium**: **Subsets II (With Duplicates)**
  - **Problem**: Given an integer array `nums` that may contain duplicates, return all possible unique subsets (the power set).
  - **Example**: Input: `nums = [1,2,2]` -> Output: `[[],[1],[1,2],[1,2,2],[2],[2,2]]`
  - **Approach**: Similar to Subsets I, but crucial to handle duplicates to avoid redundant subsets. First, sort `nums`. In the recursive step, when iterating through choices, skip duplicate elements that have already been considered at the current level of recursion.
- **Medium**: **Combination Sum** (revisited)
  - **Problem**: Given an array of distinct integers `candidates` and a target integer `target`, return a list of all unique combinations where the chosen numbers sum to `target`. The same number may be chosen from `candidates` an unlimited number of times.
  - **Example**: Input: `candidates = [2,3,6,7]`, `target = 7` -> Output: `[[2,2,3],[7]]`
  - **Approach**: Use backtracking. Sort `candidates`. In each recursive call `find_combinations(remaining_target, start_index, current_combination)`, iterate from `start_index` to `len(candidates)-1`. If `candidate <= remaining_target`, add it to `current_combination`, and recursively call with `remaining_target - candidate` and the *same* `start_index` (to allow reuse). Backtrack after the call.
- **Medium**: **Combination Sum II (Unique Combinations, No Reuse)**
  - **Problem**: Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique

combinations in `candidates` where the candidate numbers sum to `target`. Each number in `candidates` may only be used *once* in the combination.
    - **Example**: Input: `candidates = [10,1,2,7,6,1,5]`, `target = 8` -> Output: `[[1,1,6],[1,2,5],[1,7],[2,6]]`
    - **Approach**: Similar to Combination Sum, but with two key differences:
        1. After choosing a candidate, recursively call with `start_index + 1` (to prevent reuse of the same element).
        2. Handle duplicates: If `candidates[i]` is the same as `candidates[i-1]` and `i > start_index`, skip `candidates[i]` to avoid duplicate combinations.
- **Hard**: **Partition Equal Subset Sum**
    - **Problem**: Given a non-empty array `nums` containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.
    - **Example**: Input: `nums = [1,5,11,5]` -> Output: `true` (can be partitioned into `[1, 5, 5]` and `[11]`)
    - **Approach**: First, calculate the total sum of `nums`. If the total sum is odd, it's impossible to partition into two equal sum subsets, so return `false`. Otherwise, the problem reduces to finding if there exists a subset in `nums` that sums exactly to `total_sum / 2`. This is a classic subset sum problem. A backtracking approach would explore all combinations, while a DP approach (discussed later) is more efficient.
- **Hard**: **K-Sum Subsets (e.g., 4Sum)**
    - **Problem**: Given an array `nums` and a target, find all unique quadruplets `[nums[a], nums[b], nums[c], nums[d]]` such that `nums[a] + nums[b] + nums[c] + nums[d] == target`. Generalize to K-Sum.
    - **Example**: Input: `nums = [1,0,-1,0,-2,2]`, `target = 0` -> Output: `[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]`
    - **Approach**: This is typically solved by extending the `2Sum` and `3Sum` patterns. Sort the array. Use nested loops for the first `K-2` elements, then apply the two-pointer approach for the remaining `2Sum` problem. Backtracking can be used for a more generalized K-Sum solution (e.g., `kSum(nums, target, k, start_index)`), which might use `2Sum` as a base case. This often involves significant pruning to handle duplicates and optimize performance.

Pattern 2.3: Grid Traversal / Pathfinding

Backtracking is a natural fit for exploring paths in grids, mazes, and other graph-like structures, where decisions are made at each cell (e.g., move up, down, left, right) and paths are explored until a goal is reached or a dead end is hit.

- **Easy**: **Number of Islands** (revisited)
  - **Problem**: Given an `m x n` 2D binary grid, which represents a map of '1's (land) and '0's (water), return the number of islands. An island is formed by connecting adjacent lands horizontally or vertically.
  - **Example**: Input: `grid = [["1","1","1","1","0"],["1","1","0","1","0"],["1","1","0","0","0"],["0","0","0","0","0"]]` -> Output: `1`
  - **Approach**: Iterate through the grid. When a '1' (unvisited land) is found, increment the `island_count` and start a recursive Depth-First Search (DFS) from that cell. The DFS function will explore all connected land cells ('1's), marking them as visited (e.g., by changing their value to '0' or using a separate `visited` array) to prevent recounting.
- **Easy**: **Flood Fill**
  - **Problem**: An image is represented by an `m x n` integer grid `image`. You are given coordinates `(sr, sc)` representing the starting pixel (row and column) and a `color`. Perform a flood fill on the image.
  - **Example**: Input: `image = [[1,1,1],[1,1,0],[1,0,1]]`, `sr=1, sc=1, color=2` -> Output: `[[2,2,2],[2,2,0],[2,0,1]]`
  - **Approach**: Use recursive DFS. Start from `(sr, sc)`. If the current pixel's color is the same as the starting pixel's original color and different from the new `color`, change its color to `new_color` and recursively call DFS for its four-directional neighbors. Base case: `(sr, sc)` is out of bounds, already the `new_color`, or not the `original_color`.
- **Medium**: **Unique Paths I & II**
  - **Problem**: A robot is located at the top-left corner of an `m x n` grid. The robot can only move either down or right at any point in time. It is trying to reach the bottom-right corner. How many unique paths are there? (Unique Paths II adds obstacles).
  - **Example**: Input: `m=3, n=7` -> Output: `28`
  - **Approach**: Recursive solution: `num_paths(row, col) = num_paths(row+1, col) + num_paths(row, col+1)`. Base cases: if `(row, col)` is the target, return 1; if out of bounds, return 0. For Unique Paths II, if `(row, col)` is an obstacle, return 0. This problem has overlapping subproblems, making it better suited for memoization or DP (discussed later).
- **Medium**: **Word Search**

    - **Problem**: Given an `m x n` `board` 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 adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.
    - **Example**: Input: `board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, `word = "ABCCED"` -> Output: `true`
    - **Approach**: Iterate through each cell of the board. If `board[row][col]` matches the first character of `word`, start a recursive DFS from that cell. The DFS function will check if the subsequent characters of `word` can be found in adjacent cells. To avoid using the same cell twice for a word, mark visited cells (e.g., temporarily change `board[row][col]` to a sentinel value) and backtrack by restoring them.
- **Hard**: **Unique Paths III**
    - **Problem**: You are given an `m x n` grid where: `1` is the starting square, `2` is the ending square, `0` is an empty square, and `-1` is an obstacle. Find the number of unique paths from the start to the end that walk over every non-obstacle empty square exactly once.
    - **Example**: Input: `grid = [[1,0,0,0],[0,0,0,0],[0,0,2,-1]]` -> Output: `2`
    - **Approach**: This is a more constrained grid traversal. First, count the total number of empty cells (`0`s) and locate the start (`1`) and end (`2`) positions. Use a recursive backtracking function `dfs(row, col, visited_count)`. At each step, explore valid neighbors (not out of bounds, not obstacle, not already visited). Mark the current cell as visited. If the `end` cell is reached and `visited_count` (including start and end) equals the total required cells, increment the path count. Backtrack by unmarking the cell.
- **Hard**: **Shortest Path in Binary Matrix**
    - **Problem**: Given an `n x n` binary matrix `grid`, return the length of the shortest clear path in the matrix. A clear path is a path from the top-left cell `(0, 0)` to the bottom-right cell `(n-1, n-1)` such that all visited cells are `0` and all adjacent cells in the path are 8-directionally connected.
    - **Example**: Input: `grid = [[0,1],[1,0]]` -> Output: `2`
    - **Approach**: While this can be solved with recursive DFS (with backtracking and memoization for visited paths/lengths), Breadth-First Search (BFS) is typically the most efficient approach for shortest path problems on unweighted graphs (like a grid where each step has a cost of 1). The recursive solution would

involve exploring all paths, marking distances, and tracking minimums.

3. Dynamic Programming (DP): Optimized Recursion

Dynamic Programming is an optimization technique for recursive algorithms. It's used when a problem can be broken down into smaller subproblems, and these subproblems have two key properties:

1. **Overlapping Subproblems**: The same subproblems are encountered and solved multiple times during the recursive process.
2. **Optimal Substructure**: The optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems.

DP primarily comes in two forms:

- **Memoization (Top-Down DP)**: This is a recursive approach where results of subproblems are stored (cached) in a data structure (e.g., an array or hash map). Before computing a subproblem, the function checks if the result is already in the cache. If so, it returns the cached value; otherwise, it computes and stores the result before returning. It's "top-down" because it starts from the main problem and recursively breaks it down.
- **Tabulation (Bottom-Up DP)**: This is an iterative approach. It solves the subproblems starting from the smallest (base cases) and iteratively builds up solutions to larger subproblems, storing them in a table (usually an array or 2D array). It's "bottom-up" because it starts from the simplest cases and builds towards the full solution.

Essential Concepts of Dynamic Programming

- **Memoization (Top-Down)**: As described above. It's recursion + caching.
- **Tabulation (Bottom-Up)**: Iterative approach. Build a `dp` table.
- **DP State**: Defining what `dp[i]` or `dp[i][j]` represents (e.g., the minimum cost to reach index `i`, or the maximum value for a subproblem involving `i` elements and `j` capacity).
- **Recurrence Relation**: The mathematical formula that defines how the solution to a larger subproblem is computed from smaller subproblems. This is the heart of any DP solution.

- **Base Cases**: The initial values in the `dp` table that correspond to the smallest subproblems.
- **Space Optimization**: Often, the full `dp` table is not needed. If a `dp[i]` only depends on `dp[i-1]` or `dp[i-2]`, the space complexity can be reduced from O(N) to O(1) by only storing the necessary previous values. Similarly, 2D DP can sometimes be optimized to 1D.

Patterns in Dynamic Programming

Pattern 3.1: 1D DP (Sequence Problems)

Problems where the DP state depends on a single dimension, often related to an index in a sequence or array.

- **Easy**: **Climbing Stairs**
  - **Problem**: You are climbing a staircase. It takes `n` steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?
  - **Example**: Input: `n=3` -> Output: `3` (1+1+1, 1+2, 2+1)
  - **Approach**: This is a classic Fibonacci sequence. `dp[i]` represents the number of ways to reach step `i`.
    - Recurrence: `dp[i] = dp[i-1] + dp[i-2]`
    - Base Cases: `dp[0] = 1` (or 0 depending on definition), `dp[1] = 1`.
    - Tabulation: Initialize `dp` array and fill iteratively.
- **Easy**: **House Robber I**
  - **Problem**: You are a professional robber planning to rob houses along a street. Each house has a certain amount of money. If two adjacent houses are robbed, an alarm will be triggered. Determine the maximum amount of money you can rob tonight without alerting the police.
  - **Example**: Input: `nums = [1,2,3,1]` -> Output: `4` (Rob house 1 (value 1) and house 3 (value 3))
  - **Approach**: `dp[i]` represents the maximum amount robbed up to house `i`.
    - Recurrence: `dp[i] = max(dp[i-1], dp[i-2] + nums[i])` (either skip current house, or rob current house and skip previous)
    - Base Cases: `dp[0] = nums[0]`, `dp[1] = max(nums[0], nums[1])`.
- **Medium**: **Longest Increasing Subsequence (LIS)**
  - **Problem**: Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

    - **Example**: Input: `nums = [10,9,2,5,3,7,101,18]` -> Output: `4` (LIS is `[2,3,7,101]`)
    - **Approach**: `dp[i]` represents the length of the LIS ending at index `i`.
        - Recurrence: `dp[i] = 1 + max(dp[j])` for all `j < i` where `nums[j] < nums[i]`. If no such `j` exists, `dp[i] = 1`.
        - Initialize all `dp[i]` to 1.
- **Medium**: **Coin Change**
    - **Problem**: You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money. Return the fewest number of coins that you need to make up that amount. If that amount cannot be made up by any combination of the coins, return `-1`.
    - **Example**: Input: `coins = [1,2,5]`, `amount = 11` -> Output: `3` (11=5+5+1)
    - **Approach**: `dp[i]` represents the minimum number of coins to make amount `i`.
        - Recurrence: `dp[i] = min(dp[i], 1 + dp[i - coin])` for each `coin` in `coins`.
        - Initialize `dp` array with `Infinity` except `dp[0] = 0`.
- **Medium**: **House Robber II**
    - **Problem**: Same as House Robber I, but all houses are arranged in a circle, meaning the first and last houses are adjacent.
    - **Example**: Input: `nums = [2,3,2]` -> Output: `3` (Cannot rob 2 and 2)
    - **Approach**: Due to the circular nature, you cannot rob both the first and the last house. The problem breaks down into two independent linear House Robber I problems:
        - Rob houses `[0...n-2]` (excluding the last house).
        - Rob houses [1...n-1] (excluding the first house).
          The answer is the maximum of the results from these two subproblems. Handle edge cases for small n.
- **Hard**: **Maximum Subarray Sum (Kadane's Algorithm)**
    - **Problem**: Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.
    - **Example**: Input: `nums = [-2,1,-3,4,-1,2,1,-5,4]` -> Output: `6` (subarray `[4,-1,2,1]`)
    - **Approach**: While often solved iteratively with just two variables (`current_max`, `global_max`), it's a classic example of DP where `dp[i]` is the maximum sum of a subarray ending at index `i`.

        - Recurrence: `dp[i] = max(nums[i], nums[i] + dp[i-1])`. `global_max` is the maximum value in `dp` array.
- **Hard**: **Longest Common Subsequence (LCS)**
    - **Problem**: Given two strings `text1` and `text2`, return the length of their longest common subsequence. A subsequence is formed by deleting zero or more characters from the original string without changing the order of the remaining characters.
    - **Example**: Input: `text1 = "abcde"`, `text2 = "ace"` -> Output: `3` (LCS is "ace")
    - **Approach**: `dp[i][j]` represents the length of the LCS of `text1[0...i-1]` and `text2[0...j-1]`.
        - Recurrence:
            - If `text1[i-1] == text2[j-1]`: `dp[i][j] = 1 + dp[i-1][j-1]`
            - Else: `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`
        - Base Cases: `dp[i][0] = 0`, `dp[0][j] = 0`.

Pattern 3.2: 2D DP (Grid / Matrix / String Matching / Knapsack)

Problems where the DP state depends on two dimensions, often representing indices in two strings, a grid, or items and capacity.

- **Easy**: **Unique Paths I**
    - **Problem**: A robot is located at the top-left corner of an `m x n` grid. The robot can only move either down or right at any point in time. It is trying to reach the bottom-right corner. How many unique paths are there?
    - **Example**: Input: `m=3, n=7` -> Output: `28`
    - **Approach**: `dp[i][j]` represents the number of unique paths to reach cell `(i, j)`.
        - Recurrence: `dp[i][j] = dp[i-1][j] + dp[i][j-1]`
        - Base Cases: `dp[0][j] = 1` for all `j`, `dp[i][0] = 1` for all `i`.
- **Medium**: **Unique Paths II (with Obstacles)**
    - **Problem**: Same as Unique Paths I, but some cells are obstacles.
    - **Example**: Input: `grid = [[0,0,0],[0,1,0],[0,0,0]]` -> Output: `2`
    - **Approach**: Similar to Unique Paths I, but if `grid[i][j]` is an obstacle, `dp[i][j] = 0`.

- **Medium**: **Edit Distance (Levenshtein Distance)** (revisited)
  - **Problem**: Given two strings `word1` and `word2`, return the minimum number of operations (insert, delete, or replace) required to convert `word1` to `word2`.
  - **Example**: Input: `word1 = "horse"`, `word2 = "ros"` -> Output: `3`
  - **Approach**: `dp[i][j]` represents the minimum operations to convert `word1[0...i-1]` to `word2[0...j-1]`.
    - Recurrence:
      - If `word1[i-1] == word2[j-1]`: `dp[i][j] = dp[i-1][j-1]`
      - Else: `dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])` (delete, insert, replace)
    - Base Cases: `dp[i][0] = i` (insert `i` chars), `dp[0][j] = j` (delete `j` chars).
- **Medium**: **0/1 Knapsack Problem**
  - **Problem**: Given weights and values of N items, put these items in a knapsack of capacity `W` to get the maximum total value in the knapsack. Each item can either be put or not put in the knapsack (0-1 property).
  - **Example**: Input: `weights = [1,2,3]`, `values = [6,10,12]`, `W = 5` -> Output: `22` (items with weights 2 and 3)
  - **Approach**: `dp[i][w]` represents the maximum value that can be obtained from the first `i` items with a knapsack capacity of `w`.
    - Recurrence:
      - If `weights[i-1] > w`: `dp[i][w] = dp[i-1][w]` (cannot include item `i`)
      - Else: `dp[i][w] = max(dp[i-1][w], values[i-1] + dp[i-1][w - weights[i-1]])` (max of excluding or including item `i`)
- **Hard**: **Regular Expression Matching** (revisited)
  - **Problem**: Given an input string `s` and a pattern `p`, implement regular expression matching with support for `.` (matches any single character) and `*` (matches zero or more of the preceding element).
  - **Example**: Input: `s = "aab"`, `p = "c*a*b"` -> Output: `true`
  - **Approach**: `dp[i][j]` is `true` if `s[0...i-1]` matches `p[0...j-1]`.
    - If `p[j-1]` is `.` or `s[i-1]`: `dp[i][j] = dp[i-1][j-1]`
    - If `p[j-1]` is `*`:
      - Zero occurrences: `dp[i][j] = dp[i][j-2]`

        - One or more occurrences: `dp[i][j] = dp[i-1][j]` if `s[i-1]` matches `p[j-2]`
    - Base Case: `dp[0][0] = true`.
- **Hard**: **Wildcard Matching**
    - **Problem**: Given an input string `s` and a pattern `p`, implement wildcard matching with support for `?` (matches any single character) and `*` (matches zero or more sequence of characters).
    - **Example**: Input: `s = "adceb"`, `p = "*a*b"` -> Output: `true`
    - **Approach**: Similar to Regular Expression Matching, but rules for `*` are simpler (it can match an empty string or any sequence of characters). `dp[i][j]` is `true` if `s[0...i-1]` matches `p[0...j-1]`.
        - If `p[j-1]` is `?` or `s[i-1]`: `dp[i][j] = dp[i-1][j-1]`
        - If `p[j-1]` is `*`: `dp[i][j] = dp[i-1][j] || dp[i][j-1]` (match `s` with `*` matching one char, or `*` matching empty string)

Pattern 3.3: Interval DP

These problems involve optimal solutions over intervals (subarrays/substrings), where the solution for a larger interval depends on the solutions of its smaller sub-intervals.

- **Medium**: **Palindromic Substrings (Count)**
    - **Problem**: Given a string `s`, return the number of palindromic substrings in it.
    - **Example**: Input: `s = "aaa"` -> Output: `6` ("a", "a", "a", "aa", "aa", "aaa")
    - **Approach**: Use a 2D DP table `dp[i][j]` to indicate if `s[i...j]` is a palindrome.
        - Recurrence: `dp[i][j] = (s[i] == s[j]) && (j - i <= 1 || dp[i+1][j-1])`.
        - Iterate with `length` from 1 to `n`. For each length, iterate `i` from 0 to `n - length`. Calculate `j = i + length - 1`. If `dp[i][j]` is true, increment count.
- **Hard**: **Burst Balloons**
    - **Problem**: You are given `n` balloons, indexed from `0` to `n-1`. Each balloon has numbers written on it. If you burst balloon `i`, you will get `nums[left] * nums[i] * nums[right]` coins. `left` and `right` are adjacent balloons. Find the maximum coins you can collect.

    - **Example**: Input: `nums = [3,1,5,8]` -> Output: `167`
    - **Approach**: This is a tricky interval DP. Define `dp[i][j]` as the maximum coins from bursting balloons in the range `(i, j)` (exclusive indices, meaning balloons `i+1` to `j-1`). Add dummy balloons `1` at ends: `[1, nums..., 1]`.
        - Recurrence: `dp[i][j] = max(dp[i][k] + nums[i]*nums[k]*nums[j] + dp[k][j])` for `k` from `i+1` to `j-1`. (This means `k` is the *last* balloon to be burst in `(i, j)`).
        - Iterate `length` from 2 to `n+1`. Iterate `i` from 0. Calculate `j = i + length`.

Pattern 3.4: Tree DP (Dynamic Programming on Trees)

More advanced DP problems where the structure is a tree. Solutions for a node typically depend on solutions from its children or parents.

- **Medium**: **Binary Tree Maximum Path Sum**
    - **Problem**: A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. The path does not need to pass through the root. Return the maximum path sum of any non-empty path.
    - **Example**: Input: `root = [1,2,3]` -> Output: `6` (Path `2 -> 1 -> 3`)
    - **Approach**: Use a recursive DFS function that returns the maximum path sum *starting from the current node and going downwards* (unilateral path). Maintain a `global_max` variable. At each node, calculate the path sum that *passes through* the current node (unilateral left + node.val + unilateral right) and update `global_max`. Return the unilateral path sum.
- **Hard**: **House Robber III**
    - **Problem**: Same as House Robber I, but houses are arranged in a binary tree. If two directly linked houses (parent-child) are robbed, it will trigger an alarm.
    - **Example**: Input: `root = [3,2,3,null,3,null,1]` -> Output: `7` (Rob 3 (root) and 3 (right child's left child) and 1 (right child's right child))
    - **Approach**: For each node, you have two choices: rob it or not.
        - If you rob the current node, you cannot rob its immediate children.

- If you don't rob the current node, you can choose to rob its children or not.
  Define a recursive function that returns a pair of values: [max_robbed_if_current_is_robbed, max_robbed_if_current_is_not_robbed]. Combine results from left and right children. Memoize results for each node.

This comprehensive guide covers the essential theoretical foundations and practical patterns for Recursion, Backtracking, and Dynamic Programming. Each pattern includes a range of problems from easy to hard, demonstrating the progressive complexity and application of these techniques. Consistent practice across these patterns will significantly enhance your algorithmic problem-solving skills for SDE interviews.

Linked Lists: Comprehensive Guide for SDE Interviews

Linked Lists are fundamental linear data structures where elements are not stored at contiguous memory locations. Instead, each element (node) stores a reference (or pointer) to the next element in the sequence. A strong understanding of linked lists is crucial for SDE interviews, as they often involve pointer manipulation and recursive thinking.

1. Essential Topics to Know in Linked Lists

Before diving into problem patterns, ensure you have a solid grasp of these core linked list concepts:

- **Definition**: Understand that a linked list is a collection of nodes where each node contains data and a pointer/reference to the next node. The last node's pointer points to `null` (or the head in a circular list).
- **Node Structure**: How a typical node is defined (e.g., `value`, `next` pointer; for doubly linked lists, `prev` pointer).
- **Head and Tail**: The `head` points to the first node, and the `tail` points to the last node.
- **Comparison with Arrays**:
    - **Arrays**: Contiguous memory, access by index, O(N) for insertion/deletion in the middle.
    - **Linked Lists**: Non-contiguous memory, O(N) for access by index, O(1) for insertion/deletion at specific points (if pointer to previous node is known). Dynamic size.
- **Types of Linked Lists**:
    - **Singly Linked List**: Each node points only to the next node. Traversal is unidirectional.
    - **Doubly Linked List**: Each node has pointers to both the next and previous nodes. Traversal is bidirectional.
    - **Circular Linked List**: The last node points back to the first node (head), forming a circle. Can be singly or doubly circular.
- **Basic Operations**:
    - **Traversal**: Iterating through the list from head to tail.
    - **Insertion**: At the head, at the tail, after a specific node, before a specific node.
    - **Deletion**: From the head, from the tail, of a specific node (by value or position).

    - **Searching**: Finding a node with a specific value.
- **Dummy Nodes (Sentinel Nodes)**: A pseudo-node often used at the beginning of a linked list to simplify operations (especially insertion/deletion at the head) by providing a consistent previous node.
- **Two Pointers (Fast & Slow Pointers / Floyd's Cycle-Finding Algorithm)**: A powerful technique used for:
    - Detecting cycles.
    - Finding the middle of the list.
    - Finding the k-th node from the end.
- **Recursion in Linked Lists**: Solving problems by breaking them down into subproblems where the solution for the current node depends on the solution for the rest of the list. Often elegant for reversal or specific traversals.
- **Edge Cases**: Handling empty lists, single-node lists, lists with two nodes, null pointers, and duplicate values.

2. Questions Categorized by Patterns

Here are the most common patterns encountered in linked list-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Two Pointers (Fast & Slow / Tortoise and Hare)

This pattern uses two pointers that traverse the linked list at different speeds. It's incredibly versatile for problems involving cycles, finding middle elements, or identifying specific positions relative to the end of the list.

- **Easy**: **Find the Middle of a Linked List**
    - **Problem**: Given the head of a singly linked list, return the middle node of the list. If there are two middle nodes, return the second middle node.
    - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 5` -> Output: `3`
    - **Approach**: Use two pointers, `slow` and `fast`. `slow` moves one step at a time, `fast` moves two steps at a time. When `fast` reaches the end of the list (or `null`), `slow` will be at the middle.
- **Medium**: **Linked List Cycle**
    - **Problem**: Given the head of a singly linked list, determine if the linked list has a cycle in it.
    - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 2` (4 points to 2) -> Output: `true`

    - **Approach**: Use `slow` and `fast` pointers. If they ever meet, there is a cycle. If `fast` reaches `null` (or `fast.next` is `null`), there is no cycle.
- **Medium**: **Linked List Cycle II (Find Cycle Start Node)**
    - **Problem**: Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return `null`.
    - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 2` (4 points to 2) -> Output: `node 2`
    - **Approach**: First, use fast and slow pointers to detect a cycle. If a cycle exists, move `slow` back to the head. Then, move both `slow` and `fast` (which is still at their meeting point) one step at a time. The node where they meet again is the start of the cycle.
- **Hard**: **Find K-th Node from End of Linked List**
    - **Problem**: Given the head of a singly linked list and an integer k, return the k-th node from the end of the list. Assume k is valid.
    - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 5`, k=2 -> Output: `node 4`
    - **Approach**: Use two pointers, `first` and `second`. Move `first` k steps ahead. Then move both `first` and `second` one step at a time until `first` reaches the end. When `first` is at the end, `second` will be at the k-th node from the end.

Pattern 2: Reversal

Reversing a linked list, or parts of it, is a fundamental operation that frequently appears in interviews. It tests your ability to manipulate pointers carefully.

- **Easy**: **Reverse Linked List**
    - **Problem**: Given the head of a singly linked list, reverse the list, and return the reversed list's new head.
    - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 5` -> Output: `5 -> 4 -> 3 -> 2 -> 1`
    - **Approach**: Iterative: Maintain three pointers: `prev` (initially `null`), `current` (initially `head`), and `next_node` (temporarily stores `current.next`). In each step, set `current.next = prev`, then update `prev = current`, and `current = next_node`.
    - **Recursive**: Base case: `if head == null or head.next == null, return head`. Recursive step: `new_head =`

`reverseList(head.next)`. Set `head.next.next = head` and `head.next = null`. Return `new_head`.

- **Medium**: **Reverse Linked List II (Reverse Sub-list)**
  - **Problem**: Given the head of a singly linked list and two integers `left` and `right`, reverse the nodes of the list from position `left` to `right`, and return the reversed list.
  - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 5`, `left = 2`, `right = 4` -> Output: `1 -> 4 -> 3 -> 2 -> 5`
  - **Approach**: First, traverse to the node just before `left` (let's call it `pre_left`). Then, perform the standard linked list reversal iteratively on the sub-list from `left` to `right`. Connect `pre_left` to the new head of the reversed sub-list, and the original `left` node (which becomes the new tail) to the node after `right`. A dummy node can simplify handling `left = 1`.
- **Hard**: **Reverse Nodes in K-Group**
  - **Problem**: Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list. If the number of nodes is not a multiple of k, leave the remaining nodes as they are.
  - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 5`, `k = 2` -> Output: `2 -> 1 -> 4 -> 3 -> 5`
  - **Approach**: This problem often uses a recursive approach. In each recursive call:
    1. Check if there are at least k nodes remaining. If not, return the current head.
    2. Reverse the first k nodes (using iterative reversal).
    3. The `head` of the original k-group becomes the tail of the reversed group. Its `next` pointer should point to the result of recursively calling the function on the remainder of the list.
    4. Return the new head of the reversed k-group.

Pattern 3: Merging and Splitting

These problems involve combining multiple sorted linked lists or dividing a single linked list based on certain criteria.

- **Easy**: **Merge Two Sorted Lists**
  - **Problem**: Given the heads of two sorted singly linked lists, `list1` and `list2`, merge the two lists into a single sorted list. The new list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

    - **Example**: Input: `list1 = 1->2->4`, `list2 = 1->3->4` -> Output: `1->1->2->3->4->4`
    - **Approach**: Create a dummy node to simplify handling the head. Use a `current` pointer to build the new list. Compare the values of `list1` and `list2`. Append the smaller node to `current.next` and advance that list's pointer. Repeat until one list is exhausted, then append the remainder of the other list.
- **Medium**: **Split Linked List in Parts**
    - **Problem**: Given the head of a singly linked list `root` and an integer k, return an array of `k` linked list heads. Each part should be a singly linked list.
    - **Example**: Input: `root = 1->2->3->4->5->6->7->8->9->10`, k=3 -> Output: `[[1,2,3,4],[5,6,7],[8,9,10]]`
    - **Approach**: First, find the total length of the list. Calculate `base_length = total_length / k` and `remainder = total_length % k`. Iterate k times. For each part, determine its length (`base_length + 1` for the first `remainder` parts, then `base_length`). Traverse and break the list at the calculated length, storing the head of each part.
- **Hard**: **Merge K Sorted Lists**
    - **Problem**: You are given an array of k linked-lists `lists`, each sorted in ascending order. Merge all the linked lists into one sorted linked list and return it.
    - **Example**: Input: `lists = [[1,4,5],[1,3,4],[2,6]]` -> Output: `1->1->2->3->4->4->5->6`
    - **Approach**: Multiple ways:
        1. **Iterative pairwise merge**: Merge `list1` with `list2`, then the result with `list3`, and so on. O(N×k) where N is total nodes.
        2. **Divide and Conquer**: Merge `lists[0]` and `lists[1]`, `lists[2]` and `lists[3]`, etc., recursively until one list remains. O(Nlogk).
        3. **Min-Heap (Priority Queue)**: Add the head of each list to a min-heap. Repeatedly extract the smallest node from the heap, append it to the result list, and if that node has a `next`, add `next` to the heap. O(Nlogk). This is generally the most efficient solution for large k.

Pattern 4: In-place Modifications / Pointer Manipulation

These problems focus on altering the structure of the linked list (reordering, removing nodes) without creating new

nodes, often using minimal extra space.

- **Easy**: **Remove Duplicates from Sorted List**
  - **Problem**: Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.
  - **Example**: Input: `1 -> 1 -> 2 -> 3 -> 3` -> Output: `1 -> 2 -> 3`
  - **Approach**: Iterate through the list. If `current.val == current.next.val`, skip the duplicate: `current.next = current.next.next`. Otherwise, move `current = current.next`.
- **Medium**: **Delete Node in a Linked List**
  - **Problem**: Write a function to delete a node in a singly-linked list. You will *not* be given access to the head of the list, only to the node to be deleted. It is guaranteed that the node to be deleted is not a tail node.
  - **Example**: Input: `head = 4->5->1->9`, `node = 5` -> Output: `4->1->9`
  - **Approach**: Copy the value of the next node into the current node (`node.val = node.next.val`). Then, delete the next node (`node.next = node.next.next`). This effectively "deletes" the given node by overwriting it and removing the subsequent node.
- **Medium**: **Odd Even Linked List**
  - **Problem**: Given the head of a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes. The first node is considered odd, the second node even, and so on. You should try to do it in-place.
  - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 5` -> Output: `1 -> 3 -> 5 -> 2 -> 4`
  - **Approach**: Use two pointers, `odd_tail` and `even_head`. Iterate through the list, re-linking nodes to separate odd-indexed nodes from even-indexed nodes. Finally, connect the `odd_tail` to `even_head`.
- **Hard**: **Reorder List**
  - **Problem**: You are given the head of a singly linked list `L`. Reorder the list to be `L0 -> Ln -> L1 -> Ln-1 -> L2 -> Ln-2 -> ...`. You may not modify the values in the list's nodes. Only nodes themselves may be changed.
  - **Example**: Input: `1 -> 2 -> 3 -> 4 -> 5` -> Output: `1 -> 5 -> 2 -> 4 -> 3`
  - **Approach**: This typically involves three steps:
    1. Find the middle of the list using fast/slow pointers.
    2. Split the list into two halves.

3. Reverse the second half.
4. Merge the two lists by alternating nodes from the first half and the reversed second half.

Pattern 5: Recursion in Linked Lists

Recursive solutions for linked lists are often elegant and concise, especially for operations like reversal, deep copying, or specific traversals.

- **Easy**: **Print Linked List in Reverse (without modifying)**
  - **Problem**: Given the head of a singly linked list, print its elements in reverse order without modifying the list structure.
  - **Example**: Input: `1 -> 2 -> 3` -> Output: `3, 2, 1`
  - **Approach**: Base case: `if head == null, return`. Recursive step: call `printReverse(head.next)`, then print `head.val`. The calls will return from the deepest (last) node first.
- **Medium**: **Palindrome Linked List**
  - **Problem**: Given the head of a singly linked list, return `true` if it is a palindrome.
  - **Example**: Input: `1 -> 2 -> 2 -> 1` -> Output: `true`
  - **Approach**:
    1. **Iterative**: Find the middle of the list. Reverse the second half. Compare the first half with the reversed second half. Restore the list (optional, but good practice).
    2. **Recursive (using global/class variable or stack)**: A recursive approach can be used, but it's less intuitive than iterative or using a stack. The idea is to pass the head and a reference to a `front_pointer`. As recursion unwinds, compare `front_pointer.val` with `current_node.val`. Advance `front_pointer` globally.
- **Hard**: **Flatten a Multilevel Doubly Linked List**
  - **Problem**: You are given a doubly linked list which in addition to the next and previous pointers, it could have a child pointer, which may or may not point to a separate doubly linked list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure. Flatten the list so that

all the nodes appear in a single-level, doubly linked list.
- Example: Input: 1---2---3---4---5---6
  |
  7---8---9---10
  |
  11--12
  -> Output: 1---2---3---7---8---11---12---9---10---4---5---6
- **Approach**: Use recursive DFS. For each node, if it has a child:
  1. Save the `next` pointer of the current node.
  2. Connect `current.next` to `current.child`.
  3. Connect `current.child.prev` to `current`.
  4. Recursively flatten the child list.
  5. After the child list is flattened and the recursive call returns (yielding the tail of the flattened child list), connect this tail to the saved `next` pointer (the original `next` of the current node). This requires careful handling of `prev` pointers as well.

This guide provides a comprehensive overview of essential Linked List topics and key interview patterns, along with illustrative examples for various difficulty levels. Mastering linked list problems primarily involves strong pointer manipulation skills and a good understanding of recursive thinking. Consistent practice across these patterns will significantly improve your problem-solving abilities.

Stack: Comprehensive Guide for SDE Interviews

A Stack is a linear data structure that follows a particular order in which operations are performed. The order may be **LIFO (Last In, First Out)** or **FILO (First In, Last Out)**. It's a crucial concept frequently tested in SDE interviews, often used to manage function calls, parse expressions, or solve problems requiring tracking "recent" elements.

1. Essential Topics to Know in Stacks

Before diving into problem patterns, ensure you have a solid grasp of these core stack concepts:

- **Definition**: Understand that a stack is a linear data structure that operates on the LIFO (Last In, First Out) principle.
- **Core Operations**:
  - **Push**: Adds an element to the top of the stack.
  - **Pop**: Removes the top element from the stack.
  - **Peek/Top**: Returns the top element without removing it.
  - **isEmpty**: Checks if the stack is empty.
  - **Size**: Returns the number of elements in the stack.
- **Implementation**: Understand how stacks can be implemented using:
  - **Arrays**: Fixed-size, potential for overflow/underflow, simple to implement.
  - **Linked Lists**: Dynamic size, more flexible, but slight overhead for pointers.
- **Comparison with other Data Structures**:
  - **Vs. Queue**: Stack is LIFO, Queue is FIFO.
  - **Vs. Array**: Stacks are restricted arrays (only top accessible), arrays offer random access.
- **Common Applications**:
  - Function call management (recursion implementation).
  - Expression evaluation (infix to postfix/prefix, arithmetic expression evaluation).
  - Browser history (back/forward buttons).
  - Undo/Redo functionalities in software.

    - Syntax parsing (e.g., balanced parentheses).
    - Graph traversals (DFS).
- **Edge Cases**: Handling empty stack operations (pop from empty, peek from empty), stack overflow (if using fixed-size array).

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in stack-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Basic Stack Operations / Manipulation

These problems involve direct use of stack's push, pop, and peek operations to achieve a specific ordering or filtering of elements.

- **Easy**: **Implement a Stack using an Array/Linked List**
    - **Problem**: Design a stack data structure that supports `push`, `pop`, `top`, and `empty` operations.
    - **Example**: `MyStack stack = new MyStack(); stack.push(1); stack.push(2); stack.top(); // returns 2; stack.pop(); // returns 2; stack.empty(); // returns false;`
    - **Approach**: Use either a dynamic array (like `ArrayList` or `vector`) or a singly linked list. For array, maintain a `top` index. For linked list, `push` adds to head, `pop` removes from head.
- **Easy**: **Valid Parentheses**
    - **Problem**: Given a string `s` containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. An input string is valid if open brackets are closed by the same type of brackets and in the correct order.
    - **Example**: Input: `s = "()[]{}"` -> Output: `true`; Input: `s = "([)]"` -> Output: `false`
    - **Approach**: Iterate through the string. If an opening bracket, push it onto the stack. If a closing bracket, check if the stack is empty or if its top element is the corresponding opening bracket. If a match, pop; otherwise, invalid. At the end, stack must be empty.
- **Medium**: **Remove All Adjacent Duplicates In String**

    - **Problem**: You are given a string `s` and a `k` integer. A `k` duplicate removal consists of choosing `k` adjacent and equal letters, and removing them. You repeatedly make `k` duplicate removals on `s` until you no longer can. Return the final string after all such duplicate removals.
    - **Example**: Input: `s = "deeedbbcccbdaa", k = 3` -> Output: `"aa"`
    - **Approach**: Use a stack to store characters along with their counts. Iterate through the string. If current char matches top of stack, increment count. If count reaches `k`, pop `k-1` times. Otherwise, push char with count 1.
- **Medium**: **Asteroid Collision**
    - **Problem**: We are given an array `asteroids` of integers representing asteroids in a row. For each asteroid, the absolute value represents its size, and the sign represents its direction (positive means right, negative means left). Find the state of asteroids after all collisions.
    - **Example**: Input: `asteroids = [5,10,-5]` -> Output: `[5,10]`
    - **Approach**: Use a stack to simulate collisions. Iterate through asteroids. If stack is empty or current asteroid moves right, push. If current asteroid moves left: while stack is not empty and top is right-moving: if top is smaller, pop top; if top is equal, pop top and break (current destroys top); if top is larger, break (current destroyed). If current asteroid survives, push.
- **Hard**: **Largest Rectangle in Histogram**
    - **Problem**: Given an array of integers `heights` representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.
    - **Example**: Input: `heights = [2,1,5,6,2,3]` -> Output: `10`
    - **Approach**: Use a monotonic stack (specifically, an increasing stack of indices). Iterate through `heights`. If `heights[i]` is greater than `heights[stack.top()]`, push `i`. Otherwise, pop elements from the stack that are taller than `heights[i]`. For each popped bar, calculate its area (`height * width`), where `width = i - stack.top() - 1` (or `i` if stack becomes empty). Update maximum area. Handle remaining bars in stack at the end.

Pattern 2: Monotonic Stack

A monotonic stack maintains elements in a strictly increasing or strictly decreasing order. It's often used to find the

"next greater/smaller element" or related problems efficiently.

- **Easy**: **Next Greater Element I**
  - **Problem**: Given two arrays `nums1` and `nums2` (is a subset of nums2), find the next greater element for each number in `nums1` in `nums2`. The next greater element of a number `x` in `nums2` is the first greater number to its right.
  - **Example**: Input: `nums1 = [4,1,2]`, `nums2 = [1,3,4,2]` -> Output: `[-1,3,-1]`
  - **Approach**: Use a monotonic decreasing stack and a hash map. Iterate `nums2`. For each element `num`: while stack is not empty and `stack.top() < num`, `hash_map[stack.pop()] = num`. Push `num` onto stack. After iterating `nums2`, map elements from `nums1` using the `hash_map`.
- **Medium**: **Daily Temperatures**
  - **Problem**: Given an array of integers `temperatures` representing the daily temperatures, return an array `answer` such that `answer[i]` is the number of days you have to wait after the `i`-th day to get a warmer temperature. If there is no future day for which this is possible, keep `answer[i] == 0`.
  - **Example**: Input: `temperatures = [73,74,75,71,69,72,76,73]` -> Output: `[1,1,4,2,1,1,0,0]`
  - **Approach**: Use a monotonic decreasing stack to store indices. Iterate through `temperatures`. If `temperatures[i]` is greater than `temperatures[stack.top()]`, it means `stack.top()` has found its warmer day. Pop `stack.top()`, calculate `answer[stack.top()] = i - stack.top()`, and repeat until stack is empty or condition is met. Then push `i`.
- **Hard**: **Sum of Subarray Minimums**
  - **Problem**: Given an array of integers `arr`, find the sum of `min(b)` for every contiguous subarray `b` of `arr`.
  - **Example**: Input: `arr = [3,1,2,4]` -> Output: `17`
  - **Approach**: This requires finding the "previous smaller element" (PSE) and "next smaller element" (NSE) for each element in the array. For each element `arr[i]`, it is the minimum in subarrays starting from `PSE[i]+1` to `NSE[i]-1`. Use a monotonic stack to find PSE and NSE for all elements in two passes. Then calculate `sum += arr[i] * (i - PSE[i]) * (NSE[i] - i)`.

Pattern 3: Expression Evaluation / Parsing

Stacks are fundamental for parsing and evaluating mathematical expressions, particularly when dealing with operator precedence and parentheses.

- **Easy**: **Evaluate Reverse Polish Notation (RPN)**
  - **Problem**: Evaluate the value of an arithmetic expression in Reverse Polish Notation. Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.
  - **Example**: Input: `tokens = ["2","1","+","3","*"]` -> Output: `9` (((2+1)*3)=9)
  - **Approach**: Iterate through the tokens. If a number, push it onto the stack. If an operator, pop the top two operands from the stack, perform the operation, and push the result back onto the stack. The final result is the only element left in the stack.
- **Medium**: **Basic Calculator II**
  - **Problem**: Given a string `s` which represents an arithmetic expression, evaluate it and return its value. Integer division should truncate toward zero. It's guaranteed that the given expression is always valid. All intermediate results will be in the range of [−231,231−1].
  - **Example**: Input: `s = "3+2*2"` -> Output: `7`
  - **Approach**: Iterate through the string, maintaining `current_number`, `last_operator`, and a stack. Push numbers onto the stack based on `last_operator`. Handle multiplication/division immediately. Sum up stack elements at the end.
- **Hard**: **Basic Calculator I**
  - **Problem**: Implement a basic calculator to evaluate a simple expression string. The expression string may contain open `(` and closing parentheses `)`, the plus `+` or minus `-` sign, non-negative integers, and empty spaces.
  - **Example**: Input: `s = "(1+(4+5+2)-3)+(6+8)"` -> Output: `23`
  - **Approach**: This is more complex due to parentheses. Use two stacks: one for numbers and one for operators (or signs). When `(` is encountered, push current `total_sum` and `current_sign` onto stacks and reset. When `)`, pop previous `total_sum` and `sign` to incorporate the sub-expression result.

Pattern 4: Using Stack to Implement Other Data Structures / Algorithms

Stacks can be used as building blocks for other data structures (like queues) or to simulate recursive calls for iterative solutions.

- **Easy**: **Implement Queue using Stacks**
  - **Problem**: Implement a first in, first out (FIFO) queue using only two stacks.
  - **Example**: `MyQueue queue = new MyQueue(); queue.push(1); queue.push(2); queue.peek(); // returns 1; queue.pop(); // returns 1; queue.empty(); // returns false;`
  - **Approach**: Use two stacks: `input_stack` for `push` operations and `output_stack` for `pop`/`peek`. When `pop` or `peek` is called, if `output_stack` is empty, move all elements from `input_stack` to `output_stack` (this reverses order). Then `pop`/`peek` from `output_stack`.
- **Medium**: **Min Stack**
  - **Problem**: Design a stack that supports `push`, `pop`, `top`, and retrieving the minimum element in constant time.
  - **Example**: `MinStack minStack = new MinStack(); minStack.push(-2); minStack.push(0); minStack.push(-3); minStack.getMin(); // return -3; minStack.pop(); minStack.top(); // return 0; minStack.getMin(); // return -2;`
  - **Approach**: Use two stacks: one `main_stack` for all elements, and a `min_stack` to keep track of minimums. When pushing `val`, push `val` to `main_stack`. Push `min(val, min_stack.top())` to `min_stack`. When popping, pop from both stacks.
- **Hard**: **Largest Rectangle in Histogram** (revisited, more general problem solving)
  - **Problem**: (Same as above) This problem is also a classic example of using a stack to solve a geometric problem, often involving concepts similar to finding the "next smaller element" for each bar. It highlights how a stack can efficiently track potential boundaries.
  - **Approach**: (As described in Pattern 1, Largest Rectangle in Histogram) The core is using the monotonic stack to find the boundaries (left and right smaller elements) for each bar, which then allows calculating the maximum area.

Pattern 5: Backtracking Simulation / DFS Iteration

While backtracking and DFS are inherently recursive, stacks can be used to convert recursive algorithms into iterative

ones, managing the state and choices manually.

- **Easy**: **DFS Traversal of a Graph/Tree (Iterative)**
  - **Problem**: Implement iterative Depth-First Search for a graph or tree.
  - **Example**: Given a graph (adjacency list), perform DFS starting from a node.
  - **Approach**: Use a stack and a `visited` set. Push the starting node onto the stack. While stack is not empty, pop a node. If not visited, mark as visited, process it, and push all its unvisited neighbors onto the stack.
- **Medium**: **Flatten Nested List Iterator**
  - **Problem**: Given a nested list of integers, implement an iterator to flatten it. Each element is either an integer or a list (which may contain other lists).
  - **Example**: Input: `[[1,1],2,[1,1]]` -> Output: `[1,1,2,1,1]`
  - **Approach**: Use a stack. When initializing the iterator, push all elements from the nested list onto the stack in reverse order (so that the first element comes to top). When `hasNext()` is called, pop from stack. If it's an integer, return true. If it's a list, push its elements onto stack (again, in reverse). Repeat until an integer is found or stack is empty.
- **Hard**: **Binary Tree Zigzag Level Order Traversal**
  - **Problem**: Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and so on).
  - **Example**: Input: `root = [3,9,20,null,null,15,7]` -> Output: `[[3],[20,9],[15,7]]`
  - **Approach**: Can be solved with two stacks (or one stack with a flag). Use a `current_level_stack` and a `next_level_stack`. Iterate while `current_level_stack` is not empty. Pop nodes from `current_level_stack`. If traversing left-to-right, push left child then right child to `next_level_stack`. If right-to-left, push right child then left child. When `current_level_stack` is empty, swap `current_level_stack` and `next_level_stack` and reverse direction.

This guide provides a comprehensive overview of essential Stack topics and key interview patterns, along with illustrative examples for various difficulty levels. Mastering stacks primarily involves understanding their LIFO principle and how to apply it creatively to solve problems that involve managing temporal order or maintaining specific element

relationships. Consistent practice across these patterns will significantly improve your problem-solving abilities.

Queue: Comprehensive Guide for SDE Interviews

A Queue is a linear data structure that follows a particular order in which operations are performed. The order is **FIFO (First In, First Out)** or **LILO (Last In, Last Out)**. It's a fundamental concept in computer science, often used in scenarios where processing order matters, such as task scheduling, breadth-first search (BFS), or buffer management.

1. Essential Topics to Know in Queues

Before diving into problem patterns, ensure you have a solid grasp of these core queue concepts:

- **Definition**: Understand that a queue is a linear data structure that operates on the FIFO (First In, First Out) principle. Elements are added to the rear (enqueue) and removed from the front (dequeue).
- **Core Operations**:
    - **Enqueue / Offer / Add**: Inserts an element at the rear of the queue.
    - **Dequeue / Poll / Remove**: Removes and returns the element from the front of the queue.
    - **Peek / Front**: Returns the element at the front of the queue without removing it.
    - **isEmpty**: Checks if the queue is empty.
    - **Size**: Returns the number of elements in the queue.
- **Implementation**: Understand how queues can be implemented using:
    - **Arrays**: Can be fixed-size (with potential for issues like "full" queue even if space exists, leading to circular array necessity) or dynamic.
    - **Linked Lists**: More flexible and dynamic size, but introduces pointer overhead. This is generally the most straightforward and common implementation.
    - **Circular Arrays**: Efficiently handles fixed-size queues by wrapping around indices to reuse space.
    - **Using Stacks**: As seen previously, a queue can be implemented using two stacks.
- **Comparison with other Data Structures**:
    - **Vs. Stack**: Queue is FIFO, Stack is LIFO.
    - **Vs. Array**: Queues are restricted arrays (only front/rear accessible), arrays offer random access.
- **Common Applications**:

    - **BFS (Breadth-First Search)**: Crucial for traversing graphs level by level.
    - Task Scheduling / Job Processing.
    - Buffering (e.g., I/O buffers).
    - Print spooling.
    - Handling events in simulations.
    - Message Queues in distributed systems.
- **Edge Cases**: Handling operations on an empty queue (dequeue/peek from empty), queue overflow (if using fixed-size array).

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in queue-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Basic Queue Operations / Simulation

These problems involve directly using the fundamental enqueue, dequeue, and peek operations to simulate processes, manage order, or filter elements.

- **Easy**: **Implement a Queue using an Array/Linked List**
    - **Problem**: Design a queue data structure that supports `enqueue`, `dequeue`, `front`, and `empty` operations.
    - **Example**: `MyQueue queue = new MyQueue(); queue.enqueue(1); queue.enqueue(2); queue.front(); // returns 1; queue.dequeue(); // returns 1; queue.empty(); // returns false;`
    - **Approach**: For a linked list implementation, maintain `front` and `rear` pointers. `enqueue` adds to `rear`, `dequeue` removes from `front`. For array, manage `front` and `rear` indices, possibly using a circular array approach.
- **Easy**: **Recent Counter (Fixed-Size Queue)**
    - **Problem**: Implement a `RecentCounter` class that counts recent requests within a specified time frame. It has a `ping(t)` method that records a ping at time `t` and returns the number of pings that have occurred in the inclusive time interval `[t - 3000, t]`.
    - **Example**: `rc = new RecentCounter(); rc.ping(1); // queue=[1], returns 1; rc.ping(100); // queue=[1,100], returns 2;`

`rc.ping(3001); // queue=[1,100,3001], returns 3 (1 is included); rc.ping(3002); // queue=[100,3001,3002], returns 3 (1 is removed)`

    - **Approach**: Use a `Queue` (or `Deque`). When `ping(t)` is called, enqueue `t`. Then, while the front of the queue is less than `t - 3000`, dequeue elements. Finally, return the size of the queue.
- **Medium**: **Number of Students Unable to Eat Lunch**
    - **Problem**: The school cafeteria offers circular and square lunches. Students take turns picking lunches. Students have preferences (circular or square). If a student at the front of the queue cannot eat the lunch at the top of the stack, they go to the end of the queue. Return the number of students who will eventually be unable to eat.
    - **Example**: Input: `students = [1,1,0,0]`, `sandwiches = [0,1,0,1]` -> Output: `0`
    - **Approach**: Use a queue for students and a stack for sandwiches. Simulate the process: if `queue.front()` matches `stack.top()`, both dequeue/pop. If not, `queue.front()` is moved to the end of the queue. If, at any point, the number of students preferring the current `stack.top()` becomes zero, then the remaining students cannot eat.
- **Medium**: **Queue Reconstruction by Height**
    - **Problem**: You are given an array of people `people`, where `people[i] = [h_i, k_i]` represents the `i`-th person with height `h_i` and `k_i` number of people in front of this person who have a height greater than or equal to `h_i`. Reconstruct the queue.
    - **Example**: Input: `people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]` -> Output: `[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]`
    - **Approach**: This is a greedy problem. Sort the people: first by height in descending order, then by `k` in ascending order. Then, iterate through the sorted people and insert each person into a `list` (or `LinkedList` in Java, which behaves like a queue for insertion) at their `k`-th position. This works because when inserting, all people already in the list are taller or equal height, thus fulfilling the `k` requirement for later insertions.
- **Hard**: **Sliding Window Maximum**
    - **Problem**: You are given an array of integers `nums`, and a sliding window of size `k`. Return the maximum sliding window.
    - **Example**: Input: `nums = [1,3,-1,-3,5,3,6,7]`, `k = 3` -> Output: `[3,3,5,5,6,7]`
    - **Approach**: Use a **deque (double-ended queue)** to maintain indices of elements in the current window in

decreasing order of their values.

1. Add elements: For each `nums[i]`, remove elements from the back of the deque that are smaller than `nums[i]`. Then add `i` to the back.
2. Remove outdated elements: If `deque.front()` is outside the current window (`< i - k + 1`), remove it from the front.
3. The maximum for the current window is `nums[deque.front()]`.

Pattern 2: BFS (Breadth-First Search)

Queues are the fundamental data structure for implementing Breadth-First Search (BFS), an algorithm for traversing or searching tree or graph data structures. BFS explores all of the neighbor nodes at the present depth level before moving on to nodes at the next depth level.

- **Easy**: **Binary Tree Level Order Traversal**
  - **Problem**: Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).
  - **Example**: Input: `root = [3,9,20,null,null,15,7]` -> Output: `[[3],[9,20],[15,7]]`
  - **Approach**: Use a queue. Enqueue the root. While the queue is not empty:
    1. Get the `size` of the current level.
    2. Dequeue `size` number of nodes. Add their values to a `current_level_list`.
    3. For each dequeued node, enqueue its left child then its right child (if they exist).
    4. Add `current_level_list` to the final result.
- **Medium**: **Number of Islands**
  - **Problem**: Given an `m x n` 2D binary grid, which represents a map of '1's (land) and '0's (water), return the number of islands. An island is formed by connecting adjacent lands horizontally or vertically.
  - **Example**: Input: `grid = [["1","1","1","1","0"],["1","1","0","1","0"],["1","1","0","0","0"],["0","0","0","0","0"]]` -> Output: `1`
  - **Approach**: Iterate through the grid. When an unvisited '1' (land) is found:
    1. Increment the `island_count`.

2. Start a BFS from this cell: enqueue its coordinates.
3. While the queue is not empty, dequeue a cell, mark it as visited (e.g., change '1' to '0').
4. Enqueue all its valid (in-bounds, '1', unvisited) horizontal and vertical neighbors. This ensures all connected land is visited and not recounted.

- **Medium**: **Rotting Oranges**
  - **Problem**: You are given an `m x n` grid where each cell can be `0` (empty), `1` (fresh orange), or `2` (rotten orange). Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten. Return the minimum number of minutes that must elapse until no fresh oranges remain. If this is impossible, return `-1`.
  - **Example**: Input: `grid = [[2,1,1],[1,1,0],[0,1,1]]` -> Output: `4`
  - **Approach**: This is a multi-source BFS.
    1. Initialize a queue with all initially rotten oranges (multi-source). Count all fresh oranges.
    2. Perform a BFS. In each "minute" (level of BFS):
       - Dequeue all oranges from the current level.
       - For each dequeued orange, check its 4-directional neighbors. If a fresh orange is found, mark it as rotten, decrement `fresh_orange_count`, and enqueue it for the next minute.
    3. Increment a `minutes` counter for each level processed.
    4. After BFS, if `fresh_orange_count` is 0, return `minutes`. Otherwise, return `-1`.
- **Hard**: **Shortest Path in Binary Matrix**
  - **Problem**: Given an `n x n` binary matrix `grid`, return the length of the shortest clear path in the matrix. A clear path is a path from `(0, 0)` to `(n-1, n-1)` where all visited cells are `0` and are 8-directionally connected.
  - **Example**: Input: `grid = [[0,1],[1,0]]` -> Output: `2`
  - **Approach**: This is a classic BFS problem for shortest path on an unweighted grid.
    1. If `grid[0][0]` or `grid[n-1][n-1]` is 1, return -1 (start/end is blocked).
    2. Initialize a queue with `(0, 0)` and distance 1. Mark `(0, 0)` as visited (e.g., change `grid[0][0]` to 1).
    3. Perform BFS. When dequeuing a cell `(r, c)` with distance `d`:
       - If `(r, c)` is `(n-1, n-1)`, return `d`.
       - Explore all 8-directional neighbors. If a neighbor is valid (in-bounds, 0, not visited), mark it

visited, enqueue `(neighbor_r, neighbor_c)` with distance `d+1`.
4. If the queue becomes empty and the target is not reached, return -1.

Pattern 3: Priority Queue / Min-Heap (Advanced Queue Concepts)

While a standard queue is FIFO, a Priority Queue (often implemented using a Min-Heap or Max-Heap) is a conceptual queue where elements are dequeued based on their priority. This is crucial for problems requiring processing elements in a specific order other than arrival time.

- **Easy**: **Kth Largest Element in a Stream**
  - **Problem**: Design a class to find the `k`-th largest element in a stream.
  - **Example**: `KthLargest kthLargest = new KthLargest(3, [4,5,8,2]); kthLargest.add(3); // returns 4; kthLargest.add(5); // returns 5;`
  - **Approach**: Use a min-heap (priority queue). Initialize the heap with the first few elements. When `add(val)` is called, push `val`. If the heap size exceeds `k`, pop the smallest element. The `k`-th largest element is always at the top of the min-heap.
- **Medium**: **Meeting Rooms II**
  - **Problem**: Given an array of meeting time intervals `intervals` where `intervals[i] = [start_i, end_i]`, find the minimum number of conference rooms required.
  - **Example**: Input: `intervals = [[0, 30],[5, 10],[15, 20]]` -> Output: `2`
  - **Approach**: Sort the intervals by their start times. Use a min-heap (priority queue) to store the *end times* of currently occupied rooms.
    1. Add the end time of the first meeting to the heap.
    2. For subsequent meetings: if `start_time >= heap.top()` (the earliest ending meeting), it means a room is free; pop from the heap.
    3. Always add the current meeting's `end_time` to the heap.
    4. The size of the heap at any point represents the number of rooms needed, so the final `heap.size()` is the answer.

- **Hard**: **Find K Closest Elements**
    - **Problem**: Given a sorted integer array `arr`, two integers `k` and `x`, return the `k` closest integers to `x` in the array. The result should also be sorted in ascending order.
    - **Example**: Input: `arr = [1,2,3,4,5]`, `k = 4`, `x = 3` -> Output: `[1,2,3,4]`
    - **Approach**:
        1. A straightforward way is to use a max-heap (priority queue) of size `k`. Iterate through `arr`. For each `num`, add `(abs(num-x), num)` to the heap. If heap size exceeds `k`, pop.
        2. Alternatively, find `x`'s position using binary search. Then, use two pointers expanding outwards from `x` (or its closest element), adding elements to a result list. A `Deque` can also be used to add to front/back and then sort.

Pattern 4: Deque (Double-Ended Queue)

A deque is a double-ended queue that allows elements to be added or removed from both the front and the rear. It combines properties of both stacks and queues and is particularly useful for problems involving sliding windows or maintaining ordered subsets of elements.

- **Easy**: **Design Circular Deque**
    - **Problem**: Design your implementation of the circular double-ended queue (deque). It should support operations like `insertFront`, `insertLast`, `deleteFront`, `deleteLast`, `getFront`, `getRear`, `isEmpty`, `isFull`.
    - **Example**: Standard operations for a circular deque.
    - **Approach**: Implement using a fixed-size array and managing `front` and `rear` pointers with modulo arithmetic (`(index + 1) % capacity`).
- **Medium**: **Sliding Window Maximum** (revisited)
    - **Problem**: (Same as above) This problem is the quintessential application of a deque for maintaining a "monotonic" or "partially sorted" window.
    - **Approach**: (As described in Pattern 1, Sliding Window Maximum) The deque stores indices, and elements are added/removed from both ends to ensure the front always holds the index of the maximum element in

the current window.

- **Hard**: **Longest Subarray with Absolute Diff Less Than or Equal to Limit**
  - **Problem**: Given an array of integers `nums` and an integer `limit`, return the size of the longest non-empty subarray such that the absolute difference between any two elements in this subarray is less than or equal to `limit`.
  - **Example**: Input: `nums = [8,2,4,7]`, `limit = 4` -> Output: `2` (e.g., `[4,7]` -> `|4-7|=3 <= 4`)
  - **Approach**: Use a sliding window approach with two deques: one for maintaining a monotonic increasing sequence of elements' indices (to find the minimum in the window) and one for maintaining a monotonic decreasing sequence (to find the maximum in the window).
    1. Expand window: Add `nums[right]` to both deques, maintaining their monotonicity.
    2. Check condition: If `nums[max_deque.front()] - nums[min_deque.front()] > limit`, shrink the window from the left (`left++`), removing elements from deques if their index is `left`.
    3. Update answer: The current window length (`right - left + 1`) is a potential answer.

This guide provides a comprehensive overview of essential Queue topics and key interview patterns, along with illustrative examples for various difficulty levels. Mastering queues is fundamental for understanding graph traversals like BFS and for efficiently solving problems that require strict ordering of elements or dynamic window management. Consistent practice across these patterns will significantly improve your problem-solving abilities.

General Trees: Comprehensive Guide for SDE Interviews

A General Tree is a non-linear hierarchical data structure where each node can have an arbitrary number of children. Unlike binary trees which restrict children to at most two, general trees offer more flexibility in representing hierarchical relationships. While less common for direct implementation problems than binary trees, understanding general tree concepts is fundamental for grasping tree-based data structures in general, including file systems, organizational charts, and XML/JSON document structures.

1. Essential Topics to Know in General Trees

Before diving into problem patterns, ensure you have a solid grasp of these core general tree concepts:

- **Definition**: A tree is a collection of nodes, one of which is designated as the `root`, and the remaining nodes are partitioned into disjoint sets, each of which is a tree. Key characteristic: a hierarchical structure with no cycles.
- **Node Structure**: A node typically contains data and a list/array of pointers/references to its children.
- **Terminology (Revisited for General Trees)**:
    - **Root**: The unique topmost node.
    - **Parent/Child**: Direct hierarchical relationship.
    - **Sibling**: Nodes sharing the same parent.
    - **Leaf Node (External Node)**: A node with no children.
    - **Internal Node**: A node with at least one child.
    - **Edge**: The connection between a parent and a child.
    - **Path**: A sequence of connected nodes from one to another.
    - **Depth of a Node**: The number of edges from the root to the node. The root has depth 0.
    - **Height of a Node**: The number of edges on the longest path from the node to a leaf. A leaf node has height 0.
    - **Height of a Tree**: The height of its root node.
    - **Degree of a Node**: The number of children a node has.
- **Representation**: How general trees can be represented in memory:

    - **List of Children (Adjacency List-like)**: Each node stores a list (e.g., `ArrayList`, `vector`) of its children nodes. This is the most common and intuitive way.
    - **First-Child, Next-Sibling Representation**: A trick to represent any general tree as a binary tree. Each node has a pointer to its first child and a pointer to its next sibling. This can simplify algorithms by mapping general tree problems to binary tree problems.
- **Traversals**: While standard inorder/preorder/postorder are for binary trees, the concepts apply to general trees:
    - **Depth-First Traversal (DFT)**: Explore deeply before exploring siblings.
        - **Preorder**: Visit node, then recursively visit all its children.
        - **Postorder**: Recursively visit all its children, then visit node.
    - **Breadth-First Traversal (BFT) / Level Order Traversal**: Visit nodes level by level. Requires a queue.
- **Recursion**: General tree problems are naturally recursive, as a tree is defined recursively (a root and subtrees).
- **Edge Cases**: Handling empty trees (`null` root), single-node trees.

2. Questions Categorized by Patterns

Here are common patterns encountered in general tree-based SDE interview questions, focusing on the unique aspects where the arbitrary number of children plays a role.

Pattern 1: Basic Traversal and Information Gathering

These problems involve applying standard tree traversals (often BFS for level-based problems, or DFS for aggregated information) to general trees.

- **Easy**: **N-ary Tree Preorder Traversal**
    - **Problem**: Given the `root` of an N-ary tree, return the preorder traversal of its nodes' values.
    - **Example**: Input: N-ary tree with root 1, children [3,2,4], 3's children [5,6] -> Output: `[1,3,5,6,2,4]`
    - **Approach**: Recursive DFS. Visit the current node, then iterate through its `children` list and recursively call the traversal for each child.
- **Easy**: **N-ary Tree Level Order Traversal**
    - **Problem**: Given the `root` of an N-ary tree, return the level order traversal of its nodes' values.

    - **Example**: Input: N-ary tree with root 1, children [3,2,4], 3's children [5,6] -> Output: `[[1],[3,2,4],[5,6]]`
    - **Approach**: Use a queue (BFS). Enqueue the root. While the queue is not empty, process all nodes at the current level: dequeue, add to current level list, then for each dequeued node, iterate through its `children` and enqueue them. Add current level list to overall result.
- **Medium**: **Maximum Depth of N-ary Tree**
    - **Problem**: Given the `root` of an N-ary tree, return its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.
    - **Example**: Input: N-ary tree with root 1, children [3,2,4], 3's children [5,6] -> Output: `3`
    - **Approach**: Recursive. The depth of a node is 1 + maximum depth of its children. Base case: `if node is null, return 0`.
        - `maxDepth(node) = 1 + max(maxDepth(child) for each child in node.children)`
        - If a node has no children, its depth contribution is 1.
- **Hard**: **Clone N-ary Tree**
    - **Problem**: Given a reference of a node in a connected N-ary tree, return a deep copy (clone) of the tree.
    - **Example**: Input: N-ary tree structure.
    - **Approach**: Use recursive DFS or BFS. Create a new node for the current node, then recursively (or iteratively with queue) create and connect its children. A hash map is usually needed to store mappings from original nodes to cloned nodes to handle multiple references correctly (though N-ary trees usually don't have back-references to parents, they might appear multiple times in a serialization string if not careful).

Pattern 2: Tree Structure Transformation / Manipulation

Problems involving modifying the structure of a general tree, or converting it to another representation.

- **Easy**: **Reorder N-ary Tree (Custom Order)**
    - **Problem**: (Conceptual, not standard LeetCode) Given an N-ary tree, reorder the children of each node based on a specific criteria (e.g., sort children by value).
    - **Approach**: Perform a traversal (e.g., postorder DFS). At each node, sort its list of children based on the

given criteria.

- **Medium**: **N-ary Tree Postorder Traversal**
  - **Problem**: Given the `root` of an N-ary tree, return the postorder traversal of its nodes' values.
  - **Example**: Input: N-ary tree with root 1, children [3,2,4], 3's children [5,6] -> Output: `[5,6,3,2,4,1]`
  - **Approach**: Recursive DFS: Iterate through `node.children`, recursively call traversal for each child, then add `node.val`. Iterative: More complex, typically using a stack to manage the order of processing children before the parent.
- **Hard**: **Encode and Decode N-ary Tree**
  - **Problem**: Design an algorithm to encode an N-ary tree into a single string and then decode the string back into the original tree structure.
  - **Example**: Input: N-ary tree `root = [1,null,3,2,4,null,5,6]` -> Output: `[1,null,3,2,4,null,5,6]` (reconstructed tree)
  - **Approach**:
    - **Encode**: Use a traversal (e.g., preorder or level order). For each node, include its value. For an N-ary tree, you need a delimiter to indicate the end of a node's children list (e.g., a special marker like `null` or `#`). For example, `[1,3,5,null,6,null,2,null,4,null]` could encode `1` with children `3,2,4`, where `3` has children `5,6`. `null` signifies end of children for a node.
    - **Decode**: Use a queue (for level order) or stack (for preorder) and carefully parse the string using the delimiters to reconstruct the parent-child relationships.

Pattern 3: Tree Construction from Specific Data

Problems that involve building general trees from data that implicitly defines a hierarchical structure.

- **Easy**: **Convert Array to N-ary Tree (Explicit Parent Pointers)**
  - **Problem**: (Conceptual) Given an array of nodes, where each node has a value and a list of indices of its children in the array, construct the N-ary tree.
  - **Approach**: Iterate through the array to create all node objects first. Then, iterate again to link children based on the provided indices. Identify the root (node with no parent or specifically marked).

- **Medium**: **Deepest Node in N-ary Tree**
    - **Problem**: Find the deepest node(s) in an N-ary tree.
    - **Approach**: Use BFS (level order traversal). The last node(s) visited at the deepest level will be the deepest. Alternatively, recursive DFS: for each node, return its height; track the node(s) associated with the maximum height found.
- **Hard**: **Build N-ary Tree from Parent Array**
    - **Problem**: Given an array `parent` where `parent[i]` is the parent of node `i`. Assume `parent[0] = -1` (root). Construct the N-ary tree.
    - **Example**: `parent = [-1,0,0,1,1,2]` (node 0 is root, 1 & 2 are children of 0, 3 & 4 children of 1, 5 child of 2)
    - **Approach**:
        1. Create all nodes (e.g., `Node[] nodes = new Node[n]`).
        2. Iterate from `i = 0` to `n-1`. For each `i`, if `parent[i]` is not `-1`, find `nodes[parent[i]]` and add `nodes[i]` to its children list.
        3. The node at `nodes[0]` (or the one whose parent is -1) is the root.

This guide covers the essential theoretical foundations and practical patterns for working with General Trees. While specific questions might be less frequent than Binary Trees, understanding these concepts is crucial for any hierarchical data structure. Consistent practice across these patterns will significantly improve your problem-solving abilities.

Binary Tree: Comprehensive Guide for SDE Interviews

A Binary Tree is a specialized form of a tree data structure where each node has at most two children, typically referred to as the `left` child and the `right` child. Binary trees are extensively used in various computer science applications due to their simplicity and efficiency, appearing frequently in SDE interviews.

1. Essential Topics to Know in Binary Trees

Before diving into problem patterns, ensure you have a solid grasp of these core binary tree concepts:

- **Definition**: A tree in which each node has at most two children. These children are distinct, generally designated as the left child and the right child.
- **Node Structure**: A basic node typically contains:
    - `value` (or `data`)
    - `left` pointer/reference (to the left child)
    - `right` pointer/reference (to the right child)
- **Types of Binary Trees**:
    - **Full Binary Tree**: Every node has either 0 or 2 children.
    - **Complete Binary Tree**: All levels are completely filled except possibly the last level, which is filled from left to right.
    - **Perfect Binary Tree**: All internal nodes have two children and all leaf nodes are at the same depth.
    - **Skewed Binary Tree**: A tree where all nodes have only left children or only right children.
    - **Degenerate/Pathological Tree**: Each parent node has only one child. It behaves like a linked list.
- **Tree Terminologies**: (Revisited for Binary Trees, similar to General Trees)
    - **Root**: Topmost node.
    - **Parent/Child**: Direct relationship.
    - **Sibling**: Nodes with the same parent.
    - **Leaf Node**: Node with no children.
    - **Internal Node**: Node with at least one child.

    - **Edge**: Connection between nodes.
    - **Path**: Sequence of nodes.
    - **Depth of a Node**: Distance from the root.
    - **Height of a Node**: Longest path from the node to a leaf.
    - **Height of a Tree**: Height of its root.
- **Binary Tree Traversals**: These are fundamental for visiting all nodes in a binary tree.
    - **Depth-First Search (DFS)**:
        - **Preorder Traversal**: Root -> Left -> Right. (Useful for copying trees, expression trees).
        - **Inorder Traversal**: Left -> Root -> Right. (Produces sorted output for Binary Search Trees).
        - **Postorder Traversal**: Left -> Right -> Root. (Useful for deleting trees, evaluating expression trees).
    - **Breadth-First Search (BFS) / Level Order Traversal**: Visits nodes level by level from left to right. (Requires a Queue).
- **Recursion**: Binary tree problems are inherently recursive due to their recursive definition. Many solutions are elegant recursive functions.
- **Iterative Solutions**: While recursion is natural, iterative solutions using explicit stacks (for DFS) or queues (for BFS) are often required, especially to avoid stack overflow for deep trees or for specific traversal orders.
- **Edge Cases**: Handling `null` root (empty tree), single-node trees, trees with only left or only right children (skewed trees).

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in binary tree-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Tree Traversals (BFS & DFS)

Mastering both recursive and iterative implementations of standard binary tree traversals is critical. Many problems are direct applications or slight modifications of these.

- **Easy**: **Binary Tree Inorder Traversal**

    - **Problem**: Given the `root` of a binary tree, return the inorder traversal of its nodes' values.
    - **Example**: Input: `root = [1,null,2,3]` -> Output: `[1,3,2]`
    - **Approach (Recursive)**:
        1. If `node` is `null`, return.
        2. Recursively call for `node.left`.
        3. Add `node.val` to result list.
        4. Recursively call for `node.right`.
    - **Approach (Iterative)**: Use a stack. Push `node` and go `left` until `null`. Then `pop`, add `val` to list, and go `right`.
- **Easy**: **Binary Tree Level Order Traversal**
    - **Problem**: Given the `root` of a binary tree, return the level order traversal of its nodes' values.
    - **Example**: Input: `root = [3,9,20,null,null,15,7]` -> Output: `[[3],[9,20],[15,7]]`
    - **Approach**: Use a queue. Enqueue the `root`. While the queue is not empty:
        1. Get the `size` of the current level.
        2. Create a list for the current level's nodes.
        3. Loop `size` times: dequeue a node, add its value to `current_level_list`. Enqueue its `left` then `right` children (if they exist).
        4. Add `current_level_list` to the overall result.
- **Medium**: **Binary Tree Zigzag Level Order Traversal**
    - **Problem**: Given the `root` of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and so on).
    - **Example**: Input: `root = [3,9,20,null,null,15,7]` -> Output: `[[3],[20,9],[15,7]]`
    - **Approach**: Use BFS (queue). Keep track of the current `level_number` (or `direction` flag). When processing nodes for a level, if `level_number` is even, add nodes to the current level list normally. If odd, add them to the beginning of the list (or reverse the list after filling).
- **Hard**: **Iterative Postorder Traversal**
    - **Problem**: Implement a non-recursive postorder traversal for a binary tree.
    - **Example**: For `root = [1,2,3]` (Left, Right, Root) -> Output: `[2,3,1]`
    - **Approach**: This is typically done using two stacks. Push the `root` to `stack1`. While `stack1` is not empty, pop

`node` from `stack1` and push `node.val` to `stack2`. Then push `node.left` and `node.right` (if not null) to `stack1`. Finally, pop all elements from `stack2` to get the postorder sequence.

Pattern 2: Tree Properties / Calculations

Problems that involve calculating properties of a tree (height, depth, diameter, balance, sum of values) or checking structural characteristics. These often rely on recursive DFS, where information is passed up from children to parents.

- **Easy**: **Maximum Depth of Binary Tree**
  - **Problem**: Given the `root` of a binary tree, return its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.
  - **Example**: Input: `root = [3,9,20,null,null,15,7]` -> Output: `3`
  - **Approach (Recursive)**:
    1. Base case: `if node is null, return 0`.
    2. Recursive step: `return 1 + max(maxDepth(node.left), maxDepth(node.right))`.
- **Easy**: **Symmetric Tree**
  - **Problem**: Given the `root` of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).
  - **Example**: Input: `root = [1,2,2,3,4,4,3]` -> Output: `true`
  - **Approach**: Use a recursive helper function `isMirror(node1, node2)`.
    1. Base case: If both `node1` and `node2` are `null`, return `true`.
    2. Base case: If only one is `null` or their values don't match, return `false`.
    3. Recursive step: Return `isMirror(node1.left, node2.right) AND isMirror(node1.right, node2.left)`.
- **Medium**: **Balanced Binary Tree**
  - **Problem**: Given the `root` of a binary tree, determine if it is height-balanced. A height-balanced binary tree is one in which the left and right subtrees of every node differ in height by no more than one.
  - **Example**: Input: `root = [3,9,20,null,null,15,7]` -> Output: `true`
  - **Approach**: Use a recursive helper function that returns the height of the subtree rooted at the current node. If, at any point, the difference in heights of left and right children is greater than 1, propagate an

error/flag (e.g., return -1) upwards, indicating the tree is unbalanced. Otherwise, return `1 + max(left_height, right_height)`.

- **Medium**: **Binary Tree Maximum Path Sum**
  - **Problem**: A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. The path does not need to pass through the root. Return the maximum path sum of any non-empty path.
  - **Example**: Input: `root = [1,2,3]` -> Output: `6` (Path `2 -> 1 -> 3`)
  - **Approach**: Use a recursive DFS function that returns the maximum path sum *starting from the current node and going downwards* (unilateral path). Maintain a `global_max_sum` variable. At each node, calculate the path sum that *passes through* the current node (`node.val + max(0, left_unilateral_sum) + max(0, right_unilateral_sum)`) and update `global_max_sum` with this value. Return the unilateral path sum from this node: `node.val + max(0, left_unilateral_sum, right_unilateral_sum)`.
- **Hard**: **Binary Tree Diameter**
  - **Problem**: Given the `root` of a binary tree, return the length of the diameter of the tree. The diameter is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.
  - **Example**: Input: `root = [1,2,3,4,5]` -> Output: `3` (path 4-2-1-3 or 5-2-1-3)
  - **Approach**: Use a recursive DFS function that returns the height of the current subtree. Maintain a `global_max_diameter` variable. At each node, the potential diameter *passing through this node* is `height(left_subtree) + height(right_subtree)`. Update `global_max_diameter` with this value. The function returns `1 + max(height(left_subtree), height(right_subtree))` to its parent.

Pattern 3: Path Sum / Path Problems

Problems involving finding specific paths in a tree that satisfy a sum condition, or counting such paths. These often use recursive DFS combined with backtracking or maintaining state.

- **Easy**: **Path Sum**
  - **Problem**: Given the `root` of a binary tree and an integer `targetSum`, return `true` if the tree has a root-to-leaf

path such that adding up all the values along the path equals `targetSum`.
    - **Example**: Input: `root = [5,4,8,11,null,13,4,7,2,null,null,null,1]`, `targetSum = 22` -> Output: `true` (5->4->11->2)
    - **Approach**: Recursive DFS. For each node, subtract `node.val` from `targetSum`. If it's a leaf node and remaining `targetSum` is 0, return `true`. Otherwise, recurse for left or right child.
- **Medium**: **Path Sum II (Return all paths)**
    - **Problem**: Given the `root` of a binary tree and an integer `targetSum`, return all root-to-leaf paths where the sum of the node values in the path equals `targetSum`.
    - **Example**: Input: `root = [5,4,8,11,null,13,4,7,2,null,null,null,1]`, `targetSum = 22` -> Output: `[[5,4,11,2]]`
    - **Approach**: Use recursive DFS (backtracking). Maintain a `current_path` list. When a root-to-leaf path summing to `targetSum` is found, add a *copy* of `current_path` to the `result_list`. After each recursive call returns, backtrack by removing the current node from `current_path`.
- **Hard**: **Path Sum III (Any path)**
    - **Problem**: Given the `root` of a binary tree and an integer `targetSum`, return the number of paths that sum to `targetSum`. The path does not need to start or end at the root or a leaf, but it must be a downward path.
    - **Example**: Input: `root = [10,5,-3,3,2,null,11,3,-2,null,1]`, `targetSum = 8` -> Output: `3` (Paths: `5 -> 3`, `5 -> 2 -> 1`, `-3 -> 11`)
    - **Approach**: Use recursive DFS. Maintain a hash map (or dictionary) to store `prefix_sums` encountered along the current path from the root to the current node, and their frequencies. For each node, calculate `current_sum` from the root. Check if `(current_sum - targetSum)` exists in the `prefix_sums` map; if so, add its frequency to the total count. Increment the frequency of `current_sum` in the map. Recurse for children. After recursive calls return, `backtrack` by decrementing the frequency of `current_sum` in the map.

Pattern 4: Tree Construction / Reconstruction

Problems that involve building a binary tree from given input (e.g., arrays, traversal sequences) or copying a tree.

- **Easy**: **Convert Sorted Array to Binary Search Tree** (Though this is for BSTs, the approach is general)
    - **Problem**: Given an integer array `nums` where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.
    - **Example**: Input: `nums = [-10,-3,0,5,9]` -> Output: `[0,-3,9,-10,null,5]`

    - **Approach**: Use recursion. The middle element of the array becomes the root. Recursively build the left subtree from the left half of the array and the right subtree from the right half.
- **Medium**: **Construct Binary Tree from Preorder and Inorder Traversal**
    - **Problem**: Given two integer arrays `preorder` and `inorder` where `preorder` is the preorder traversal of a binary tree and `inorder` is the inorder traversal of the same tree, construct and return the binary tree.
    - **Example**: Input: `preorder = [3,9,20,15,7]`, `inorder = [9,3,15,20,7]` -> Output: `[3,9,20,null,null,15,7]`
    - **Approach**: Recursive. The first element in `preorder` is always the root. In `inorder`, find the position of this root. Elements to its left in `inorder` form the left subtree, and elements to its right form the right subtree. Recursively call for left and right subtrees using corresponding portions of `preorder` and `inorder` arrays. Use a hash map to quickly find the index of the root in `inorder`.
- **Hard**: **Serialize and Deserialize Binary Tree**
    - **Problem**: Design an algorithm to serialize (convert to a string) and deserialize (convert back from a string) a binary tree.
    - **Example**: Input: `root = [1,2,3,null,null,4,5]` -> Serialize: `"[1,2,null,null,3,4,null,null,5,null,null]"` (Example format), Deserialize: Reconstruct the tree from the string.
    - **Approach**:
        - **Serialization**: Use a traversal (e.g., preorder DFS or BFS level order). Store node values and `null` markers (e.g., "null" string, or a specific character). Use a delimiter between values.
        - **Deserialization**: Use a queue (for BFS) or a queue/list with an index (for preorder) to process the string. Reconstruct the tree based on the serialized string, processing values and nulls to correctly establish parent-child relationships.

Pattern 5: Lowest Common Ancestor (LCA)

Finding the lowest common ancestor of two nodes in a tree. For general binary trees (not necessarily BSTs), the approach is different from BSTs.

- **Easy**: **Lowest Common Ancestor of a Binary Search Tree** (Revisited, as it's a specific binary tree type, but

important to distinguish from general binary tree LCA)
    - **Problem**: Given a Binary Search Tree (BST), find the lowest common ancestor (LCA) of two given nodes `p` and `q`.
    - **Example**: Input: `root = [6,2,8,0,4,7,9,null,null,3,5]`, `p = 2`, `q = 8` -> Output: `6`
    - **Approach**: Utilize BST property. If both `p.val` and `q.val` are less than `root.val`, recurse `root.left`. If both are greater than `root.val`, recurse `root.right`. Otherwise, `root` is the LCA (because `p` and `q` are on opposite sides, or one of them is the root).
- **Medium**: **Lowest Common Ancestor of a Binary Tree**
    - **Problem**: Given a binary tree, find the lowest common ancestor (LCA) of two given nodes `p` and `q`.
    - **Example**: Input: `root = [3,5,1,6,2,0,8,null,null,7,4]`, `p = 5`, `q = 1` -> Output: `3`
    - **Approach**: Recursive DFS. The recursive function `findLCA(node, p, q)` returns the LCA in the subtree rooted at `node`.
        1. Base cases: If `node` is `null`, or `node == p`, or `node == q`, return `node`.
        2. Recursive calls: `left_result = findLCA(node.left, p, q)` and `right_result = findLCA(node.right, p, q)`.
        3. Combine results:
            - If both `left_result` and `right_result` are non-null, it means `p` and `q` are in different subtrees, so `node` is the LCA. Return `node`.
            - Else if `left_result` is non-null, return `left_result` (both `p` and `q` or only one of them is in the left subtree).
            - Else if `right_result` is non-null, return `right_result` (both `p` and `q` or only one of them is in the right subtree).
            - Else (both null), return `null`.
- **Hard**: **Lowest Common Ancestor of a Binary Tree III (With Parent Pointers)**
    - **Problem**: Given two nodes `p` and `q` in a binary tree, find their LCA. Each node has a `parent` pointer.
    - **Example**: `p = Node(5), q = Node(4)` -> Output: `Node(3)` (if 3 is parent of 5 and 4)
    - **Approach**:
        1. Traverse up from `p` to the root, storing all ancestors of `p` (including `p`) in a hash set.
        2. Traverse up from `q` to the root. The first ancestor of `q` that is found in the hash set is the LCA. This

takes time where H is the height of the tree.
3. Alternatively, find the heights (depths) of `p` and `q`. Align them by moving the deeper node up. Then move both nodes up simultaneously until they meet.

This comprehensive guide covers the essential theoretical foundations and practical patterns for working with Binary Trees. Mastering binary tree problems involves understanding their recursive nature, different traversal strategies, and how to combine these with various algorithmic techniques. Consistent practice across these patterns will significantly improve your problem-solving skills for SDE interviews.

Binary Search Tree (BST): Comprehensive Guide for SDE Interviews

A Binary Search Tree (BST) is a special type of Binary Tree where the nodes are arranged in a specific order that facilitates efficient searching, insertion, and deletion operations. For every node in a BST:

- All values in its **left subtree** are **less than** the node's value.
- All values in its **right subtree** are **greater than** the node's value.
- There are **no duplicate values** (though some definitions allow duplicates, the standard is unique values).

This ordered property makes BSTs a fundamental data structure for many applications, including implementing sets and maps, and they are very common in SDE interviews.

1. Essential Topics to Know in Binary Search Trees

Before diving into problem patterns, ensure you have a solid grasp of these core BST concepts:

- **Definition and Properties**: The fundamental left < root < right ordering property. Understanding how this property allows for efficient operations.
- **Node Structure**: Same as a Binary Tree node: `value`, `left` pointer, `right` pointer.
- **Core Operations (where H is height of tree)**:
    - **Search**: Efficiently finding a value.
    - **Insertion**: Adding a new node while maintaining BST property.
    - **Deletion**: Removing a node while maintaining BST property (most complex basic operation, involving handling 0, 1, or 2 children cases).
    - **Min/Max Element**: Finding the smallest (leftmost node) or largest (rightmost node) element.
- **Tree Traversals**:
    - **Inorder Traversal (Left -> Root -> Right)**: When applied to a BST, it produces elements in sorted (ascending) order. This is a very important property.
    - Preorder and Postorder traversals also apply.
    - Level Order Traversal (BFS) can be used.

- **Time Complexity**:
  - **Average Case**: O(logN) for search, insert, delete, min, max (if the tree is balanced).
  - **Worst Case**: O(N) for search, insert, delete, min, max (if the tree is skewed, behaving like a linked list).
- **Space Complexity**: O(H) for recursive calls (call stack), O(N) for storing the tree.
- **Recursion vs. Iteration**: Many BST operations can be implemented both recursively (often more concise) and iteratively (can avoid stack overflow for deep trees).
- **Successor and Predecessor**: Finding the next greater (successor) or next smaller (predecessor) element for a given node. Essential for deletion.
- **Edge Cases**: Handling empty BST, single-node BST, skewed BSTs, searching for non-existent values, deleting non-existent values.

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in Binary Search Tree-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Basic Operations & Property Checks

These problems involve implementing the core BST operations (search, insert, delete, find min/max) or verifying if a given tree adheres to the BST property.

- **Easy**: **Search in a Binary Search Tree**
  - **Problem**: Given the `root` of a BST and a `val`, find the node in the BST that the node's value equals `val` and return the subtree rooted with that node. If such a node does not exist, return `null`.
  - **Example**: Input: `root = [4,2,7,1,3]`, `val = 2` -> Output: `[2,1,3]` (subtree rooted at 2)
  - **Approach (Recursive)**:
    1. Base cases: If `root` is `null` or `root.val == val`, return `root`.
    2. Recursive step: If `val < root.val`, recursively search `root.left`. Else (`val > root.val`), recursively search `root.right`.
  - **Approach (Iterative)**: Use a `current` pointer starting from `root`. While `current` is not `null`: if `val == current.val`,

return `current`; if `val < current.val`, `current = current.left`; else `current = current.right`.

- **Easy**: **Minimum Absolute Difference in BST**
  - **Problem**: Given the `root` of a Binary Search Tree (BST), return the minimum absolute difference between the values of any two different nodes in the tree.
  - **Example**: Input: `root = [4,2,6,1,3]` -> Output: `1` (e.g., `|2-3|=1`, `|3-4|=1`)
  - **Approach**: The key property of a BST is that an inorder traversal yields nodes in sorted order. Perform an inorder traversal and keep track of the `previous_value` encountered. The minimum difference will be the minimum of `current_value - previous_value`.
- **Medium**: **Validate Binary Search Tree**
  - **Problem**: Given the `root` of a binary tree, determine if it is a valid Binary Search Tree (BST).
  - **Example**: Input: `root = [2,1,3]` -> Output: `true`
  - **Approach**: Use recursive DFS with `min_val` and `max_val` bounds. A helper function `isValidBST(node, min_bound, max_bound)` checks if `node.val` is strictly between `min_bound` and `max_bound`.
    1. Base case: If `node` is `null`, return `true`.
    2. Check current node: If `node.val <= min_bound` or `node.val >= max_bound`, return `false`.
    3. Recursive step: Return `isValidBST(node.left, min_bound, node.val) AND isValidBST(node.right, node.val, max_bound)`.
    4. Initial call: `isValidBST(root, -Infinity, Infinity)`.
- **Medium**: **Insert into a Binary Search Tree**
  - **Problem**: You are given the `root` of a BST and a `val` to insert. Return the root of the BST after the insertion.
  - **Example**: Input: `root = [4,2,7,1,3]`, `val = 5` -> Output: `[4,2,7,1,3,5]`
  - **Approach (Recursive)**: If `root` is `null`, create and return a new node with `val`. If `val < root.val`, `root.left = insertIntoBST(root.left, val)`. Else, `root.right = insertIntoBST(root.right, val)`. Return `root`.
  - **Approach (Iterative)**: Traverse to find the correct insertion point. Keep track of the `parent` node. Once `current` becomes `null`, attach the new node to the `parent`'s left or right child.
- **Hard**: **Delete Node in a BST**
  - **Problem**: Given the `root` of a BST and a `key`, delete the `key`-node in the BST and return the root of the BST.
  - **Example**: Input: `root = [5,3,6,2,4,null,7]`, `key = 3` -> Output: `[5,4,6,2,null,null,7]`

- **Approach (Recursive)**:
    1. If `root` is `null`, return `null`.
    2. If `key < root.val`, `root.left = deleteNode(root.left, key)`.
    3. If `key > root.val`, `root.right = deleteNode(root.right, key)`.
    4. If `key == root.val`:
        - Case 1: No children or one child (left or right). Return the existing child or `null`.
        - Case 2: Two children. Find the inorder successor (smallest node in the right subtree) or inorder predecessor (largest node in the left subtree). Replace `root.val` with the successor's `val`. Then, recursively call `deleteNode` on `root.right` to delete the successor node.

Pattern 2: Conversion & Reconstruction

Problems involving converting a BST to another data structure (e.g., sorted list, array) or constructing a BST from a sorted array/list.

- **Easy**: **Convert Sorted Array to Binary Search Tree**
    - **Problem**: Given an integer array `nums` where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.
    - **Example**: Input: `nums = [-10,-3,0,5,9]` -> Output: `[0,-3,9,-10,null,5]`
    - **Approach**: Recursive. The middle element of the array becomes the root. Recursively build the left subtree from the left half of the array and the right subtree from the right half. This ensures balance.
- **Medium**: **Convert Sorted List to Binary Search Tree**
    - **Problem**: Given the `head` of a singly linked list where elements are sorted in ascending order, convert it to a height-balanced binary search tree.
    - **Example**: Input: `head = [-10,-3,0,5,9]` -> Output: `[0,-3,9,-10,null,5]`
    - **Approach**: Similar to converting from a sorted array. You can first convert the list to an array, then use the array approach. More elegantly, use a slow/fast pointer to find the middle of the linked list for the root,

and recursively build left and right subtrees from the sub-lists. For O(N) time, a "bottom-up" recursive approach (similar to constructing from inorder traversal) is also possible where you build left subtree, then use current `head` as root, then build right subtree.

- **Hard**: **Flatten BST to a Single Linked List (In-place)**
  - **Problem**: Given the `root` of a BST, flatten it into a "linked list" in-place. The "linked list" should be in the same order as a pre-order traversal of the BST.
  - **Example**: Input: `root = [1,2,5,3,4,null,6]` -> Output: `[1,null,2,null,3,null,4,null,5,null,6]`
  - **Approach**: This problem typically requires careful pointer manipulation. One common approach involves finding the rightmost node of the left subtree and connecting it to the root's right child. Then, the root's left child becomes the new right child, and the original left child is set to null. Recursively apply this to the new right subtree. An alternative uses a `prev` pointer during a reverse preorder traversal to build the flattened list from right to left.

Pattern 3: Kth Smallest/Largest Element

Leveraging the BST property that inorder traversal yields sorted elements.

- **Easy**: **Kth Smallest Element in a BST**
  - **Problem**: Given the `root` of a BST, and an integer `k`, return the `k`-th smallest value (1-indexed) of all the values in the tree.
  - **Example**: Input: `root = [3,1,4,null,2]`, `k = 1` -> Output: `1`
  - **Approach**: Perform an inorder traversal (recursive or iterative). Maintain a counter. When the counter reaches `k`, the current node's value is the answer.
- **Medium**: **Find K-th Largest Element in BST**
  - **Problem**: Given the `root` of a BST and an integer `k`, return the `k`-th largest value.
  - **Approach**: Similar to Kth Smallest, but perform a reverse inorder traversal (Right -> Root -> Left). Maintain a counter and stop when it reaches `k`.
- **Hard**: **Closest K Elements to X in BST**
  - **Problem**: Given the `root` of a BST, a target value `x`, and an integer `k`, return the `k` values in the BST that are

closest to `x` (absolute difference).
    - **Example**: Input: `root = [4,2,5,1,3]`, `x = 3.7`, `k = 2` -> Output: `[3,4]`
    - **Approach**:
        1. Perform an inorder traversal to get all elements in a sorted list. Then use two pointers or a fixed-size window to find the `k` closest elements.
        2. More optimally: Use two stacks to simulate two-pointer traversal. One stack for elements smaller than `x`, another for elements larger than `x`. Then merge from both stacks, always taking the element closer to `x`. This avoids converting to a full list.

Pattern 4: LCA (Lowest Common Ancestor) in BSTs

Finding the lowest common ancestor of two nodes, specifically leveraging the BST property.

- **Easy**: **Lowest Common Ancestor of a Binary Search Tree**
    - **Problem**: Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes `p` and `q`.
    - **Example**: Input: `root = [6,2,8,0,4,7,9,null,null,3,5]`, `p = 2`, `q = 8` -> Output: `6`
    - **Approach (Recursive)**:
        1. Base case: If `root` is `null`, return `null`.
        2. If `p.val` and `q.val` are both less than `root.val`, recurse `root.left`.
        3. If `p.val` and `q.val` are both greater than `root.val`, recurse `root.right`.
        4. Otherwise, `root` is the LCA (because `p` and `q` are on opposite sides of `root`, or one of them is the `root` itself). Return `root`.
    - **Approach (Iterative)**: Start from `root`. While `current` is not `null`: if both `p` and `q` are smaller, move `current = current.left`. If both are larger, move `current = current.right`. Else, `current` is the LCA.

Pattern 5: Range / Interval Queries

Problems involving operations on elements within a specific range in a BST.

- **Easy**: **Range Sum of BST**
  - **Problem**: Given the `root` of a BST, and two integers `low` and `high`, return the sum of all node values `v` such that `low <= v <= high`.
  - **Example**: Input: `root = [10,5,15,3,7,null,18]`, `low = 7`, `high = 15` -> Output: `32` (7+10+15)
  - **Approach**: Use recursive DFS. If `node.val` is within the range `[low, high]`, add `node.val` to sum. Always recurse on the appropriate child: if `node.val > low`, recurse `node.left`; if `node.val < high`, recurse `node.right`. This prunes unnecessary branches.
- **Medium**: **Trim a Binary Search Tree**
  - **Problem**: Given the `root` of a binary search tree and two integers `low` and `high`, trim the tree such that all its elements lie in `[low, high]`. Return the root of the trimmed BST.
  - **Example**: Input: `root = [1,0,2]`, `low = 1`, `high = 2` -> Output: `[1,null,2]`
  - **Approach**: Recursive.
    1. Base case: If `root` is `null`, return `null`.
    2. If `root.val < low`, the current root and its left subtree are out of range. The new root must be in the right subtree. So, return `trimBST(root.right, low, high)`.
    3. If `root.val > high`, the current root and its right subtree are out of range. The new root must be in the left subtree. So, return `trimBST(root.left, low, high)`.
    4. If `low <= root.val <= high`, the current root is in range. Recursively trim its left and right subtrees: `root.left = trimBST(root.left, low, high)` and `root.right = trimBST(root.right, low, high)`. Then return `root`.

This comprehensive guide covers the essential theoretical foundations and practical patterns for working with Binary Search Trees. Mastering BST problems involves understanding their unique ordering property and how to leverage it for efficient operations, particularly search, insertion, deletion, and inorder traversal. Consistent practice across these patterns will significantly improve your problem-solving skills for SDE interviews.

Heap: Comprehensive Guide for SDE Interviews

A Heap is a specialized tree-based data structure that satisfies the **heap property**. It is typically implemented as a **complete binary tree**, which allows it to be efficiently stored and manipulated using an array. Heaps are crucial for implementing **Priority Queues** and are widely used in algorithms like Heap Sort, Dijkstra's algorithm, and Prim's algorithm.

1. Essential Topics to Know in Heaps

Before diving into problem patterns, ensure you have a solid grasp of these core Heap concepts:

- **Definition**: A heap is a complete binary tree that satisfies the heap property.
- **Heap Property**:
    - **Min-Heap**: For every node `i`, the value of `i` is less than or equal to the values of its children. The minimum element is always at the root.
    - **Max-Heap**: For every node `i`, the value of `i` is greater than or equal to the values of its children. The maximum element is always at the root.
- **Complete Binary Tree**: All levels are completely filled except possibly the last level, which is filled from left to right. This property is crucial because it allows efficient array-based implementation.
- **Array Representation**: How a complete binary tree (and thus a heap) is stored in an array:
    - If a node is at index `i`:
        1. Its left child is at index `2i + 1`.
        2. Its right child is at index `2i + 2`.
        3. Its parent is at index `(i - 1) / 2` (integer division).
    - The root is at index `0`.
- **Core Operations ()**:
    - **Insert (or `push` / `add`)**: Add a new element to the heap.
        1. Add the new element to the end of the array (maintaining completeness).
        2. Perform `heapify-up` (or "bubble up" / "swim"): repeatedly swap the new element with its parent if it

violates the heap property, until it reaches its correct position or the root.
    - **Delete Min/Max (or `pop` / `poll`)**: Remove the root element (min for min-heap, max for max-heap).
        1. Swap the root with the last element in the array.
        2. Remove the (now last) element.
        3. Perform `heapify-down` (or "bubble down" / "sink"): repeatedly swap the new root with its smallest/largest child (depending on heap type) if it violates the heap property, until it reaches its correct position or becomes a leaf.
    - **Peek Min/Max (or `top`)**: Return the root element without removing it. O(1) operation.
    - **Build Heap**: Construct a heap from an unsorted array in O(N) time by applying `heapify-down` from the last non-leaf node up to the root.
- **Time Complexity**:
    - Insert / Delete: O(logN) due to `heapify-up`/`heapify-down` operations (height of complete binary tree).
    - Peek: O(1).
    - Build Heap: O(N).
- **Space Complexity**: O(N) for storing elements in the array. O(logN) for recursive heapify if implemented recursively (typically iterative).
- **Applications**:
    - **Priority Queue**: Heaps are the most common way to implement priority queues, where elements are retrieved based on priority.
    - **Heap Sort**: An efficient, in-place sorting algorithm.
    - **Finding Kth Largest/Smallest Elements**: Using a min-heap or max-heap of size K.
    - **Graph Algorithms**: Dijkstra's algorithm (shortest path), Prim's algorithm (minimum spanning tree) use priority queues.
    - **Median of a Stream**: Using two heaps (a min-heap and a max-heap).
- **Standard Library Implementations**: Be aware of how heaps are implemented in common programming languages (e.g., `PriorityQueue` in Java, `heapq` module in Python, `std::priority_queue` in C++). These are usually min-heaps or max-heaps by default or configurable.

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in Heap-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Basic Heap Operations & Priority Queues

These problems directly involve using a heap (often as a priority queue) to manage elements based on their priority, or implementing heap operations from scratch.

- **Easy**: **Kth Largest Element in a Stream**
  - **Problem**: Design a class to find the `k`-th largest element in a stream.
  - **Example**: `KthLargest kthLargest = new KthLargest(3, [4,5,8,2]); kthLargest.add(3); // returns 4; kthLargest.add(5); // returns 5;`
  - **Approach**: Use a **min-heap** of size `k`. When `add(val)` is called, push `val` onto the heap. If the heap size exceeds `k`, pop the smallest element (which is at the root). The `k`-th largest element will always be the root of this min-heap.
- **Easy**: **Last Stone Weight**
  - **Problem**: You are given an array of integers `stones` where `stones[i]` is the weight of the `i`-th stone. On each turn, you choose the heaviest two stones and smash them. If `x == y`, both are destroyed. If `x != y`, the stone of weight `x` is destroyed, and the stone of weight `y - x` is left. Return the weight of the last remaining stone, or 0 if no stones are left.
  - **Example**: Input: `stones = [2,7,4,1,8,1]` -> Output: `1`
  - **Approach**: Use a **max-heap**. Push all stone weights into the max-heap. While the heap has more than one stone: pop the two largest stones (`y` then `x`). If `x != y`, push `y - x` back to the heap. Finally, return the remaining stone's weight (or 0 if heap is empty).
- **Medium**: **Top K Frequent Elements**
  - **Problem**: Given an integer array `nums` and an integer `k`, return the `k` most frequent elements.
  - **Example**: Input: `nums = [1,1,1,2,2,3]`, `k = 2` -> Output: `[1,2]`
  - **Approach**:

        - Use a hash map to count the frequencies of all elements in `nums`.
        - Use a **min-heap** to store `(frequency, element)` pairs. Iterate through the hash map. Push each `(freq, elem)` pair onto the min-heap.
        - If the heap's size exceeds `k` at any point, `pop` the smallest frequency pair.
        - After processing all elements, the heap will contain the `k` most frequent elements. Extract the elements and return.
- **Medium**: **Sort Characters By Frequency**
    - **Problem**: Given a string `s`, sort it in decreasing order based on the frequency of the characters.
    - **Example**: Input: `s = "tree"` -> Output: `"eert"` (or `"eetr"`)
    - **Approach**:
        - Use a hash map to count character frequencies.
        - Use a **max-heap** to store `(frequency, character)` pairs. Push all pairs from the hash map into the max-heap.
        - While the heap is not empty, pop the `(freq, char)` pair with the highest frequency. Append `char` to the result string `freq` times.
- **Hard**: **Find Median from Data Stream**
    - **Problem**: Design a `MedianFinder` class that supports `addNum(int num)` and `findMedian()`. `findMedian()` should return the median of all elements added so far.
    - **Example**: `MedianFinder mf = new MedianFinder(); mf.addNum(1); mf.addNum(2); mf.findMedian(); // returns 1.5; mf.addNum(3); mf.findMedian(); // returns 2.0;`
    - **Approach**: Use two heaps:
        - A **max-heap (**`max_left_half`**)** to store the smaller half of the numbers.
        - A min-heap (min_right_half) to store the larger half of the numbers.
          Maintain two properties:
        - `max_left_half.top() <= min_right_half.top()`
        - Sizes of heaps differ by at most 1.
          When addNum(num): Add num to max_left_half. Then, move max_left_half.top() to min_right_half.
          Rebalance if min_right_half becomes too big (move min_right_half.top() to max_left_half).

When findMedian(): If sizes are equal, median is (max_left_half.top() + min_right_half.top()) / 2. Else, it's the top of the larger heap.

Pattern 2: K-Way Merge / Merging Sorted Data

Heaps are excellent for efficiently merging multiple sorted lists or streams, or processing elements from multiple sources based on priority.

- **Easy**: **Merge K Sorted Lists**
  - **Problem**: Given an array of `k` linked lists, each sorted in ascending order, merge all the linked lists into one sorted linked list and return it.
  - **Example**: Input: `lists = [[1,4,5],[1,3,4],[2,6]]` -> Output: `[1,1,2,3,4,4,5,6]`
  - **Approach**: Use a **min-heap**. Push the first node of each linked list into the min-heap, storing `(node.val, node_reference)`. While the heap is not empty:
    1. Pop the smallest `(val, node)` from the heap.
    2. Append `node` to the result linked list.
    3. If `node` has a `next` element, push `(node.next.val, node.next)` into the heap.
- **Medium**: **Kth Smallest Element in a Sorted Matrix**
  - **Problem**: Given an `n x n` matrix where each row and column is sorted in ascending order, return the `k`-th smallest element in the matrix.
  - **Example**: Input: `matrix = [[1,5,9],[10,11,13],[12,13,15]]`, `k = 8` -> Output: `13`
  - **Approach**: Use a **min-heap**. Push the first element of each row `(matrix[i][0], row_index, col_index = 0)` into the heap. Then, `k-1` times, pop the smallest element. For each popped element `(val, r, c)`: if `c + 1 < n`, push `(matrix[r][c+1], r, c+1)` into the heap. The `k`-th popped element is the answer. (This is similar to `Merge K Sorted Lists`).
- **Hard**: **Smallest Range Covering Elements from K Lists**
  - **Problem**: You have `k` lists of sorted integers. Find the smallest range `[a,b]` that includes at least one number from each of the `k` lists.

- **Example**: Input: `nums = [[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]` -> Output: `[20,24]`
- **Approach**: Use a **min-heap**. Initially, add the first element from each of the `k` lists into the min-heap (store `(value, list_index, element_index_in_list)`). Keep track of the current `max_value` across elements in the heap.
    1. Initialize `current_max` to the maximum of initial elements.
    2. Initialize `min_range = 0`, `max_range = infinity`.
    3. While the heap contains `k` elements (one from each list):
        - Pop the smallest element `(min_val, list_idx, elem_idx)` from the heap.
        - If `current_max - min_val` is smaller than `max_range - min_range`, update `min_range = min_val`, `max_range = current_max`.
        - If there is a next element in `list_idx`, add `(next_val, list_idx, elem_idx + 1)` to the heap. Update `current_max` if `next_val` is larger.
        - If any list runs out of elements, stop.
    4. Return `[min_range, max_range]`.

Pattern 3: Graph Algorithms (with Priority Queues)

Heaps are the core data structure for efficient implementations of certain graph algorithms where edges (or nodes) need to be processed based on their "cost" or "priority".

- **Easy**: **Connecting Cities With Minimum Cost** (Conceptual, simpler Prim's/Kruskal's)
    - **Problem**: Given `n` cities and a list of `connections = [city1, city2, cost]`, find the minimum cost to connect all cities such that there is a path between any two cities. Return -1 if not all cities can be connected.
    - **Approach**: This is a Minimum Spanning Tree (MST) problem, solvable with Prim's or Kruskal's algorithm. Prim's algorithm uses a **min-heap** to efficiently select the next minimum-cost edge to add to the MST.
- **Medium**: **Network Delay Time** (Dijkstra's Algorithm)
    - **Problem**: You are given a network of `n` nodes, labeled from 1 to `n`. You are also given `times` as a list of travel `[u, v, w]` where `u` is the source node, `v` is the target node, and `w` is the time it takes to travel from `u` to `v`.

Calculate the minimum time it takes for all `n` nodes to receive the signal if the signal starts from `k`. If it's impossible for all nodes to receive the signal, return `-1`.
    - **Example**: Input: `times = [[2,1,1],[2,3,1],[3,4,1]]`, `n = 4`, `k = 2` -> Output: `2`
    - **Approach**: This is a Single-Source Shortest Path problem on a weighted graph, solvable with Dijkstra's algorithm. Use a **min-heap** to store `(time_to_reach_node, node_id)`.
        1. Initialize `dist[i] = infinity` for all nodes, `dist[k] = 0`.
        2. Push `(0, k)` into the min-heap.
        3. While heap is not empty: pop `(d, u)`. If `d > dist[u]`, continue (already found a shorter path). For each neighbor `v` of `u`: if `dist[u] + weight(u,v) < dist[v]`, update `dist[v]` and push `(dist[v], v)` to heap.
        4. The result is the maximum `dist` value among all nodes (if all nodes were visited).
- **Hard**: **Path with Maximum Probability** (Modified Dijkstra)
    - **Problem**: You are given an undirected graph with `n` nodes and `edges`. Each edge `[u,v]` has a `succProb` (probability of success). Find the path from `start` to `end` with the maximum success probability.
    - **Example**: Input: `n = 3, edges = [[0,1,0.5],[1,2,0.5],[0,2,0.2]]`, `start = 0, end = 2` -> Output: `0.25000` (path `0 -> 1 -> 2` with probability `0.5 * 0.5 = 0.25`)
    - **Approach**: This is a variation of Dijkstra's algorithm. Instead of minimizing sum of weights, you want to maximize product of probabilities. Use a **max-heap** to store `(probability_to_reach_node, node_id)`.
        1. Initialize `maxProb[i] = 0.0` for all nodes, `maxProb[start] = 1.0`.
        2. Push `(1.0, start)` into the max-heap.
        3. While heap is not empty: pop `(p, u)`. If `p < maxProb[u]`, continue. For each neighbor `v` of `u`: if `maxProb[u] * prob_uv > maxProb[v]`, update `maxProb[v]` and push `(maxProb[v], v)` to heap.
        4. Return `maxProb[end]`.

This comprehensive guide covers the essential theoretical foundations and practical patterns for working with Heaps. Mastering heap problems involves understanding the heap property, the array-based implementation, and the `heapify-up`/`heapify-down` operations. Heaps are crucial for priority queue applications and optimizing various graph and selection algorithms. Consistent practice across these patterns will significantly improve your problem-solving skills for

SDE interviews.

Trie (Prefix Tree): Comprehensive Guide for SDE Interviews

A Trie, also known as a Prefix Tree or Digital Tree, is a tree-like data structure used to store a dynamic set or associative array where keys are typically strings. Unlike a binary search tree, where nodes store the full key, in a Trie, each node represents a character or a prefix of a key. Traversal from the root to a node spells out the prefix represented by that node. Tries are highly efficient for operations involving prefixes, such as searching, inserting, and deleting strings, as well as for autocomplete and spell-checking functionalities.

1. Essential Topics to Know in Tries

Before diving into problem patterns, ensure you have a solid grasp of these core Trie concepts:

- **Definition**: A tree-like data structure used to store a collection of strings where common prefixes are shared among nodes.
- **Node Structure**:
    - Each node typically has an array (e.g., `children[26]` for lowercase English letters) or a hash map (for broader character sets) to store pointers to its child nodes.
    - A boolean flag (e.g., `isEndOfWord`) to indicate if a string ends at this node.
- **Root Node**: Represents an empty string.
- **Space Complexity**: Can be large if there are many unique characters at each level or very long strings. However, it can be much smaller than a hash table for datasets with many common prefixes.
- **Time Complexity**:
    - **Insert**: , where L is the length of the key (string).
    - **Search**: O(L), where L is the length of the key.
    - **Starts With (Prefix Search)**: O(P), where P is the length of the prefix.
- **Advantages**:
    - Fast lookup for prefixes and words (O(L) or O(P), independent of number of words).
    - Efficient for finding all words with a common prefix.
    - Lexicographical sorting of keys is implicit.

- **Disadvantages**:
  - Can consume a lot of memory, especially for a large alphabet or sparse string sets.
- **Key Operations**:
  - **Insertion**: Traverse the Trie based on characters of the string. Create new nodes for characters not present. Mark the final node as `isEndOfWord = true`.
  - **Search (Exact Word)**: Traverse the Trie. If any character path doesn't exist, the word isn't present. The final node must have `isEndOfWord = true`.
  - **Search (Prefix)**: Traverse the Trie. If the prefix path exists, the prefix is present. The `isEndOfWord` flag of the last node in the prefix path doesn't matter for `startsWith`.
  - **Deletion**: More complex. Can involve marking `isEndOfWord = false` or physically removing nodes if no other word uses that prefix.
- **Applications**:
  - Autocomplete and predictive text.
  - Spell checker.
  - IP routing (longest prefix match).
  - Dictionary search.
  - Browser history/URL caching.

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in Trie-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Basic Trie Implementation & Usage

These problems involve direct implementation of the Trie data structure and its fundamental operations (insert, search, startsWith).

- **Easy**: **Implement Trie (Prefix Tree)**

    - **Problem**: Implement a Trie (prefix tree) data structure with `insert`, `search`, and `startsWith` methods.
    - **Example**: `Trie trie = new Trie(); trie.insert("apple"); trie.search("apple"); // returns true; trie.startsWith("app"); // returns true; trie.search("app"); // returns false;`
    - **Approach**: Create a `TrieNode` class with an array/map of `children` and a boolean `isEndOfWord`. The `Trie` class will have a `root` node. Implement `insert` by traversing character by character, creating new nodes if a path doesn't exist. For `search` and `startsWith`, traverse and check existence and the `isEndOfWord` flag for `search`.
- **Medium**: **Design Add and Search Words Data Structure**
    - **Problem**: Design a data structure that supports adding new words and finding if a string matches any previously added word. The search string can contain dots `.` where `.` can represent any single letter.
    - **Example**: `WordDictionary wd = new WordDictionary(); wd.addWord("bad"); wd.addWord("dad"); wd.search(".ad"); // returns true; wd.search("b.."); // returns true;`
    - **Approach**: Use a Trie. `addWord` is standard Trie insertion. `search` becomes a recursive DFS. When a `.` is encountered in the search string, the `search` function recursively tries matching all possible children (a-z) of the current Trie node. If any path leads to a match, return `true`. This often involves backtracking.
- **Hard**: **Shortest Unique Prefix for Every Word**
    - **Problem**: Given a list of words, find the shortest unique prefix for every word. If a word has no unique prefix, return the word itself.
    - **Example**: Input: `["zebra", "dog", "duck", "dove"]` -> Output: `["z", "dog", "du", "dov"]`
    - **Approach**: Build a Trie. During insertion, or by a separate DFS/BFS traversal, store the frequency of how many words pass through each node. The shortest unique prefix for a word is found by traversing from the root until a node is reached whose `count` (number of words passing through it) is 1.

Pattern 2: Word Search / Dictionary Problems

Problems that involve finding words in a grid or using a dictionary (often implemented as a Trie) for efficient lookups.

- **Easy**: **Longest Word in Dictionary (by longest path in Trie)**
    - **Problem**: Given a list of strings `words`, find the longest word in `words` that can be built one character at a time by other words in `words`. If there is more than one such word, return the lexicographically smallest

one.
    - **Example**: Input: `words = ["w","wo","wor","worl","world"]` -> Output: `"world"`
    - **Approach**: Insert all words into a Trie. Then, perform a DFS on the Trie. Only traverse a path if the current node also marks the end of a valid word (i.e., `isEndOfWord` is true). Keep track of the longest word found, and handle lexicographical comparison.
- **Medium**: **Word Search II**
    - **Problem**: Given an `m x n` `board` of characters and a list of strings `words`, return all words on the `board` that can be formed by concatenating cells horizontally or vertically, where the same cell cannot be used more than once in a word.
    - **Example**: Input: `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"]`
    - **Approach**: Build a Trie from the `words` list. Then, iterate through each cell of the `board` and start a recursive Depth-First Search (DFS) from that cell. During the DFS, traverse the Trie simultaneously. If the current path of characters on the board matches a prefix in the Trie, continue the DFS. If a complete word is found in the Trie, add it to the results and mark it as found in the Trie (e.g., by setting a flag or deleting from Trie) to avoid duplicates. Use a `visited` array or modify board cells temporarily to avoid re-using cells in a single word.
- **Hard**: **Concatenated Words**
    - **Problem**: Given a list of words, return all words that are a concatenation of at least two shorter words in the given list.
    - **Example**: Input: `words = ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]` -> Output: `["catsdogcats","dogcatsdog","ratcatdogcat"]`
    - **Approach**: First, sort the words by length (shortest first). Insert all words into a Trie (or hash set). For each word, use dynamic programming (or a recursive helper function with memoization) to check if it can be formed by concatenating other words from the set/Trie. `dp[i]` is `true` if `s[0...i-1]` can be formed by concatenating words. The transition `dp[i] = dp[j] && word_in_trie(s[j...i-1])` helps.

Pattern 3: Bitwise Tries (Binary Tries)

A specialized form of Trie where branches are based on bits (0 or 1) of a number. Used for problems involving XOR sums or finding numbers with specific bit patterns.

- **Easy**: **Maximum XOR of Two Numbers in an Array**
  - **Problem**: Given an integer array `nums`, find the maximum result of `nums[i] XOR nums[j]`, where `0 <= i <= j < n`.
  - **Example**: Input: `nums = [3,10,5,25,2,8]` -> Output: `28` (5 XOR 25=28)
  - **Approach**: Build a binary Trie. Insert all numbers into the Trie by traversing their bits (from MSB to LSB). For each number `num` in the input array, try to find a number `other_num` in the Trie that maximizes `num XOR other_num`. To do this, for each bit of `num`, try to go the opposite path in the Trie (e.g., if `num` has a 0 bit, try to go to the 1-bit child in the Trie). If the opposite path exists, take it (as it maximizes XOR). If not, take the same path.
- **Medium**: **Maximum XOR of Two Numbers in an Array** (More generalized for specific bit length)
  - **Problem**: Same as above, but with an emphasis on handling numbers up to 231–1, meaning 31 bits.
  - **Approach**: Build a Trie where each node has `0` and `1` children. Iterate through `nums`. For each `num`, insert it into the Trie bit by bit from MSB to LSB. Then, for each `num`, iterate its bits and try to find a complementary path in the Trie that maximizes XOR. If `bit` is 0, try to go to `1` child; if `bit` is 1, try to go to `0` child. If preferred path exists, take it and add `2^k` to current XOR; else take same path. Update max XOR.
- **Hard**: **Maximum XOR Sum of a K-Length Path in a Tree** (Advanced, combines Tree + Trie)
  - **Problem**: (Conceptual, advanced variant) Given a tree with weighted edges, find a path of length `k` such that the XOR sum of all edge weights along the path is maximized.
  - **Approach**: This would likely involve a combination of DFS/BFS to explore paths and a Trie to efficiently query for maximum XOR sums. For example, during DFS, you could build a Trie of path XOR sums seen so far from the root to current node. Then, for each current node, query the Trie for a path XOR sum that maximizes the XOR with the current path sum from the root.

This comprehensive guide covers the essential theoretical foundations and practical patterns for working with Tries. Mastering Trie problems involves understanding their prefix-based structure, efficient insertion and search, and how to

apply them to various string-related and bitwise problems. Consistent practice across these patterns will significantly improve your problem-solving skills for SDE interviews.

Self-Balancing Binary Search Trees: Comprehensive Guide for SDE Interviews

Self-balancing Binary Search Trees (BSTs) are special types of binary search trees that automatically keep their height balanced. This balancing act ensures that operations like search, insertion, and deletion always complete in time, preventing the worst-case O(N) performance that can occur in unbalanced BSTs (like skewed trees). The two most common types are AVL Trees and Red-Black Trees.

1. Essential Topics to Know in Self-Balancing BSTs

Before diving into problem patterns, ensure you have a solid grasp of these core concepts:

- **Motivation for Self-Balancing**: Understand why regular BSTs can degrade to O(N) performance in worst-case scenarios (e.g., inserting sorted elements) and how self-balancing trees address this by maintaining a logarithmic height.
- **Rotations**: The fundamental operation used to rebalance a tree.
    - **Left Rotation**: Used when a node's right child subtree is too heavy.
    - **Right Rotation**: Used when a node's left child subtree is too heavy.
    - Understand how single (left-left, right-right) and double (left-right, right-left) rotations work to restore balance.
- **AVL Trees**:
    - **Definition**: A self-balancing binary search tree where for every node, the heights of its left and right subtrees differ by at most 1 (the "balance factor").
    - **Balance Factor**: `height(left_subtree) - height(right_subtree)`. Must be -1, 0, or 1.
    - **Insertion/Deletion**: After every insertion or deletion, the tree is rebalanced by performing rotations if any node's balance factor becomes invalid.
    - **Complexity**: O(logN) for search, insert, delete.
- **Red-Black Trees**:
    - **Definition**: A self-balancing binary search tree that maintains specific properties (color rules) to ensure balance. It's more complex but often preferred in practice (e.g., `std::map` in C++, `TreeMap` in Java).

    - **Red-Black Properties**:
        1. Every node is either Red or Black.
        2. The root is Black.
        3. Every leaf (NIL node) is Black.
        4. If a node is Red, then both its children are Black. (No two adjacent Red nodes).
        5. For each node, all simple paths from the node to descendant leaves contain the same number of Black nodes (Black-height property).
    - **Insertion/Deletion**: Involves coloring nodes and performing rotations to restore properties.
    - **Complexity**: O(logN) for search, insert, delete.
- **Comparison (AVL vs. Red-Black)**:
    - **Strictness**: AVL trees are more strictly balanced (height difference at most 1), leading to faster lookups.
    - **Rotation Count**: AVL trees may require more rotations during insertion/deletion to maintain their strict balance. Red-Black trees are less strictly balanced, generally requiring fewer rotations but having slightly slower lookups (larger height constant factor).
    - **Implementation Complexity**: Red-Black trees are generally considered more complex to implement from scratch.
- **Applications**:
    - Database indexing systems.
    - Sets and Maps in standard libraries.
    - Filesystems.
    - Associative arrays.

2. Questions Categorized by Patterns

While direct implementation of AVL or Red-Black trees from scratch is *rarely* asked in standard SDE interviews (due to their complexity), understanding their properties and how they guarantee O(logN) complexity is crucial. Interview questions often focus on problems where using a self-balancing BST (or knowing that one is needed) is the optimal

approach, or they might test specific properties.

Pattern 1: Understanding & Applying Self-Balancing Concepts

These problems don't necessarily require you to implement the full AVL/Red-Black tree, but rather to understand when and why they are useful, or to leverage their properties for efficient solutions.

- **Easy**: **Is AVL Balanced? (Check Balance Factor)**
  - **Problem**: (Conceptual) Given a binary tree, check if it satisfies the AVL tree balance property (height difference of left and right subtrees of every node is at most 1).
  - **Approach**: This is essentially the same as the "Balanced Binary Tree" problem for general binary trees. A recursive DFS function can return the height of a subtree, and also return a flag if it detects an imbalance.
- **Medium**: **Find Kth Smallest/Largest Element (with self-balancing awareness)**
  - **Problem**: Given a stream of numbers, find the `k`-th smallest element at any point. (This is a variation of Median Finder).
  - **Approach**: While two heaps (min-heap and max-heap) are common, a self-balancing BST could also solve this. Each node would store its value and the size of its subtree. `addNum` and `findKthSmallest` would involve BST insertion/search operations, augmented with subtree size updates and rotations to maintain balance. The `k`-th smallest element would be found by traversing left/right based on subtree sizes.
- **Hard**: **Range Sum Query - Data Structure Design (supporting updates)**
  - **Problem**: Design a data structure that supports `sumRange(i, j)` (sum of elements in a range) and `update(idx, val)` (update element at index `idx`) operations efficiently. Assume the array is initially populated.
  - **Approach**: For static arrays, prefix sums work for `sumRange`. For updates, a Fenwick tree (BIT) or Segment Tree is standard. However, a self-balancing BST (like a Treap or Augmented Red-Black Tree) could also solve this. Each node would store its own value and the sum of values in its subtree. Updates would trigger rebalancing, and range sums would involve traversing the tree and summing up relevant nodes. This problem often points to Segment Trees or Fenwick Trees as optimal, but a balanced BST can achieve similar logarithmic complexity.

Pattern 2: Problems Solved Best by Self-Balancing BSTs (Implicit Use)

These problems are best solved using a self-balancing BST implementation, even if you just use a standard library's `TreeSet` or `TreeMap` (which are typically implemented with Red-Black Trees). The focus is on recognizing when O(logN) insertion/deletion/lookup is required for sorted data that changes frequently.

- **Easy**: **Two Sum (with updates/removals)**
  - **Problem**: (Conceptual variant) Design a data structure that allows you to `add` numbers and efficiently check if any two numbers `sum` up to a target, where numbers can be added and removed frequently.
  - **Approach**: A `HashSet` provides O(1) average time for `add` and `lookup`. However, if you need to maintain sorted order or find a specific K-th element (which a hash set doesn't easily do), a balanced BST could be considered. For `Two Sum`, a hash set is usually sufficient and simpler.
- **Medium**: **Meeting Rooms III (Advanced Scheduling)**
  - **Problem**: Given a list of meeting intervals `[start, end]` and `n` rooms, return the earliest time a room becomes free after a series of meetings have been scheduled. (A variant of Meeting Rooms II).
  - **Approach**: Sort meetings by start time. Use a min-heap for available rooms (storing their `free_time`). To find the earliest available room, a min-heap works. However, if rooms have different "types" or specific properties you need to search for *efficiently* in a sorted manner, a balanced BST could be used to manage room availability by their free times.
- **Hard**: **Count of Smaller Numbers After Self**
  - **Problem**: Given an integer array `nums`, return an array `answer` where `answer[i]` is the number of smaller elements to the right of `nums[i]`.
  - **Example**: Input: `nums = [5,2,6,1]` -> Output: `[2,1,1,0]`
  - **Approach**: Iterate through `nums` from right to left. Maintain a self-balancing BST (like an AVL or Red-Black Tree). When inserting `nums[i]`, query the BST for the count of elements smaller than `nums[i]`. The BST can be augmented to store subtree sizes, making this query O(logN). This is a common advanced application where a balanced BST excels.

Pattern 3: Rotations and Balance Factor Logic (Less Common, More Theoretical)

These problems are more theoretical or involve specific understanding of the rebalancing mechanisms (rotations) rather than just using a pre-built balanced BST. These are very rare in typical SDE interviews unless the role is specifically for low-level data structure design.

- **Easy**: **Implement Single Left/Right Rotation**
  - **Problem**: (Conceptual) Given a root of a subtree that is unbalanced due to a left-left or right-right case, implement a single left or right rotation to rebalance it.
  - **Approach**: Understand the pointer changes involved. For a left rotation on `A` (where `B` is `A`'s right child), `A` becomes `B`'s left child, and `B`'s original left child becomes `A`'s right child.
- **Medium**: **Implement Double Rotations (Left-Right, Right-Left)**
  - **Problem**: (Conceptual) Given a root of a subtree unbalanced due to a left-right or right-left case, implement the double rotation (which is a sequence of two single rotations) to rebalance it.
  - **Approach**: This involves two single rotations. E.g., for Left-Right, first a left rotation on the left child, then a right rotation on the root.
- **Hard**: **AVL Tree Insertion (Full Implementation Sketch)**
  - **Problem**: (Highly unlikely in a typical interview, but for completeness) Sketch or explain the algorithm for inserting a node into an AVL tree, including rebalancing with rotations.
  - **Approach**: Recursively insert the node like in a regular BST. On returning from recursive calls, update heights and balance factors. If any node's balance factor becomes `>1` or `< -1`, identify the type of imbalance (LL, RR, LR, RL) and apply the corresponding single or double rotation.

This guide covers the essential theoretical foundations and practical patterns for working with Self-Balancing Binary Search Trees. While full implementations are rare, understanding their properties and when to leverage them for logarithmic time complexity in scenarios with frequent insertions/deletions/lookups is crucial for SDE interviews. Consistent practice with problems that benefit from their balanced nature will significantly improve your problem-solving abilities.

Graphs: Comprehensive Guide for SDE Interviews

A Graph is a non-linear data structure consisting of a set of nodes (or vertices) and a set of edges (or arcs) that connect these nodes. Graphs are powerful tools for modeling relationships between entities and are used extensively in various domains, including social networks, mapping and navigation systems, computer networks, and more. Understanding graph algorithms is fundamental for SDE interviews.

1. Essential Topics to Know in Graphs

Before diving into problem patterns, ensure you have a solid grasp of these core graph concepts:

- **Definition**: A graph G=(V,E) consists of a set of vertices and a set of edges E.
- **Terminology**:
  - **Vertex (Node)**: A point or a node in the graph.
  - **Edge**: A connection between two vertices.
  - **Adjacent Vertices**: Two vertices connected by an edge.
  - **Degree of a Vertex**: The number of edges incident to a vertex (for undirected graphs).
    - **In-degree**: Number of incoming edges (for directed graphs).
    - **Out-degree**: Number of outgoing edges (for directed graphs).
  - **Path**: A sequence of distinct vertices where each consecutive pair is connected by an edge.
  - **Cycle**: A path that starts and ends at the same vertex.
  - **Connected Graph**: A graph where there is a path between every pair of vertices.
  - **Disconnected Graph**: A graph that is not connected.
  - **Strongly Connected Graph (Directed)**: A directed graph where there is a path from every vertex to every other vertex.
  - **Component**: A maximal connected subgraph.
- **Types of Graphs**:
  - **Undirected Graph**: Edges have no direction (connection is bidirectional).
  - **Directed Graph (Digraph)**: Edges have a direction (connection is unidirectional).

    - **Weighted Graph**: Edges have associated numerical values (weights or costs).
    - **Unweighted Graph**: Edges have no associated weights.
    - **Cyclic Graph**: Contains at least one cycle.
    - **Acyclic Graph**: Contains no cycles. A Directed Acyclic Graph (DAG) is particularly important.
    - **Dense Graph**: Number of edges E is close to V2.
    - **Sparse Graph**: Number of edges E is much less than V2.
- **Graph Representations**: How graphs are stored in memory:
    - **Adjacency Matrix**: A 2D array `adj[V][V]` where `adj[i][j] = 1` (or weight) if an edge exists from `i` to `j`, else 0.
        - Pros: Fast edge lookup (O(1)), easy to add/remove edges.
        - Cons: High space complexity (O(V2)), slow to iterate neighbors.
    - **Adjacency List**: An array of lists where `adj[i]` contains a list of vertices adjacent to vertex `i`.
        - Pros: Space efficient (O(V+E)), fast to iterate neighbors.
        - Cons: Slower edge lookup (O(degree(V))).
    - **Edge List**: A list of all edges, where each edge is represented as a pair `(u, v)` (and possibly weight).
- **Graph Traversal Algorithms**:
    - **Breadth-First Search (BFS)**: Explores level by level. Uses a **Queue**. Finds shortest path in unweighted graphs.
    - **Depth-First Search (DFS)**: Explores as far as possible along each branch before backtracking. Uses a **Stack** (explicit or implicit via recursion).
- **Time and Space Complexity Analysis**: For graph algorithms, often expressed in terms of V (number of vertices) and E (number of edges).
- **Cycle Detection**: How to detect cycles in both directed and undirected graphs (using DFS/BFS).

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in graph-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Graph Traversal (BFS & DFS)

These are the foundational algorithms for exploring graphs. BFS is typically used for shortest path on unweighted graphs and level-by-level exploration. DFS is used for finding paths, cycles, and exploring all reachable nodes from a starting point, often in grid-based problems.

- **Easy**: **Number of Islands**
  - **Problem**: Given an `m x n` 2D binary grid, which represents a map of '1's (land) and '0's (water), return the number of islands. An island is formed by connecting adjacent lands horizontally or vertically.
  - **Example**: Input: `grid = [["1","1","1","1","0"],["1","1","0","1","0"],["1","1","0","0","0"],["0","0","0","0","0"]]` -> Output: `1`
  - **Approach (BFS or DFS)**: Iterate through the grid. When an unvisited '1' (land) is found, increment `island_count` and start a BFS/DFS from that cell. The traversal function will visit all connected '1's, marking them as visited (e.g., by changing their value to '0') to prevent recounting.
- **Easy**: **Find if Path Exists in Graph**
  - **Problem**: Given the number of vertices `n`, a list of `edges`, and two vertices `source` and `destination`, return `true` if there is a valid path from `source` to `destination`, and `false` otherwise.
  - **Example**: Input: `n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2` -> Output: `true`
  - **Approach (BFS or DFS)**: Convert edge list to adjacency list. Start a BFS or DFS from `source`. Use a `visited` set to avoid cycles and redundant work. If `destination` is visited during the traversal, return `true`. If traversal completes and `destination` is not visited, return `false`.
- **Medium**: **Rotting Oranges**
  - **Problem**: You are given an `m x n` grid where each cell can be `0` (empty), `1` (fresh orange), or `2` (rotten orange). Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten. Return the minimum number of minutes that must elapse until no fresh oranges remain. If this is impossible, return `-1`.
  - **Example**: Input: `grid = [[2,1,1],[1,1,0],[0,1,1]]` -> Output: `4`
  - **Approach (Multi-source BFS)**: This is a BFS on a grid, representing time. Initialize a queue with all initially rotten oranges. Perform BFS level by level. In each minute, dequeue all oranges from the current level, mark their fresh neighbors as rotten, and enqueue them for the next minute.

- **Medium**: **Clone Graph**
    - **Problem**: Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph.
    - **Example**: Input: A graph node structure.
    - **Approach (BFS or DFS)**: Use BFS or DFS to traverse the original graph. While traversing, create new nodes for each original node and store a mapping from original node to cloned node (e.g., in a hash map). When processing an original node's neighbors, link them to the corresponding cloned neighbors.
- **Hard**: **Word Ladder**
    - **Problem**: Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return the length of the shortest transformation sequence from `beginWord` to `endWord` such that only one letter can be changed at a time, and each transformed word must exist in the `wordList`.
    - **Example**: Input: `beginWord = "hit"`, `endWord = "cog"`, `wordList = ["hot","dot","dog","lot","log","cog"]` -> Output: `5` (hit -> hot -> dot -> dog -> cog)
    - **Approach (BFS)**: This is a shortest path problem on an unweighted graph where words are nodes and an edge exists between words differing by one character. Use BFS. Create neighbor relationships (e.g., by changing one char at a time and checking `wordList`). Start BFS from `beginWord` and maintain distance.

Pattern 2: Shortest Path Algorithms (Weighted Graphs)

For graphs with weighted edges, specialized algorithms are needed to find the shortest path.

- **Easy**: **Dijkstra's Algorithm (Conceptual/Simple Case)**
    - **Problem**: Given a small, simple weighted directed graph and a source node, find the shortest path to all other nodes.
    - **Approach**: Explain the core idea of Dijkstra's: greedily selecting the unvisited node with the smallest known distance from the source, and updating distances of its neighbors. This often uses a `min-priority queue`.
- **Medium**: **Network Delay Time (Dijkstra's Application)**
    - **Problem**: You are given a network of `n` nodes and `times` as a list of travel `[u, v, w]` (source, target, time).

Calculate the minimum time it takes for all `n` nodes to receive the signal if it starts from `k`. Return -1 if impossible.
    - **Example**: Input: `times = [[2,1,1],[2,3,1],[3,4,1]]`, `n = 4`, `k = 2` -> Output: `2`
    - **Approach (Dijkstra's Algorithm)**: Use a **min-priority queue** to store `(time_to_reach_node, node_id)`. Initialize distances to infinity, source to 0. Repeatedly extract the node with the smallest current distance and relax its neighbors.
- **Hard**: **Cheapest Flights Within K Stops**
    - **Problem**: There are `n` cities connected by `flights`. Find the cheapest price from `src` to `dst` with at most `k` stops. If no such route, return `-1`.
    - **Example**: Input: `n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]]`, `src = 0, dst = 2, k = 1` -> Output: `200`
    - **Approach (BFS with limited depth or Bellman-Ford variant)**: Can be solved with a modified BFS (where `stops_made` is part of the queue state) or a Bellman-Ford-like DP approach (`dp[stops][city]` = min cost to reach `city` with `stops`).
- **Hard**: **All-Pairs Shortest Path (Floyd-Warshall)**
    - **Problem**: Given a weighted graph, find the shortest paths between all pairs of vertices. Weights can be negative, but no negative cycles.
    - **Example**: Input: Graph represented by an adjacency matrix with weights.
    - **Approach (Floyd-Warshall Algorithm)**: This is a Dynamic Programming algorithm. It uses a 2D array `dist[i][j]` to store the shortest distance from vertex `i` to `j`. The algorithm iterates through all possible intermediate vertices `k` and updates `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`.
        - **Complexity**: O(V3).
        - **Use Case**: When you need all-pairs shortest paths, especially for dense graphs or when negative weights are present (but no negative cycles).

Pattern 3: Minimum Spanning Trees (MST)

For weighted, undirected graphs, an MST is a subset of the edges that connects all the vertices together, without any

cycles and with the minimum possible total edge weight.

- **Easy**: **Connecting Cities With Minimum Cost (Prim's/Kruskal's Basic Idea)**
  - **Problem**: Given `n` cities and a list of `connections = [city1, city2, cost]`, find the minimum cost to connect all cities such that there is a path between any two cities. Return -1 if not all cities can be connected.
  - **Approach**: This is a direct application of MST. Explain the concept of building an MST.
- **Medium**: **Min Cost to Connect All Points (Kruskal's / Prim's)**
  - **Problem**: You are given an array of points representing coordinates. The cost of connecting two points is their Manhattan distance. Find the minimum cost to connect all points.
  - **Example**: Input: `points = [[0,0],[2,2],[3,10],[5,2],[7,0]]` -> Output: `20`
  - **Approach (Kruskal's Algorithm)**:
    1. Generate all possible edges between points with their Manhattan distances.
    2. Sort all edges by weight in ascending order.
    3. Use Union-Find to process edges. Iterate through sorted edges: if `u` and `v` are not already connected (i.e., `find(u) != find(v)`), add the edge to MST, `union(u, v)`. Stop when `n-1` edges are added.
  - **Approach (Prim's Algorithm)**:
    1. Start with any arbitrary point. Maintain a min-priority queue of edges connected to the MST-built part.
    2. Repeatedly extract the minimum-weight edge from the priority queue that connects to a new vertex. Add the vertex and its connected edges to the priority queue.
- **Hard**: **Graph Valid Tree (Check for MST properties)**
  - **Problem**: Given `n` nodes and a list of undirected `edges`, determine if these edges form a valid tree.
  - **Example**: Input: `n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]` -> Output: `true`
  - **Approach**: A graph is a valid tree if and only if it has exactly `n-1` edges and no cycles. Use Union-Find (Pattern 4) to check for cycles while counting edges. If `n-1` edges are processed without a cycle, and all nodes are connected (one component), it's a tree.

Pattern 4: Connectivity (Bridges, Articulation Points, SCCs)

These patterns involve identifying specific types of critical points or subgraphs in a graph that define its connectivity.

- **Easy**: **Number of Connected Components in an Undirected Graph**
  - **Problem**: Given `n` nodes and `edges` representing connections in an undirected graph, return the number of connected components.
  - **Example**: Input: `n = 5, edges = [[0,1],[1,2],[3,4]]` -> Output: `2`
  - **Approach (BFS/DFS or Union-Find)**: Either use BFS/DFS: iterate through nodes, start traversal from unvisited nodes, increment count. Or use Union-Find: initialize `n` components, `union` for each edge, final count of distinct roots is answer.
- **Medium**: **Critical Connections in a Network (Bridges)**
  - **Problem**: There are `n` servers numbered from `0` to `n-1` connected by `n-1` connections (edges). You are given a list of `connections` representing the connections. Return all critical connections (bridges) in the network. A connection is critical if removing it would make some server unable to reach some other server.
  - **Example**: Input: `n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]` -> Output: `[[1,3]]`
  - **Approach (Tarjan's/Bridge Algorithm)**: Use DFS. Maintain `discovery_time[u]` (time when `u` is first visited) and `low_link[u]` (lowest discovery time reachable from `u`'s subtree, including `u` itself, through at most one back-edge). An edge `(u, v)` is a bridge if `low_link[v] > discovery_time[u]`.
- **Hard**: **Articulation Points (Cut Vertices)**
  - **Problem**: (Conceptual) Given an undirected graph, find all articulation points (cut vertices). An articulation point is a vertex whose removal increases the number of connected components.
  - **Approach (Tarjan's/Articulation Point Algorithm)**: Similar to Bridge Algorithm. For a non-root vertex `u`, if it has a child `v` such that `low_link[v] >= discovery_time[u]`, then `u` is an articulation point. For the root, it's an articulation point if it has more than one child in the DFS tree.
- **Hard**: **Strongly Connected Components (SCCs)**
  - **Problem**: (Conceptual) Given a directed graph, find all its Strongly Connected Components (SCCs). An SCC is a maximal subgraph such that for every pair of vertices `u` and `v` in the subgraph, there is a path from `u` to `v` and from `v` to `u`.
  - **Approach (Kosaraju's Algorithm / Tarjan's SCC Algorithm)**: These are advanced algorithms.
    - **Kosaraju's**:

1. Perform DFS on original graph to get finishing times of all nodes (stored on a stack).
2. Compute the transpose graph (reverse all edges).
3. Perform DFS on the transpose graph in decreasing order of finishing times (from step 1). Each DFS tree in this second traversal is an SCC.

Pattern 5: Topological Sort (DAGs)

For Directed Acyclic Graphs (DAGs), topological sort produces a linear ordering of vertices such that for every directed edge `u -> v`, `u` comes before `v` in the ordering. Used for task scheduling, dependency resolution.

- **Easy**: **Course Schedule (Cycle Detection in DFS)**
  - **Problem**: There are `n` courses you have to take. Given `numCourses` and `prerequisites` `[course, prerequisite]`, return `true` if you can finish all courses.
  - **Example**: Input: `numCourses = 2, prerequisites = [[1,0]]` -> Output: `true`
  - **Approach (DFS with cycle detection)**: Build adjacency list. Use DFS and maintain three states for each node: `unvisited`, `visiting` (in current recursion stack), `visited` (finished processing). If DFS encounters a `visiting` node, a cycle is detected.
- **Medium**: **Course Schedule II (Return Order)**
  - **Problem**: Same as Course Schedule I, but return the order of courses you should take to finish all courses. If impossible, return an empty array.
  - **Example**: Input: `numCourses = 2, prerequisites = [[1,0]]` -> Output: `[0,1]`
  - **Approach (Kahn's Algorithm - BFS)**:
    1. Calculate in-degrees for all nodes.
    2. Enqueue all nodes with in-degree 0.
    3. While queue is not empty: dequeue node `u`, add to result. For each neighbor `v` of `u`: decrement `in_degree[v]`. If `in_degree[v]` becomes 0, enqueue `v`.
    4. If total nodes in result equals `numCourses`, return result. Else, return empty.

- **Hard**: **Alien Dictionary**
    - **Problem**: Given a list of words from an alien dictionary, sorted lexicographically, return a string of the unique letters sorted in lexicographical order. If no order can be determined, return an empty string.
    - **Example**: Input: `words = ["wrt","wrf","er","ett","rftt"]` -> Output: `"wertf"`
    - **Approach (Topological Sort)**: Build a directed graph where an edge `u -> v` exists if `u` must come before `v` based on word comparisons. Then apply topological sort (Kahn's or DFS-based) to this graph. If a cycle is detected, no valid order exists.

Pattern 6: Union-Find (Disjoint Set Union)

Used for efficiently determining connected components and handling dynamic connectivity (merging sets). Often used in MST algorithms like Kruskal's.

- **Easy**: **Number of Connected Components in an Undirected Graph**
    - **Problem**: Given `n` nodes and `edges` representing connections in an undirected graph, return the number of connected components.
    - **Example**: Input: `n = 5, edges = [[0,1],[1,2],[3,4]]` -> Output: `2`
    - **Approach (Union-Find)**: Initialize each node as its own parent (n components). For each `edge (u, v)`, perform `union(u, v)`. The number of distinct `roots` (or components) at the end is the answer. Path compression and union by rank/size optimize performance.
- **Medium**: **Redundant Connection**
    - **Problem**: In an undirected graph with `n` nodes, `n` edges are added one by one. Return the edge that can be removed so that the resulting graph is a tree (i.e., it contains `n-1` edges and no cycles). There is exactly one such edge.
    - **Example**: Input: `edges = [[1,2],[1,3],[2,3]]` -> Output: `[2,3]`
    - **Approach (Union-Find)**: Iterate through the edges. For each edge `(u, v)`: perform `find(u)` and `find(v)`. If `find(u) == find(v)`, it means `u` and `v` are already connected, so adding `(u, v)` creates a cycle. This is the redundant edge. Return it. Otherwise, `union(u, v)`.
- **Hard**: **Longest Consecutive Sequence (using Union-Find for connectivity)**

    - **Problem**: Given an unsorted array of integers `nums`, return the length of the longest consecutive elements sequence.
    - **Example**: Input: `nums = [100,4,200,1,3,2]` -> Output: `4` (Sequence `[1, 2, 3, 4]`)
    - **Approach (Union-Find variant)**: Can be solved by using Union-Find to group consecutive numbers. For each number `x` in the input, if `x+1` exists, union `x` and `x+1`. After processing all numbers, iterate through each `x` and find the size of its component to get the length of the consecutive sequence it belongs to. Max component size is the answer. (More commonly solved with Hashing, but this demonstrates Union-Find application).

Pattern 7: Bipartite Graph / Coloring

Problems that involve dividing vertices into two disjoint sets such that every edge connects a vertex in one set to one in the other. Often solved using graph coloring (usually 2-coloring).

- **Easy**: **Is Graph Bipartite?**
    - **Problem**: Given an undirected graph, return `true` if and only if it is bipartite. A graph is bipartite if we can split its set of nodes into two independent subsets `A` and `B` such that every edge in the graph connects a node in `A` to a node in `B`.
    - **Example**: Input: `graph = [[1,2,3],[0,2],[0,1,3],[0,2]]` -> Output: `false`
    - **Approach (BFS or DFS)**: Use BFS or DFS. Maintain a `color` array/map (e.g., 0 for uncolored, 1 for color A, 2 for color B). Iterate through all nodes. If a node is uncolored, start a traversal (BFS or DFS) from it. Assign it a color, then assign opposite colors to its neighbors. If at any point a neighbor has the same color as the current node, the graph is not bipartite.
- **Medium**: **Possible Bipartition**
    - **Problem**: We want to split a group of `n` people into two groups of `dislikes` (disjoint sets), such that no two people in the same group dislike each other. Given `n` and an array of `dislikes` where `dislikes[i] = [a_i, b_i]` means person `a_i` dislikes person `b_i`, return `true` if it is possible to split them, otherwise `false`.
    - **Example**: Input: `n = 4, dislikes = [[1,2],[1,3],[2,4]]` -> Output: `true`
    - **Approach**: This is equivalent to checking if the graph formed by people and their dislikes is bipartite.

Build an adjacency list from `dislikes`. Apply the same BFS/DFS 2-coloring approach as "Is Graph Bipartite?".

Pattern 8: Matrix / Grid-based Graph Problems

Many problems on 2D grids can be modeled as graph problems where grid cells are nodes and adjacent cells are connected by edges.

- **Easy**: **Flood Fill**
  - **Problem**: An image is represented by an `m x n` integer grid `image`. You are given coordinates `(sr, sc)` representing the starting pixel and a `color`. Perform a flood fill on the image.
  - **Example**: Input: `image = [[1,1,1],[1,1,0],[1,0,1]]`, `sr=1, sc=1, color=2` -> Output: `[[2,2,2],[2,2,0],[2,0,1]]`
  - **Approach (BFS or DFS)**: Start a traversal (BFS or DFS) from `(sr, sc)`. Recursively (DFS) or iteratively (BFS) visit all 4-directionally connected cells that have the original color of `(sr, sc)`, and change their color to `new_color`.
- **Medium**: **Pacific Atlantic Water Flow**
  - **Problem**: Given an `m x n` rectangular integer array `heights` representing the height of land at each cell, return a list of grid coordinates `(r, c)` where water can flow from that cell to both the Pacific and Atlantic oceans.
  - **Example**: Input: `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]]`
  - **Approach (Multi-source DFS/BFS)**: Instead of flowing from land to oceans, consider flowing from oceans to land. Start two independent DFS/BFS traversals: one from all cells adjacent to the Pacific Ocean (top row, left column) and another from all cells adjacent to the Atlantic Ocean (bottom row, right column). During traversal, water can flow from a cell `(r1, c1)` to `(r2, c2)` if `heights[r2][c2] >= heights[r1][c1]`. Mark cells reachable from Pacific and cells reachable from Atlantic. The intersection of these two sets of cells are the answer.
- **Hard**: **Shortest Path in Binary Matrix**
  - **Problem**: Given an `n x n` binary matrix `grid`, return the length of the shortest clear path in the matrix. A clear path is a path from `(0, 0)` to `(n-1, n-1)` such that all visited cells are `0` and all adjacent cells in the path

are 8-directionally connected.
- **Example**: Input: `grid = [[0,1],[1,0]]` -> Output: `2`
- **Approach (BFS)**: This is a classic BFS problem for shortest path on an unweighted grid with 8-directional movement. Use a queue to store `(r, c, distance)`. Mark visited cells to avoid cycles.

This enhanced guide provides an even more comprehensive overview of essential Graph topics and a wide array of interview patterns. Mastering graphs requires a strong understanding of their representations, core traversal algorithms (BFS/DFS), and then applying specialized algorithms (Dijkstra's, Prim's, Kruskal's, Topological Sort, Union-Find) based on the problem's constraints (weighted/unweighted, directed/undirected, cyclic/acyclic, connectivity requirements). Consistent and varied practice across these patterns will significantly improve your problem-solving skills for SDE interviews.

Hashing (Hash Tables): Comprehensive Guide for SDE Interviews

Hashing is a technique used to map data of arbitrary size (keys) to data of a fixed size (values) using a **hash function**. The data structure that implements this technique is called a **Hash Table** (or Hash Map). Hash tables provide highly efficient (average time complexity) operations for insertion, deletion, and lookup, making them one of the most frequently used data structures in modern software development and critical for SDE interviews.

1. Essential Topics to Know in Hashing

Before diving into problem patterns, ensure you have a solid grasp of these core hashing concepts:

- **Definition**: A hash table is a data structure that maps keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.
- **Hash Function**:
    - A function that converts a given key into an integer index (hash code or hash value).
    - Should be deterministic (same key always produces same hash code).
    - Should distribute keys uniformly across the hash table to minimize collisions.
    - Should be fast to compute.
- **Collision**: Occurs when the hash function generates the same index for two or more different keys. This is inevitable with a finite number of buckets and an infinite (or very large) number of possible keys (Pigeonhole Principle).
- **Collision Resolution Strategies**: How hash tables handle collisions to ensure all keys can be stored and retrieved.
    - **Separate Chaining**: Each bucket in the hash table array points to a linked list (or other data structure like a small array or BST) of key-value pairs that hash to the same index.
        - Pros: Simple, handles many collisions gracefully, table size can be flexible.
        - Cons: Requires extra memory for pointers/lists, can have cache performance issues if chains are long.
    - **Open Addressing**: All elements are stored directly within the hash table array itself. When a collision

occurs, the algorithm probes for an empty slot.
        - **Linear Probing**: Searches for the next available slot sequentially (e.g., `(hash(key) + i) % table_size`).
        - **Quadratic Probing**: Searches for slots using a quadratic increment (e.g., `(hash(key) + i^2) % table_size`).
        - **Double Hashing**: Uses a second hash function to determine the step size for probing.
        - Pros: Better cache performance (elements are contiguous), no overhead for pointers.
        - Cons: Sensitive to load factor, can suffer from clustering (linear probing), deletion can be complex.
- **Load Factor (α)**: The ratio of the number of items stored in the hash table to the total number of buckets. `n / M`, where `n` is number of items and `M` is number of buckets.
    - A high load factor increases the probability of collisions and degrades performance.
    - When load factor exceeds a threshold (e.g., 0.7 or 0.75), the hash table usually undergoes **resizing** (rehashing).
- **Resizing (Rehashing)**: When the load factor becomes too high, the hash table's underlying array is expanded (e.g., doubled in size), and all existing key-value pairs are re-inserted (re-hashed) into the new, larger table. This is an O(N) operation but amortized O(1) for operations over time.
- **Time Complexity**:
    - **Average Case**: O(1) for insertion, deletion, lookup. Assumes a good hash function and proper collision resolution.
    - **Worst Case**: O(N) for insertion, deletion, lookup. Occurs when many keys hash to the same bucket (poor hash function or malicious input), or if collisions are poorly resolved (e.g., linear probing with high load factor).
- **Space Complexity**: O(N) to store N elements.
- **Applications**:
    - Implementing dictionaries/maps (`HashMap`, `unordered_map`, `dict`).
    - Counting frequencies of elements.
    - Checking for duplicates or presence of elements.
    - Caching.
    - Symbol tables in compilers.

- Database indexing.
- Cryptographic hashing (for data integrity, not typically for data structures).

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in Hashing-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Frequency Counting & Duplicates

The most common application of hash tables is to count occurrences of elements or quickly check for duplicates.

- **Easy**: **Two Sum**
  - **Problem**: Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.
  - **Example**: Input: `nums = [2,7,11,15]`, `target = 9` -> Output: `[0,1]`
  - **Approach**: Use a hash map to store `(number, index)`. Iterate through `nums`. For each `num`, check if `target - num` exists in the hash map. If yes, return the current index and `hash_map[target - num]`. Otherwise, add `(num, index)` to the hash map. This converts O(N2) (brute force) or O(NlogN) (sorting + two pointers) to O(N) average time.
- **Easy**: **Contains Duplicate**
  - **Problem**: Given an integer array `nums`, return `true` if any value appears at least twice in the array, and return `false` if every element is distinct.
  - **Example**: Input: `nums = [1,2,3,1]` -> Output: `true`
  - **Approach**: Use a hash set. Iterate through `nums`. If the current element is already in the hash set, return `true`. Otherwise, add it to the hash set.
- **Medium**: **Group Anagrams**
  - **Problem**: Given an array of strings `strs`, group the anagrams together.
  - **Example**: Input: `strs = ["eat","tea","tan","ate","nat","bat"]` -> Output: `[["bat"],["nat","tan"],["ate","eat","tea"]]`

    - **Approach**: Use a hash map where the key is a canonical representation of an anagram, and the value is a list of strings that are anagrams of that key. A common canonical representation is the sorted version of the string (e.g., "eat" -> "aet") or a frequency count array converted to a string.
- **Medium**: **Longest Consecutive Sequence**
    - **Problem**: Given an unsorted array of integers `nums`, return the length of the longest consecutive elements sequence. The algorithm should run in O(N) time.
    - **Example**: Input: `nums = [100,4,200,1,3,2]` -> Output: `4` (for `[1,2,3,4]`)
    - **Approach**: Store all numbers in a hash set for O(1) average time lookups. Iterate through each `num` in the set. If `num - 1` is *not* in the set, it means `num` is the start of a new consecutive sequence. Then, count how long this sequence extends by checking for `num + 1`, `num + 2`, etc., in the hash set. Update the maximum length found.

Pattern 2: Hash Table as a Data Structure (Design & Customization)

Problems that require you to understand or implement aspects of a hash table's internal workings, or use it for specialized lookups.

- **Easy**: **Implement HashMap / HashSet from scratch (Conceptual)**
    - **Problem**: Design a `HashMap` or `HashSet` without using any built-in hash table libraries.
    - **Approach**: Use an array as the underlying storage. Implement a simple hash function (e.g., modulo operator). Choose a collision resolution strategy (e.g., separate chaining with linked lists or open addressing with linear probing). Implement `put`, `get`, `remove` for `HashMap` or `add`, `contains`, `remove` for `HashSet`. Consider resizing.
- **Medium**: **Design a Logger Rate Limiter**
    - **Problem**: Design a logger system that receives a stream of messages along with their timestamps. Each message should be printed only if it's not printed in the last 10 seconds.
    - **Example**: `logger.shouldPrintMessage(1, "foo"); // returns true; logger.shouldPrintMessage(2, "bar"); // returns true; logger.shouldPrintMessage(3, "foo"); // returns false (3 < 1 + 10);`
    - **Approach**: Use a hash map where keys are `message_string` and values are `last_printed_timestamp`. When a

`message, timestamp` comes, check `hash_map[message]`. If `timestamp >= hash_map[message] + 10`, print and update timestamp. Else, don't print.

- **Hard**: **LRU Cache**
  - **Problem**: Design a data structure that implements a Least Recently Used (LRU) cache. It should support `get(key)` (returns value if key exists, else -1, and moves key to front of cache) and `put(key, value)` (inserts/updates, if cache capacity exceeded, removes least recently used). Both operations should be O(1) average time.
  - **Example**: `LRUCache cache = new LRUCache(2); cache.put(1,1); cache.put(2,2); cache.get(1); // returns 1; cache.put(3,3); // LRU key 2 is evicted; cache.get(2); // returns -1;`
  - **Approach**: This requires a combination of a hash map and a doubly linked list. The hash map stores `(key, node_reference_in_linked_list)` for O(1) lookups. The doubly linked list maintains the order of usage (most recently used at one end, least recently used at the other) for O(1) updates to usage order and O(1) eviction.

Pattern 3: Prefix/Suffix Hashing (String-Specific)

For string problems, hashing can be used to quickly compare substrings or identify patterns, often in conjunction with rolling hashes to achieve O(1) average time for substring hashes.

- **Easy**: **Ransom Note**
  - **Problem**: Given two strings `ransomNote` and `magazine`, return `true` if `ransomNote` can be constructed by using the letters from `magazine` (each letter in `magazine` can only be used once).
  - **Example**: Input: `ransomNote = "aab"`, `magazine = "baa"` -> Output: `true`
  - **Approach**: Use a frequency map (hash map or array of size 26) for `magazine`. Then iterate through `ransomNote`, decrementing counts. If any count goes below zero, return `false`.
- **Medium**: **Longest Repeating Substring**
  - **Problem**: Given a string `s`, return the length of the longest repeating substring. If no repeating substring exists, return `0`.
  - **Example**: Input: `s = "abcd"` -> Output: `0`; Input: `s = "ababa"` -> Output: `2` (for "ab" or "ba")

    - **Approach**: Binary search on the length of the substring. For a given length `L`, use a rolling hash (polynomial hashing) to compute hashes of all substrings of length `L`. Store these hashes in a hash set. If any hash collides (and the actual strings match to avoid false positives), then a repeating substring of length `L` exists.
- **Hard**: **Find All Anagrams in a String**
    - **Problem**: Given two strings `s` and `p`, return an array of all the start indices of `p`'s anagrams in `s`.
    - **Example**: Input: `s = "cbaebabacd"`, `p = "abc"` -> Output: `[0,6]`
    - **Approach**: This is a sliding window problem combined with hashing. Maintain frequency maps for `p` and for the current sliding window in `s`. Compare the frequency maps. If they match, add the window's start index to the result. More optimally: instead of comparing full maps, track a `match_count` of characters.

Pattern 4: Hashing for Set Operations / Distinct Elements

Using hash sets (or hash maps where values are just placeholders) for efficient set operations like union, intersection, difference, or finding unique elements.

- **Easy**: **Intersection of Two Arrays**
    - **Problem**: Given two integer arrays `nums1` and `nums2`, return an array of their intersection. Each element in the result must be unique.
    - **Example**: Input: `nums1 = [1,2,2,1]`, `nums2 = [2,2]` -> Output: `[2]`
    - **Approach**: Put all elements of `nums1` into a hash set. Iterate through `nums2`. If an element from `nums2` is in the hash set, add it to a result set (to ensure uniqueness in result) and remove it from the first hash set (optional, but can optimize if `nums2` has many duplicates). Convert result set to array.
- **Medium**: **Longest Substring Without Repeating Characters**
    - **Problem**: Given a string `s`, find the length of the longest substring without repeating characters.
    - **Example**: Input: `s = "abcabcbb"` -> Output: `3` (for "abc")
    - **Approach**: Use a sliding window and a hash set. Expand the window (right pointer). If `s[right]` is already in the hash set, shrink the window (left pointer) by removing `s[left]` from the set until `s[right]` is no longer a duplicate. Update max length at each step.

- **Hard**: **Count of Unique Palindromic Subsequences**
    - **Problem**: Given a string `s`, return the number of unique palindromic subsequences of length 3.
    - **Example**: Input: `s = "aabca"` -> Output: `3` (for "aba", "aca", "aaa")
    - **Approach**: Iterate through all possible middle characters `c`. For each `c`, find its first and last occurrence in `s`. Then, count the unique characters *between* these first and last occurrences. Use a hash set to store unique characters between.

Pattern 5: Hashing with Graphs

Hash tables are frequently used in graph algorithms for storing adjacency lists, managing visited states, or optimizing neighbor lookups, especially when node IDs are not contiguous integers.

- **Easy**: **Clone Graph**
    - **Problem**: Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph.
    - **Example**: Input: A graph node structure.
    - **Approach**: Use BFS or DFS to traverse the original graph. Maintain a hash map `(original_node -> cloned_node)` to store mappings of already cloned nodes. When traversing, if a node hasn't been cloned, create its clone and add it to the map. Then iterate its neighbors and link them to their cloned counterparts, or add them to the queue for cloning.
- **Medium**: **Graph Valid Tree** (revisited - specifically using a Map for Adjacency List)
    - **Problem**: Given `n` nodes and `edges`, check if it forms a valid tree.
    - **Approach**: Instead of an array for adjacency list, use `HashMap<Integer, List<Integer>>` if `n` is very large or node IDs are sparse. This is common when nodes are arbitrary IDs (e.g., user IDs) rather than 0 to `n-1`. The BFS/DFS or Union-Find logic remains the same, but the representation adapts to hashing.
- **Hard**: **Snakes and Ladders (BFS with visited map)**
    - **Problem**: On an `n x n` board, squares are numbered from 1 to `n^2`. Some squares have snakes or ladders. Find the least number of moves to reach square `n^2`.
    - **Example**: Input: `board =`

`[[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]` -> Output: `4`

- **Approach**: This is a shortest path problem on an unweighted graph, solvable with BFS. The nodes are the squares. Build a mapping from `square_number` to `(row, col)`. Use a queue for BFS and a hash set (or 2D boolean array) for `visited` squares to track distance and avoid cycles.

This comprehensive guide covers the essential theoretical foundations and practical patterns for working with Hashing and Hash Tables. Mastering hashing involves understanding the core concepts of hash functions, collision resolution, and their impact on performance, as well as recognizing when and how to apply hash tables for efficient data management in various problem-solving scenarios. Consistent practice across these patterns will significantly improve your problem-solving skills for SDE interviews.

Segment Trees & Fenwick Trees: Comprehensive Guide for SDE Interviews

Segment Trees and Fenwick Trees (also known as Binary Indexed Trees or BITs) are powerful data structures primarily used for efficiently handling **range queries** and **point updates** on an array. They are crucial for competitive programming and can appear in advanced SDE interviews when problems require quick processing of data over ranges, especially when the underlying data changes frequently.

1. Essential Topics to Know in Segment Trees & Fenwick Trees

Before diving into problem patterns, ensure you have a solid grasp of these core concepts:

Segment Trees

- **Definition**: A binary tree used for storing information about intervals or segments. Each node in the segment tree represents an interval. The root represents the entire array, and its children represent the two halves of that interval. Leaf nodes represent individual array elements.
- **Structure**: A segment tree is typically a complete binary tree (or nearly complete).
    - If a node represents interval `[L, R]`, its left child represents `[L, (L+R)/2]` and its right child represents `[(L+R)/2 + 1, R]`.
- **Storage**: Usually implemented using an array, similar to a heap. If the original array has size `N`, the segment tree array will typically be `~2N` to `~4N` depending on implementation details (e.g., power of 2 size).
- **Core Operations ()**:
    - **Build**: Construct the segment tree from the original array.
        - Bottom-up: Build from leaves up to the root. Each internal node stores the result of combining its children's results (e.g., sum, min, max). Complexity: O(N).
    - **Update (Point Update)**: Change the value of a single element in the original array and propagate this change up to the relevant nodes in the segment tree. Complexity: O(logN).
    - **Query (Range Query)**: Retrieve information (e.g., sum, min, max) for a given range `[query_L, query_R]`. The query traverses the tree, combining results from relevant segments that fully or partially overlap with the query range. Complexity: O(logN).

- **Types of Queries**: Can support various associative operations: sum, min, max, XOR, GCD, etc.
- **Lazy Propagation (Advanced)**: Optimization for handling **range updates** (e.g., add a value to all elements in a range). Instead of updating all individual nodes, changes are "lazily" propagated down the tree only when necessary. This allows range updates in O(logN) time.
- **Time Complexity**:
    - Build: O(N)
    - Point Update: O(logN)
    - Range Query: O(logN)
    - Range Update (with lazy propagation): O(logN)
- **Space Complexity**: O(N) (typically 2N to 4N for array representation).
- **Applications**:
    - Range Sum/Min/Max Queries.
    - Counting inversions in an array.
    - Finding elements greater than `X` in a range.
    - Problems requiring efficient updates and queries on intervals.

Fenwick Trees (Binary Indexed Trees - BIT)

- **Definition**: A data structure that can efficiently update elements and calculate prefix sums in an array. It uses an array, but its indices are cleverly chosen to represent prefixes.
- **Structure**: Unlike segment trees, BITs are not explicit trees in memory but rather an array-based representation where each index stores a sum of a specific range of elements. The `i`-th element in the BIT array stores the sum of elements from `i - (i & -i) + 1` to `i`.
- **Core Operations (**O(logN)**)**:
    - **Update (Point Update)**: Add a value to an element at a given index. The update propagates up the BIT array by repeatedly adding `(i & -i)` to the index. Complexity: O(logN).
    - **Query (Prefix Sum)**: Calculate the sum of elements from index 1 up to a given index `i` (`query(i)`). This is done by repeatedly subtracting `(i & -i)` from the index and summing up the values at those indices.

Complexity: O(logN).
    - **Range Sum Query**: `sumRange(L, R) = query(R) - query(L-1)`. Complexity: O(logN).
- **Advantages**:
    - Simpler to implement than segment trees for prefix sum/point update operations.
    - Uses less space (O(N) for the BIT array, typically `N+1`).
- **Limitations**:
    - Primarily designed for prefix sums. Range updates are more complex (often requires two BITs).
    - Cannot easily support arbitrary associative operations like min/max without modification.
- **Time Complexity**:
    - Build (from scratch, by `update` operations): O(NlogN) (or O(N) if built optimally).
    - Point Update: O(logN)
    - Prefix Sum Query: O(logN)
    - Range Sum Query: O(logN)
- **Space Complexity**: O(N) (typically `N+1` for array representation).
- **Applications**:
    - Prefix Sum / Range Sum queries with point updates.
    - Counting inversions (alternative to Segment Tree).
    - Dynamic ranking and selection.

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in Segment Tree and Fenwick Tree-based SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Range Sum Queries with Point Updates

This is the most common use case for both Segment Trees and Fenwick Trees.

- **Easy**: **Range Sum Query - Mutable (Fenwick Tree / Segment Tree)**

    - **Problem**: Given an integer array `nums`, handle multiple queries of the form `sumRange(i, j)`, which returns the sum of elements between indices `i` and `j` (inclusive). Also supports `update(index, val)` which updates `nums[index]` to `val`.
    - **Example**: `NumArray numArray = new NumArray([1, 3, 5]); numArray.sumRange(0, 2); // return 9; numArray.update(1, 2); numArray.sumRange(0, 2); // return 8;`
    - **Approach (Fenwick Tree)**: Initialize BIT. `update(index, val)` computes `val - original_nums[index]` and updates this difference in BIT. `query(i)` computes prefix sum. `sumRange(i, j)` is `query(j) - query(i-1)`.
    - **Approach (Segment Tree)**: Build sum segment tree. `update` propagates change up. `query` traverses segments to sum up.
- **Medium**: **Count of Range Sum**
    - **Problem**: Given an integer array `nums`, return the number of range sums that lie in `[lower, upper]` inclusive. A range sum `S(i, j)` is the sum of `nums` in range `[i, j]`.
    - **Example**: Input: `nums = [-2,5,-1]`, `lower = -2`, `upper = 2` -> Output: `3` (Range sums are `[-2]`, `[5]`, `[-1]`, `[-2,5]=3`, `[5,-1]=4`, `[-2,5,-1]=2`. Those in `[-2,2]` are `[-2]`, `[-1]`, `[2]`).
    - **Approach**: This is typically solved using a modified merge sort or a **Fenwick Tree / Segment Tree** on prefix sums. Calculate prefix sums `P`. For each `P[i]`, we need to find `P[j]` such that `lower <= P[i] - P[j] <= upper` for `j < i`. This transforms to `P[i] - upper <= P[j] <= P[i] - lower`. Use a BIT/Segment Tree on a compressed version of `P` values to efficiently count elements within this range.
- **Hard**: **Reverse Pairs**
    - **Problem**: Given an integer array `nums`, return the number of reverse pairs. A reverse pair is a pair `(i, j)` where `0 <= i < j < nums.length` and `nums[i] > 2 * nums[j]`.
    - **Example**: Input: `nums = [1,3,2,3,1]` -> Output: `2` (`(3,1)` and `(3,1)`)
    - **Approach**: This can be efficiently solved using a **Fenwick Tree or Segment Tree** in conjunction with a custom merge sort (similar to counting inversions). As you merge, for each element `nums[i]` from the left half, count how many elements `nums[j]` in the right half satisfy `nums[i] > 2 * nums[j]` using the BIT/Segment Tree on the sorted values.

Pattern 2: Range Min/Max Queries with Point Updates

Segment Trees are particularly well-suited for these types of queries.

- **Easy**: **Range Minimum Query (Segment Tree)**
  - **Problem**: Given an array `nums`, handle `update(index, val)` and `queryMin(i, j)` operations.
  - **Approach**: Build a segment tree where each node stores the minimum value in its segment. `update` propagates the new minimum up. `queryMin` traverses the tree, returning the minimum from overlapping segments.
- **Medium**: **Range Max Query (Segment Tree)**
  - **Problem**: Given an array `nums`, handle `update(index, val)` and `queryMax(i, j)` operations.
  - **Approach**: Similar to Range Min Query, but each node stores the maximum.
- **Hard**: **The Skyline Problem**
  - **Problem**: A city's skyline is formed by a series of buildings. Given the locations and heights of all the buildings, return the skyline profile (key points of the skyline).
  - **Example**: Input: `buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]`
  - **Approach**: This is a classic advanced sweep-line algorithm that heavily relies on a **Max-Heap** or, more efficiently, a **Segment Tree (or custom data structure) to maintain current active heights**. Sort building critical points (start and end X-coordinates). Use a segment tree to efficiently query the maximum height in the currently active range of X-coordinates, updating when new buildings start or old ones end.

Pattern 3: Range Updates & Lazy Propagation (Segment Tree)

For problems where a range of elements needs to be updated with a single value (e.g., add 5 to all elements from index `i` to `j`).

- **Medium**: **Range Addition II (Simplified Range Update)**
  - **Problem**: You are given an `m x n` matrix `M` initialized with all `0`s and a list of `ops` where `ops[i] = [a_i, b_i]` means all elements `M[x][y]` with `0 <= x < a_i` and `0 <= y < b_i` are incremented by one. Return the number of maximum integers in the matrix after performing all operations.

    - **Approach**: This is a simplification. The max value will always be in the top-left corner `(0,0)`. The final value at `(0,0)` will be the minimum of all `a_i` and `b_i`. The number of elements with this max value will be `min_a * min_b`.
- **Hard**: **Range Sum Query - Mutable (with Range Updates)**
    - **Problem**: Extend "Range Sum Query - Mutable" to also support `rangeUpdate(i, j, val)`: add `val` to all elements `nums[x]` where `i <= x <= j`.
    - **Approach**: This requires a **Segment Tree with Lazy Propagation**.
        1. **Lazy Tag**: Each node in the segment tree has an additional `lazy_tag` field. This tag stores pending updates that haven't been pushed down to its children yet.
        2. `push_down` **function**: Before accessing a node's children (for query or update), if the node has a `lazy_tag`, push its value down to its children's `lazy_tag`s and update their stored sums. Clear the parent's `lazy_tag`.
        3. `rangeUpdate`: Similar to point update, but when a segment node completely covers the update range, just update its sum and `lazy_tag`, then return. If partially overlaps, `push_down`, then recurse on children.
        4. `rangeQuery`: Similar, but ensure `push_down` is called on nodes before their children are queried.

Pattern 4: Counting Inversions and Related Problems

BITs and Segment Trees can efficiently count pairs satisfying certain conditions.

- **Medium**: **Count of Smaller Numbers After Self** (using BIT / Segment Tree)
    - **Problem**: Given an integer array `nums`, return an array `answer` where `answer[i]` is the number of smaller elements to the right of `nums[i]`.
    - **Example**: Input: `nums = [5,2,6,1]` -> Output: `[2,1,1,0]`
    - **Approach**: Iterate through `nums` from right to left. Use a **Fenwick Tree (BIT)** or a **Segment Tree** on a compressed version of the values (or coordinate compression if values are large). For each `nums[i]`:
        1. Query the BIT/Segment Tree for the sum (count) of all elements seen so far that are smaller than

`nums[i]`. This sum is `answer[i]`.
    2. Update the BIT/Segment Tree by adding `1` at the index corresponding to `nums[i]`.
- **Hard**: **Longest Increasing Subsequence (N log N using BIT/Segment Tree)**
    - **Problem**: Given an integer array `nums`, return the length of the longest strictly increasing subsequence.
    - **Example**: Input: `nums = [10,9,2,5,3,7,101,18]` -> Output: `4` (LIS is `[2,3,7,101]`)
    - **Approach**: This can be solved in O(NlogN) using Binary Search (patience sorting) or more explicitly with a **Fenwick Tree / Segment Tree**. After coordinate compression of `nums`, `dp[val]` can represent the length of the LIS ending with `val`. To find `dp[nums[i]]`, query the BIT/Segment Tree for the maximum LIS length for values less than `nums[i]`. Then update the BIT/Segment Tree with `1 + max_length`.

This comprehensive guide covers the essential theoretical foundations and practical patterns for working with Segment Trees and Fenwick Trees. Mastering these advanced data structures involves understanding their array-based tree structure, how updates propagate, and how queries efficiently combine segment results. They are particularly valuable for problems requiring efficient range operations on mutable data. Consistent practice across these patterns will significantly improve your problem-solving skills for advanced SDE interviews and competitive programming.

Bit Manipulation: Comprehensive Guide for SDE Interviews

Bit Manipulation is the act of algorithmically manipulating bits or other smaller pieces of data. It's a powerful and often overlooked technique in software development and competitive programming, allowing for highly optimized and elegant solutions with constant extra space and sometimes faster execution times. Mastery of bitwise operators and their properties is crucial for certain interview problems.

1. Essential Topics to Know in Bit Manipulation

Before diving into problem patterns, ensure you have a solid grasp of these core concepts:

- **Binary Representation**: Understanding how integers are represented in binary (base-2).
  - Decimal to Binary conversion.
  - Binary to Decimal conversion.
- **Bitwise Operators**: Know the syntax, truth tables, and effects of each operator:
  - **AND (`&`)**: Sets a bit to 1 if both corresponding bits are 1. Useful for masking, checking if a bit is set.
  - **OR (`|`)**: Sets a bit to 1 if at least one corresponding bit is 1. Useful for setting a bit.
  - **XOR (`^`)**: Sets a bit to 1 if corresponding bits are different. Useful for swapping, finding unique elements, toggling bits.
  - **NOT (`~`)**: Flips all bits (one's complement).
  - **Left Shift (`<<`)**: Shifts bits to the left, filling with zeros on the right. Equivalent to multiplying by powers of 2.
  - **Right Shift (`>>`)**: Shifts bits to the right.
    - **Arithmetic Right Shift**: Fills with the sign bit on the left (preserves sign). Standard `>>` in Java/C++.
    - **Logical Right Shift (`>>>`)**: Fills with zeros on the left (only in Java, not standard in C++).
- **Common Bitwise Identities and Techniques**:
  - `1 << i`: Creates a number with only the `i`-th bit set.
  - `x & (1 << i)`: Checks if the `i`-th bit of `x` is set.
  - `x | (1 << i)`: Sets the `i`-th bit of `x`.
  - `x & ~(1 << i)`: Clears (resets to 0) the `i`-th bit of `x`.

    - `x ^ (1 << i)`: Toggles the `i`-th bit of `x`.
    - `x & (x - 1)`: Clears the least significant bit (LSB) that is set. Useful for counting set bits.
    - `x & (-x)`: Isolates the least significant bit that is set.
    - `x ^ y ^ y = x`: XOR property for canceling out.
- **Signed vs. Unsigned Integers**: How negative numbers are represented (typically Two's Complement) and its implications for bitwise operations, especially right shifts.
- **Bitmasks**: Using an integer where each bit represents a boolean state or a flag. Useful for representing subsets, permutations, or configurations.
- **Count Set Bits (Hamming Weight)**: Efficiently counting the number of 1s in a binary representation.

2. Questions Categorized by Patterns

Here are the most common and important patterns encountered in Bit Manipulation SDE interview questions, along with example problems for varying difficulty levels.

Pattern 1: Basic Bitwise Operations & Properties

Problems that directly test your understanding of fundamental bitwise operators and their simple applications.

- **Easy**: **Counting Bits (Hamming Weight)**
    - **Problem**: Given an integer `n`, return an array `ans` of length `n + 1` such that for each `i` (`0 <= i <= n`), `ans[i]` is the number of `1`'s in the binary representation of `i`.
    - **Example**: Input: `n = 2` -> Output: `[0,1,1]`
    - **Approach**: Can be solved with a loop and `x & (x-1)` for each number (Brian Kernighan's algorithm) or using Dynamic Programming: `ans[i] = ans[i & (i-1)] + 1` or `ans[i] = ans[i >> 1] + (i & 1)`.
- **Easy**: **Convert a Number to Hexadecimal**
    - **Problem**: Given an integer `num`, return a hexadecimal string representation of it. For negative integers, use two's complement.
    - **Example**: Input: `num = 26` -> Output: `"1a"`
    - **Approach**: Use bitwise AND with `0xF` (1111 binary) to extract the last 4 bits, then right shift by 4. Repeat

until number is 0. Handle negative numbers by starting with a 32-bit (or 64-bit) unsigned representation using `>>>` (if available) or `& 0xFFFFFFFF`.

- **Medium**: **Single Number**
  - **Problem**: Given a non-empty array of integers `nums`, every element appears twice except for one. Find that single one.
  - **Example**: Input: `nums = [2,2,1]` -> Output: `1`
  - **Approach**: Use the XOR property: `a ^ a = 0` and `a ^ 0 = a`. XORing all numbers in the array will cancel out all pairs, leaving only the single unique number.
- **Medium**: **Single Number II (Unique appears once, others thrice)**
  - **Problem**: Given an integer array `nums` where every element appears three times except for one, which appears exactly once. Find the single element.
  - **Example**: Input: `nums = [0,1,0,1,0,1,99]` -> Output: `99`
  - **Approach**: Count the number of set bits at each position (0 to 31/63). If the count of a bit position is not a multiple of 3, then the unique number must have that bit set. Construct the unique number from these bits.
- **Hard**: **Sum of Two Integers (Without +/-)**
  - **Problem**: Given two integers `a` and `b`, return the sum of the two integers without using the `+` or `-` operators.
  - **Example**: Input: `a = 1, b = 2` -> Output: `3`
  - **Approach**: Use bitwise XOR for the sum without carries (`a ^ b`) and bitwise AND with left shift for carries (`(a & b) << 1`). Recursively (or iteratively) sum the `sum_without_carries` and the `carries` until `carries` become 0.

Pattern 2: Checking/Setting/Clearing Bits & Powers of 2

Problems focused on manipulating specific bits or identifying numbers with specific bit patterns (like powers of 2).

- **Easy**: **Power of Two**
  - **Problem**: Given an integer `n`, return `true` if it is a power of two. Otherwise, return `false`. An integer `n` is a power of two if there exists an integer `x` such that `n == 2^x`.
  - **Example**: Input: `n = 16` -> Output: `true` (16 is 2^4)

    - **Approach**: A positive integer is a power of two if and only if it has exactly one bit set in its binary representation. `n > 0 && (n & (n - 1)) == 0`.
- **Medium**: **Missing Number**
    - **Problem**: Given an array `nums` containing `n` distinct numbers in the range `[0, n]`, return the only number in the range that is missing from the array.
    - **Example**: Input: `nums = [3,0,1]` -> Output: `2`
    - **Approach**: Use XOR. XOR all numbers from `0` to `n` with all numbers in `nums`. The result will be the missing number, as all other numbers (present in both sets) will cancel out.
- **Medium**: **Find the Difference**
    - **Problem**: You are given two strings `s` and `t`. String `t` is generated by random shuffling string `s` and then add one more letter at a random position. Return the letter that was added to `t`.
    - **Example**: Input: `s = "abcd"`, `t = "abcde"` -> Output: `'e'`
    - **Approach**: Sum (or XOR) all character ASCII values in `s` and then in `t`. The difference (or remaining XOR) will be the added character.
- **Hard**: **Maximum XOR of Two Numbers in an Array**
    - **Problem**: Given an integer array `nums`, find the maximum result of `nums[i] XOR nums[j]`.
    - **Example**: Input: `nums = [3,10,5,25,2,8]` -> Output: `28` (5 XOR 25=(001012 XOR 110012)=111002=28)
    - **Approach**: Use a **Binary Trie (Prefix Tree)**. Insert all numbers into the Trie bit by bit from MSB to LSB. For each number `num` in `nums`, traverse the Trie. For each bit of `num`, try to take the opposite path in the Trie (e.g., if `num` has a 0 bit, try to go to the 1-bit child in the Trie). If the opposite path exists, take it (as it maximizes XOR for that bit position). If not, take the same path. This finds the `other_num` that maximizes XOR.

Pattern 3: Bitmasks for Subsets & State Representation

Using bits in an integer to represent a set of items or a state in a combinatorial problem.

- **Easy**: **Subsets (using bitmasks)**
    - **Problem**: Given an integer array `nums` of unique elements, return all possible subsets (the power set).

    - **Example**: Input: `nums = [1,2,3]` -> Output: `[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`
    - **Approach**: Iterate from `0` to `2^n - 1` (where `n` is `nums.length`). Each number `i` in this range can be interpreted as a bitmask. For each `i`, iterate from `j = 0` to `n-1`. If the `j`-th bit of `i` is set (`(i >> j) & 1`), include `nums[j]` in the current subset.
- **Medium**: **Can I Win**
    - **Problem**: In the "100 game", two players take turns choosing numbers from `1` to `maxChoosableInteger` (inclusive). The first player to reach or exceed `desiredTotal` wins. Assume both players play optimally. Can the first player win?
    - **Example**: Input: `maxChoosableInteger = 10, desiredTotal = 11` -> Output: `false`
    - **Approach**: This is a minimax game with memoization. The `state` of the game (which numbers have been chosen) can be represented by a **bitmask**. A bit `i` is set if `i+1` has been chosen. Use a hash map `(bitmask, boolean)` to store memoized results for `canWin(current_mask, current_total)`.
- **Hard**: **Shortest Path Visiting All Nodes**
    - **Problem**: You have an undirected graph with `n` nodes labeled from `0` to `n-1`. You are given `graph`, where `graph[i]` is a list of all neighbors of node `i`. Return the length of the shortest path that visits every node. You may start and stop at any node, and you may revisit nodes and reuse edges.
    - **Example**: Input: `graph = [[1,2,3],[0],[0],[0]]` -> Output: `4`
    - **Approach**: This is a BFS problem where the state is `(current_node, visited_mask)`. The `visited_mask` is a bitmask representing which nodes have been visited so far. Use a queue for BFS and a `visited_state[node][mask]` 2D array (or hash set) to avoid re-exploring states. The goal is to reach a state where `visited_mask` is `(1 << n) - 1` (all bits set).

Pattern 4: Bitwise Operations in Arrays/Matrices

Problems that apply bitwise logic to arrays or grid structures to find properties or achieve specific transformations.

- **Easy**: **Maximum Odd Binary Number**
    - **Problem**: You are given a binary string `s` that contains at least one `'1'`. Return the largest odd binary number that can be formed by rearranging the bits of `s`. The returned string should not have leading

zeros.
    - **Example**: Input: `s = "010"` -> Output: `"001"`
    - **Approach**: An odd binary number must end with a `1`. To maximize the number, put all other `1`s at the most significant positions (leftmost). Count the number of `1`s and `0`s. Form the string: `(num_ones - 1)` ones, followed by `num_zeros` zeros, followed by a single `1`.
- **Medium**: **Flipping an Image**
    - **Problem**: Given an `n x n` binary matrix `image`, flip the image horizontally, then invert it.
    - **Example**: Input: `image = [[1,1,0],[1,0,1],[0,0,0]]` -> Output: `[[1,0,0],[0,1,0],[1,1,1]]`
    - **Approach**: Iterate through each row. For horizontal flip: use two pointers `left` and `right` to swap elements. For invert: after swapping, if `image[r][c]` was 0, it becomes 1, and vice-versa. This can be done efficiently with `1 - pixel_value` or `pixel_value ^ 1`.
- **Hard**: **Bitwise AND of Numbers Range**
    - **Problem**: Given two integers `left` and `right` that represent the range `[left, right]`, return the bitwise AND of all numbers in this range, inclusive.
    - **Example**: Input: `left = 5, right = 7` -> Output: `4` (5 (0101_2) \text{ & } 6 (0110_2) \text{ & } 7 (0111_2) = 0100_2 = 4)
    - **Approach**: The result of ANDing a range of numbers will have common prefix bits. The differing bits from the right will become 0. Find the common most significant prefix. This can be done by repeatedly right-shifting both `left` and `right` until they become equal, while counting the shifts. Then left-shift `left` back by the count. Alternatively, clear the LSB of `right` until `right` becomes `left` or smaller.

This comprehensive guide covers the essential theoretical foundations and practical patterns for working with Bit Manipulation. Mastering bitwise operations involves understanding binary representations, operator precedence, and how to apply these techniques creatively for efficient solutions in terms of both time and space. Consistent practice across these patterns will significantly improve your problem-solving skills for SDE interviews.