**DSA JAVA Notes and Solutions   
Blind 75**

1. Contains Duplicate (Arrays/Hashing/Easy)

**Problem Description**

* Given an integer array nums, return true if any value appears at least twice in the array, and false if every element is distinct.

**Code with Comments**

class Solution {

public boolean hasDuplicate(int[] nums) {

// Initialize a HashSet to store unique numbers

HashSet<Integer> seen = new HashSet<>();

// Iterate through each number in the input array

for (int n : nums) {

// If the number is already in the HashSet, a duplicate is found

if (seen.contains(n)) {

return true;

}

// Add the number to the HashSet if it's not already present

seen.add(n);

}

// No duplicates found after iterating through the array

return false;

}

}

**Approach**

* **Use a HashSet**: Initialize a HashSet (seen) to track unique numbers.
* **Iterate through the array**: Loop through each number n in nums using a for-each loop.
* **Check for duplicates**: For each number, use contains to check if it exists in the HashSet. If it does, return true as a duplicate is found.
* **Add number to HashSet**: If the number is not in the HashSet, add it using add to track it for future iterations.
* **Return false if no duplicates**: If the loop completes without finding a duplicate, return false.

**Time Complexity**

* **O(n)**: The algorithm performs a single pass through the array of length n. Both contains and add operations on the HashSet have an average time complexity of O(1), resulting in O(n) overall.

**Space Complexity**

* **O(n)**: The HashSet stores up to n unique elements in the worst case (when no duplicates exist), leading to O(n) space complexity.

**Key Takeaways**

* **HashSet efficiency**: The HashSet enables O(1) lookups and insertions, making it ideal for detecting duplicates in O(n) time, compared to O(n²) for a brute-force pairwise comparison.
* **Edge cases**: The code correctly handles edge cases like empty arrays (returns false) and single-element arrays (returns false). The problem assumes valid integer inputs.
* **Alternative solutions**: Sorting the array (O(n log n) time, O(1) space) and checking adjacent elements is viable when space is constrained, but the HashSet approach prioritizes time efficiency.
* **Related problems**: This problem relates to "Two Sum" (hash map for lookups) and "Valid Anagram" (hash-based counting) in the Blind 75 list.
* **Interview tip**: Be prepared to discuss trade-offs between time and space complexity (e.g., HashSet vs. sorting) and explain why you chose the HashSet approach.

1. Valid Anagram (Arrays/Hashing/Easy)

**Problem Description**

* Given two strings s and t, return true if t is an anagram of s, and false otherwise. An anagram is a word formed by rearranging the letters of another, using all the original letters exactly once.

**Code with Comments**

class Solution {

public boolean isAnagram(String s, String t) {

// Check if lengths differ; if so, they cannot be anagrams

if (s.length() != t.length()) return false;

// Initialize array to count occurrences of each character (a-z)

int[] charCount = new int[26];

// Iterate through both strings simultaneously

for (int i = 0; i < s.length(); i++) {

// Increment count for character in s

charCount[s.charAt(i) - 'a']++;

// Decrement count for character in t

charCount[t.charAt(i) - 'a']--;

}

// Check if all character counts are zero

for (int i = 0; i < 26; i++) {

if (charCount[i] != 0) {

// Non-zero count means mismatch in character frequencies

return false;

}

}

// All counts are zero, so strings are anagrams

return true;

}

}

**Approach**

* **Length check**: Compare lengths of s and t; if they differ, return false as anagrams must have the same number of characters.
* **Use a frequency array**: Create an array of size 26 to track the frequency of each lowercase letter (assuming input contains only lowercase English letters).
* **Single pass through strings**: Iterate through both strings simultaneously, incrementing the count for each character in s and decrementing for each character in t using the index char - 'a'.
* **Verify frequencies**: After the loop, check if all counts in the array are zero. A non-zero count indicates a mismatch in character frequencies, so return false.
* **Return result**: If all counts are zero, the strings are anagrams, so return true.

**Time Complexity**

* **O(n)**: The algorithm performs a single pass through the strings of length n to update the frequency array, and a constant-time loop over 26 elements to check counts. The total time complexity is O(n), as the 26-element loop is O(1).
* **Note**: If the character set were larger (e.g., Unicode), the second loop’s complexity would depend on the character set size, but for lowercase English letters, it’s constant.

**Space Complexity**

* **O(1)**: The charCount array has a fixed size of 26 (for lowercase English letters), regardless of input size, resulting in constant space complexity.

**Key Takeaways**

* **Frequency counting efficiency**: Using a single array to track character frequencies for both strings (by incrementing for s and decrementing for t) optimizes both time and space compared to using two separate arrays or hash maps.
* **Edge cases**: The length check handles cases where strings have different lengths. The code assumes lowercase English letters, as per the problem’s constraints. If Unicode characters were allowed, a hash map or larger array would be needed.
* **Alternative approaches**: Sorting both strings and comparing them gives O(n log n) time and O(1) or O(n) space (depending on the sorting implementation). A hash map approach uses O(n) time and O(k) space, where k is the character set size, but is less space-efficient for small character sets.
* **Related problems**: This technique relates to "Group Anagrams" (hash map for character counts) and "Contains Duplicate" (hash-based uniqueness check) in the Blind 75 list.
* **Interview tip**: Highlight the space efficiency of the array-based approach for fixed character sets and be ready to adapt the solution for larger character sets (e.g., using a HashMap).

1. Two Sum (Arrays/Hashing/Easy)

**Problem Description**

* Given an array of integers nums and an integer target, return the indices of two numbers such that they add up to target. Each input has exactly one solution, and you may not use the same element twice.

**Code with Comments**

class Solution {

public int[] twoSum(int[] nums, int target) {

// Initialize a HashMap to store number-to-index mappings

HashMap<Integer, Integer> map = new HashMap<>();

// Iterate through the array

for (int i = 0; i < nums.length; i++) {

// Calculate the complement needed to reach the target

int complement = target - nums[i];

// Check if complement exists in the HashMap

if (map.containsKey(complement)) {

// Return the indices of the complement and current number

return new int[] {map.get(complement), i};

}

// Add the current number and its index to the HashMap

map.put(nums[i], i);

}

// Return empty array if no solution is found (though problem guarantees a solution)

return new int[]{};

}

}

**Approach**

* **Use a HashMap**: Initialize a HashMap to store numbers as keys and their indices as values.
* **Iterate through the array**: For each number nums[i] at index i, compute the complement (target - nums[i]).
* **Check for complement**: If the complement exists in the HashMap, return an array containing the complement’s index (from the HashMap) and the current index i.
* **Store current number**: If the complement is not found, add the current number and its index to the HashMap.
* **Single pass**: Process each element once, checking for the complement before adding the current number to avoid using the same element twice.
* **Default return**: Return an empty array if no solution is found (though the problem guarantees exactly one solution).

**Time Complexity**

* **O(n)**: The algorithm iterates through the array once, where n is the length of nums. Each HashMap operation (containsKey, get, and put) has an average time complexity of O(1), resulting in O(n) overall.

**Space Complexity**

* **O(n)**: The HashMap stores at most n key-value pairs (numbers and their indices) in the worst case, leading to O(n) space complexity.

**Key Takeaways**

* **HashMap efficiency**: The HashMap reduces the time complexity from O(n²) (brute-force nested loops) to O(n) by enabling O(1) lookups, trading space for time.
* **Edge cases**: The problem guarantees exactly one solution, so no need to handle cases with no or multiple solutions. The code correctly handles duplicates (e.g., [3, 3] with target=6) by checking the complement before adding the current number.
* **Alternative approaches**: A brute-force approach (checking all pairs) takes O(n²) time and O(1) space but is inefficient. Sorting and using two pointers is possible but complicates returning indices and requires O(n log n) time.
* **Related problems**: This technique is foundational for problems like "3Sum," "4Sum," and "Contains Duplicate" in the Blind 75 list, which also leverage hash-based or two-pointer approaches.
* **Interview tip**: Emphasize the importance of checking the complement before adding to the HashMap to avoid incorrect self-pairing. Be ready to discuss the empty return case, even though it’s not triggered due to the problem’s constraints.

1. Group Anagram (Arrays/Hashing/Medium)

**Problem Description**

* Given an array of strings strs, group the anagrams together and return a list of lists where each inner list contains strings that are anagrams of each other. An anagram is a word formed by rearranging the letters of another, using all the original letters exactly once.

**Code with Comments**

class Solution {

public List<List<String>> groupAnagrams(String[] strs) {

// Handle empty input case

if (strs.length == 0) {

return new ArrayList<>();

}

// Initialize HashMap to store key (encoded character count) to list of anagrams

Map<String, List<String>> map = new HashMap<>();

// Array to count frequency of each character (a-z)

int[] count = new int[26];

// Iterate through each string in the input array

for (String s : strs) {

// Reset count array for each string

Arrays.fill(count, 0);

// Count frequency of each character in the string

for (char c : s.toCharArray()) {

count[c - 'a']++;

}

// Create a unique key by concatenating character counts with '#' separator

StringBuilder sb = new StringBuilder("");

for (int i = 0; i < 26; i++) {

sb.append("#");

sb.append(count[i]);

}

String key = sb.toString();

// If key doesn't exist in map, create a new list for it

if (!map.containsKey(key)) {

map.put(key, new ArrayList<>());

}

// Add the current string to the list corresponding to its key

map.get(key).add(s);

}

// Return all lists of anagrams from the map

return new ArrayList<>(map.values());

}

}

**Approach**

* **Handle empty input**: If the input array is empty, return an empty list.
* **Use a HashMap**: Create a HashMap where the key is a string encoding the character frequency of each string, and the value is a list of strings that are anagrams (share the same key).
* **Count character frequencies**: For each string, use a 26-element array to count the frequency of each lowercase letter (a to z).
* **Create unique key**: Convert the frequency array into a string key by concatenating each count with a # separator (e.g., [1,2,0,...,0] becomes #1#2#0#...#0).
* **Group anagrams**: Add the string to the list in the HashMap corresponding to its key. If the key doesn’t exist, initialize a new list.
* **Return result**: Convert the HashMap’s values (lists of anagrams) into a list and return it.

**Time Complexity**

* **O(n \* k)**:
  + n is the length of the input array strs, and k is the maximum length of a string in strs.
  + For each string, counting characters takes O(k), building the key takes O(1) (since the 26-element loop is constant), and HashMap operations (containsKey, put, get) are O(1) on average.
  + Total time is O(n \* k) for iterating through all strings and processing their characters.
* **Note**: If the character set were larger (e.g., Unicode), key creation could scale with the character set size, but for lowercase English letters, the 26-element loop is O(1).

**Space Complexity**

* **O(n \* k)**:
  + The HashMap stores at most n strings across all lists, with each key (a string of length ~O(1) for 26 letters) and associated lists of strings.
  + The count array is O(1) (fixed size of 26).
  + The StringBuilder for key creation is O(1) since it’s reused and has a fixed length for 26 letters.
  + In the worst case, if all strings are unique, the space for storing strings in the HashMap is O(n \* k).
* **Output space**: The output list contains all input strings grouped, which is O(n \* k) but is typically not counted in space complexity analysis unless specified.

**Key Takeaways**

* **Character frequency as key**: Encoding character frequencies as a string key is an efficient way to identify anagrams, avoiding the O(n log n) cost of sorting each string.
* **Edge cases**: The code handles empty arrays and single-character strings correctly. It assumes lowercase English letters, as per the problem’s constraints. For Unicode, a HashMap for character counts would be needed.
* **Alternative approaches**: Sorting each string (O(k log k)) and using the sorted string as the key achieves the same result but increases time complexity to O(n \* k log k). The frequency-based approach is more efficient for small character sets.
* **Related problems**: This problem builds on "Valid Anagram" (frequency counting) and relates to "Two Sum" (hash-based lookups) and "Contains Duplicate" (hash-based uniqueness) in the Blind 75 list.
* **Interview tip**: Discuss the trade-offs between the frequency-based key (O(k) per string) and sorting-based key (O(k log k)). Highlight the use of # to avoid ambiguity in keys (e.g., distinguishing 1, 23 from 12, 3). Be prepared to adapt for larger character sets.

1. Top k Frequent Elements (Arrays/Hashing/Queue/Medium)

**Problem Description**

* Given an integer array nums and an integer k, return the k most frequent elements in the array. You may return the answer in any order. It is guaranteed that the answer is unique.

**Code with Comments**

class Solution {

public int[] topKFrequent(int[] nums, int k) {

// If k equals the length of nums, return the entire array

if (k == nums.length) {

return nums;

}

// Initialize HashMap to store number-to-frequency mappings

Map<Integer, Integer> count = new HashMap<>();

// Count frequency of each number in nums

for (int i = 0; i < nums.length; i++) {

count.put(nums[i], count.getOrDefault(nums[i], 0) + 1);

}

// Initialize min-heap to store numbers, ordered by their frequency

Queue<Integer> heap = new PriorityQueue<>((a, b) -> count.get(a) - count.get(b));

// Add numbers to heap, keeping only the top k frequent

for (int n : count.keySet()) {

heap.add(n);

// If heap size exceeds k, remove the number with the smallest frequency

if (heap.size() > k) {

heap.poll();

}

}

// Build result array by polling k elements from the heap

int[] ans = new int[k];

for (int i = 0; i < k; i++) {

ans[i] = heap.poll();

}

return ans;

}

}

**Approach**

* **Handle edge case**: If k equals the length of nums, return the entire array as all elements are included.
* **Count frequencies**: Use a HashMap to store the frequency of each number in nums, iterating through the array and updating counts with getOrDefault.
* **Use a min-heap**: Create a PriorityQueue (min-heap) to store numbers, ordered by their frequencies (lowest frequency at the top).
* **Maintain top k elements**: Iterate through the HashMap keys, adding each number to the heap. If the heap size exceeds k, remove the number with the smallest frequency using poll.
* **Build result**: Extract k elements from the heap into an array. Note that the result is in ascending order of frequency due to the min-heap; the problem allows any order.
* **Optimization note**: The heap size is kept at most k, ensuring efficient memory usage compared to storing all elements.

**Time Complexity**

* **O(n log k)**:
  + Building the frequency HashMap takes O(n), where n is the length of nums.
  + Iterating through the HashMap keys (at most n unique elements) and performing heap operations (add and poll) takes O(log k) each, since the heap size is at most k + 1. This step is O(n log k) in the worst case.
  + Extracting k elements from the heap takes O(k log k).
  + Overall, the dominant term is O(n log k).

**Space Complexity**

* **O(n + k)**:
  + The HashMap stores up to n key-value pairs (for unique elements), contributing O(n) space.
  + The min-heap stores at most k + 1 elements, contributing O(k) space.
  + The output array is O(k), but this is typically not counted unless specified.
  + Total space complexity is O(n + k), often simplified to O(n) since k ≤ n.

**Key Takeaways**

* **Heap for top k**: A min-heap is efficient for maintaining the top k elements, as it ensures the least frequent element is removed when the size exceeds k.
* **Edge cases**: The code handles k == nums.length correctly by returning the input array. The problem guarantees a unique answer, so no need to handle cases with insufficient elements.
* **Alternative approaches**:
  + **Bucket sort**: Use an array of lists (buckets) where the index represents frequency (O(n) time, O(n) space), which can be faster for large n and small k.
  + **Sorting the frequency map**: Sorting the HashMap entries by frequency takes O(n log n) time, which is less efficient for large n when k is small.
* **Related problems**: This problem relates to "Kth Largest Element in an Array" (heap-based) and "Group Anagrams" (hash-based grouping) in the Blind 75 list.
* **Interview tip**: Explain the choice of a min-heap over a max-heap (to keep size at k) and discuss the bucket sort alternative for better time complexity in certain cases. Be ready to handle variations, like returning elements in descending frequency order (requires reversing the output or using a max-heap).

1. Longest Consecutive Sequence (Arrays/Hashing/Medium)

**Problem Description**

* Given an unsorted array of integers nums, return the length of the longest consecutive sequence (i.e., numbers that form a sequence like [x, x+1, x+2, ...]). The sequence must be consecutive integers without gaps.

**Code with Comments**

class Solution {

public int longestConsecutive(int[] nums) {

// Handle empty array case

if (nums.length == 0) {

return 0;

}

// Initialize HashSet to store all numbers for O(1) lookups

HashSet<Integer> numSet = new HashSet<>();

// Add all numbers to the HashSet

for (int num : nums) {

numSet.add(num);

}

// Track the longest consecutive sequence found

int longest = 0;

// Iterate through each number in the HashSet

for (int num : numSet) {

// Only start a sequence if num-1 is not present (start of sequence)

if (!numSet.contains(num - 1)) {

int length = 1; // Initialize length of current sequence

// Check for consecutive numbers (num+1, num+2, ...)

while (numSet.contains(num + length)) {

length++;

}

// Update longest sequence if current sequence is longer

longest = Math.max(longest, length);

}

}

return longest;

}

}

**Approach**

* **Handle empty input**: If the input array is empty, return 0 as there is no sequence.
* **Use a HashSet**: Store all numbers in a HashSet for O(1) average-time lookups.
* **Identify sequence starts**: Iterate through each number in the HashSet. A number num is the start of a sequence if num-1 is not in the HashSet.
* **Count sequence length**: For each sequence start, check for consecutive numbers (num+1, num+2, ...) using the HashSet and count the length of the sequence.
* **Track maximum length**: Update the longest sequence length found using Math.max.
* **Optimization**: By only processing numbers that are the start of a sequence (i.e., num-1 is absent), the algorithm avoids redundant checks for numbers within a sequence.

**Time Complexity**

* **O(n)**:
  + Building the HashSet takes O(n), where n is the length of nums.
  + Iterating through the HashSet visits each number at most once, and for each sequence start, the while loop checks consecutive numbers. Each number is checked at most once across all sequences (since a number in the middle of a sequence skips the if condition).
  + All HashSet operations (add, contains) are O(1) on average.
  + Total time is O(n) for the initial loop and O(n) for the sequence-checking loop, resulting in O(n) overall.

**Space Complexity**

* **O(n)**: The HashSet stores at most n unique numbers, leading to O(n) space complexity.
* **Note**: The output (a single integer) uses O(1) space, which is negligible.

**Key Takeaways**

* **HashSet for efficiency**: The HashSet enables O(1) lookups, making it ideal for checking consecutive numbers efficiently, reducing the time complexity from O(n log n) (e.g., sorting-based approach) to O(n).
* **Edge cases**: The code handles empty arrays (returns 0) and arrays with duplicates (handled by the HashSet, which stores unique values). Single-element arrays return 1, which is correct.
* **Alternative approaches**:
  + Sorting the array and checking for consecutive sequences takes O(n log n) time and O(1) space (excluding output), but is less efficient.
  + Union-Find could be used but is overkill and typically slower for this problem.
* **Related problems**: This problem relates to "Contains Duplicate" (hash-based uniqueness) and "Group Anagrams" (hash-based grouping) in the Blind 75 list, as it leverages a HashSet for efficient lookups.
* **Interview tip**: Emphasize the optimization of only processing sequence starts (num-1 not in HashSet) to avoid redundant checks. Be ready to discuss why sorting is less efficient and how duplicates are handled implicitly by the HashSet.

1. Encode and Decode Strings

**Problem Description**

## Design an algorithm to encode a list of strings into a single string and decode it back to the original list of strings. The encoded string should be reversible to reconstruct the ex**Problem Description**

* Design an algorithm to encode a list of strings into a single string and decode it back to the original list of strings. The encoded string should be reversible to reconstruct the exact list.

**Code with Comments**

class Solution {

// Encode: Convert list of strings into a single string

public String encode(List<String> strs) {

// Initialize StringBuilder for efficient string concatenation

StringBuilder sb = new StringBuilder();

// Iterate through each string in the input list

for (String s : strs) {

// Append string length, a delimiter '#', and the string itself

sb.append(s.length()).append('#').append(s);

}

// Return the encoded string

return sb.toString();

}

// Decode: Convert encoded string back to list of strings

public List<String> decode(String s) {

// Initialize result list to store decoded strings

List<String> result = new ArrayList<>();

// Start index for parsing the encoded string

int i = 0;

// Continue until the entire string is processed

while (i < s.length()) {

// Find the position of the next '#' delimiter

int j = s.indexOf('#', i);

// Extract the length of the string (between i and j)

int length = Integer.parseInt(s.substring(i, j));

// Extract the string of given length after the delimiter

result.add(s.substring(j + 1, j + 1 + length));

// Move index to the start of the next string (after current string)

i = j + 1 + length;

}

return result;

}

}

**Approach**

* **Encode**:
  + Use a StringBuilder for efficient string concatenation.
  + For each string in the input list, append its length, a # delimiter, and the string itself to create the encoded string.
  + The format ensures that the length prefix allows decoding to know exactly how many characters to read for each string.
* **Decode**:
  + Initialize an empty list to store the decoded strings.
  + Iterate through the encoded string using an index i.
  + For each string:
    - Find the next # delimiter to identify the length prefix.
    - Parse the substring before # as an integer to get the string length.
    - Extract the substring of that length after the # and add it to the result.
    - Update the index i to the start of the next string (after the current string).
* **Delimiter choice**: The # delimiter separates the length from the string content, ensuring unambiguous parsing.

**Time Complexity**

* **Encode: O(n)**:
  + Iterating through the list of strings takes O(n), where n is the total number of characters across all strings (sum of their lengths).
  + Appending each string and its length to the StringBuilder is O(1) per character, totaling O(n).
* **Decode: O(n)**:
  + Iterating through the encoded string takes O(n), where n is the length of the encoded string.
  + indexOf for finding # is O(n) in the worst case, but since each character is processed once (index i advances past each string), the amortized time is O(n).
  + Parsing and substring operations are O(1) per string, contributing to O(n) overall.
* **Total**: Both encode and decode are O(n), where n is the total number of characters.

**Space Complexity**

* **Encode: O(n)**:
  + The StringBuilder stores the encoded string, which is proportional to the total length of all strings plus delimiters and lengths, i.e., O(n).
  + Excluding the output, the space used during encoding is O(1) (ignoring temporary variables).
* **Decode: O(n)**:
  + The result list stores the decoded strings, which is O(n) for the output.
  + Excluding the output, temporary variables (e.g., substrings) use O(1) space per iteration, but the result list dominates.
* **Note**: The output space is typically not counted unless specified, so auxiliary space is O(1) for both methods.

**Key Takeaways**

* **Delimiter-based encoding**: Using a length prefix with a delimiter (#) ensures unambiguous decoding, even for strings containing special characters or empty strings.
* **Edge cases**:
  + Empty list: Handled by returning an empty string (encode) or empty list (decode).
  + Empty strings: Handled correctly as 0# in the encoded string.
  + Special characters: The length prefix avoids issues with strings containing # or other characters.
* **Alternative approaches**:
  + Use a different delimiter or escape special characters, but the length-prefix method is robust and simple.
  + Serialize to a structured format (e.g., JSON), but this is overkill and less efficient.
* **Related problems**: This problem relates to "Serialize and Deserialize Binary Tree" (encoding structures) and "Group Anagrams" (string manipulation) in the Blind 75 list.
* **Interview tip**: Explain why the # delimiter and length prefix are used to handle edge cases (e.g., empty strings or strings with #). Discuss trade-offs of other delimiters or encoding schemes and highlight the O(1) auxiliary space (excluding output).

1. Product of Array Except Self

**Problem Description**

* Given an integer array nums, return an array result such that result[i] is the product of all the elements of nums except nums[i]. You must solve it without using division and in O(n) time.

class Solution {

public int[] productExceptSelf(int[] nums) {

// Initialize result array with 1s

int[] result = new int[nums.length];

Arrays.fill(result, 1);

// Track product of elements to the left of each index

int pre = 1;

// Track product of elements to the right of each index

int post = 1;

// First pass: Compute product of all elements to the left of each index

for (int i = 0; i < nums.length; i++) {

result[i] = pre; // Store left product for index i

pre = nums[i] \* pre; // Update left product for next index

}

// Second pass: Multiply by product of all elements to the right of each index

for (int i = nums.length - 1; i >= 0; i--) {

result[i] = result[i] \* post; // Multiply left product by right product

post = post \* nums[i]; // Update right product for next index

}

return result;

}

}

**Approach**

* **Initialize result array**: Create an array result of the same length as nums and fill it with 1s to handle multiplication.
* **First pass (left products)**: Iterate from left to right, computing the product of all elements to the left of each index i. Store this in result[i] and update the running product (pre).
* **Second pass (right products)**: Iterate from right to left, multiplying each result[i] by the product of all elements to the right of index i. Update the running right product (post).
* **Avoid division**: By using two passes to compute left and right products, the solution avoids division while calculating the product of all elements except the current one.
* **Return result**: The result array now contains the product of all elements except nums[i] for each index i.

**Time Complexity**

* **O(n)**:
  + The first pass (left products) iterates through the array once, taking O(n) time, where n is the length of nums.
  + The second pass (right products) also takes O(n) time.
  + Initializing the result array with Arrays.fill takes O(n).
  + Total time complexity is O(n).

**Space Complexity**

* **O(1)** (excluding output):
  + The result array is required for the output and is not counted in space complexity unless specified.
  + The variables pre and post use O(1) space.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **No division**: The two-pass approach avoids division, which is critical for handling cases with zeros (division by zero) and ensuring O(n) time complexity.
* **Edge cases**:
  + If the array has one element, the result is [1] (product of no other elements), handled implicitly as pre and post remain 1.
  + Arrays with zeros are handled correctly, as the product for other indices will include the zero, and the index with the zero gets the product of all non-zero elements.
* **Alternative approaches**:
  + Using division (compute total product and divide by nums[i]) is O(n) but fails with zeros and requires special handling.
  + Using two arrays to store left and right products explicitly takes O(n) space, which is less efficient than the current approach.
* **Related problems**: This problem relates to other Blind 75 problems like "Subarray Sum Equals K" (prefix sums) and "Maximum Product Subarray" (tracking products), as it involves computing products over arrays.
* **Interview tip**: Highlight the space efficiency (O(1) excluding output) and the avoidance of division. Be prepared to explain how the two-pass method computes the product for each index and handles edge cases like zeros or small arrays.

1. Valid Palindrome (Two Pointers)

**Problem Description**

* Given a string s, return true if it is a palindrome, ignoring non-alphanumeric characters and case sensitivity, and false otherwise. A palindrome reads the same forward and backward.

class Solution {

public boolean isPalindrome(String s) {

// Initialize two pointers: left at start, right at end

int left = 0, right = s.length() - 1;

// Continue until pointers meet or cross

while (left <= right) {

// Skip non-alphanumeric characters from the left

while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {

left++;

}

// Skip non-alphanumeric characters from the right

while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {

right--;

}

// Compare characters (case-insensitive); return false if they differ

if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right))) {

return false;

}

// Move pointers inward

left++;

right--;

}

return true; // String is a palindrome

}

}

**Approach**

* **Two-pointer technique**: Use two pointers (left and right) starting from the ends of the string and moving toward the center.
* **Skip non-alphanumeric characters**: For each pointer, skip characters that are not letters or digits using Character.isLetterOrDigit.
* **Case-insensitive comparison**: Compare the characters at left and right after converting them to lowercase using Character.toLowerCase. If they differ, return false.
* **Move pointers**: Increment left and decrement right after each valid comparison.
* **Handle edge cases**: If the pointers meet or cross (left > right), the string is a palindrome, so return true.
* **Optimization**: The solution processes the string in-place without creating a new string, and only compares valid characters.

**Time Complexity**

* **O(n)**:
  + The two pointers traverse the string once, with left and right collectively covering all n characters.
  + Checking Character.isLetterOrDigit and Character.toLowerCase is O(1) per character.
  + Total time complexity is O(n), where n is the length of the string.

**Space Complexity**

* **O(1)**:
  + The solution uses only two integer variables (left and right), regardless of input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Two-pointer efficiency**: The two-pointer approach avoids preprocessing the string (e.g., removing non-alphanumeric characters), achieving O(n) time and O(1) space.
* **Edge cases**:
  + Empty strings or single characters are palindromes (returns true).
  + Strings with only non-alphanumeric characters (e.g., ".,") are palindromes (returns true).
  + Mixed case (e.g., "RaCeCaR") is handled by converting to lowercase during comparison.
* **Alternative approaches**:
  + Preprocess the string to keep only alphanumeric characters in lowercase (O(n) time, O(n) space), then use two pointers or reverse and compare.
  + The preprocessing approach is less space-efficient but may be easier to read.
* **Related problems**: This problem relates to "Longest Palindromic Substring" (two-pointer expansion) and "Palindrome Linked List" (two-pointer techniques) in the Blind 75 list.
* **Interview tip**: Highlight the O(1) space efficiency and the use of Character methods to handle alphanumeric checks. Be prepared to discuss preprocessing alternatives and edge cases like non-alphanumeric strings or case sensitivity.

1. 3 Sum (Two Pointers)

**Problem Description**

* Given an integer array nums, return all unique triplets [nums[i], nums[j], nums[k]] such that i != j != k and nums[i] + nums[j] + nums[k] == 0. The solution set must not contain duplicate triplets.

**Code with Comments**

class Solution {

public List<List<Integer>> threeSum(int[] nums) {

// Sort the array to enable two-pointer technique and handle duplicates

Arrays.sort(nums);

// Initialize result list to store triplets

List<List<Integer>> result = new ArrayList<>();

// Iterate through array, fixing the first element of the triplet

for (int i = 0; i < nums.length - 2; i++) {

// Optimization: break if current number is positive (sum cannot be 0)

if (nums[i] > 0) break;

// Skip duplicates for i to avoid duplicate triplets

if (i > 0 && nums[i] == nums[i - 1]) continue;

// Initialize two pointers for the remaining two elements

int l = i + 1, r = nums.length - 1;

while (l < r) {

// Calculate sum of the triplet

int sum = nums[i] + nums[l] + nums[r];

if (sum > 0) {

// Sum too large, move right pointer left to reduce sum

r--;

} else if (sum < 0) {

// Sum too small, move left pointer right to increase sum

l++;

} else {

// Found a valid triplet, add to result

result.add(Arrays.asList(nums[i], nums[l], nums[r]));

// Skip duplicates for left pointer

while (l < r && nums[l] == nums[l + 1]) l++;

// Skip duplicates for right pointer

while (l < r && nums[r] == nums[r - 1]) r--;

// Move both pointers inward to find next unique triplet

l++;

r--;

}

}

}

return result;

}

}

**Approach**

* **Sort the array**: Sort nums to enable the two-pointer technique and simplify duplicate handling.
* **Fix first element**: Iterate through the array with index i as the first element of the triplet, stopping at nums.length - 2 to leave room for two more elements.
* **Optimization for positive numbers**: If nums[i] > 0, break, as the sum of three numbers (with sorted array) cannot be zero.
* **Skip duplicates for** i: If nums[i] equals nums[i-1], skip to avoid duplicate triplets.
* **Two-pointer technique**: For each i, use two pointers (l and r) starting at i + 1 and the end of the array to find two numbers that sum to -nums[i].
* **Check sum**: Compute sum = nums[i] + nums[l] + nums[r]. If sum > 0, decrement r; if sum < 0, increment l; if sum == 0, add the triplet to the result.
* **Handle duplicates in two-pointer**: After finding a valid triplet, skip duplicate values for l and r to ensure unique triplets, then move both pointers inward.
* **Return result**: Return the list of all unique triplets.

**Time Complexity**

* **O(n²)**:
  + Sorting the array takes O(n log n), where n is the length of nums.
  + The outer loop runs O(n) times, and for each i, the two-pointer traversal takes O(n) in the worst case, leading to O(n²) for the nested loops.
  + Total time complexity is O(n log n + n²) = O(n²), as the quadratic term dominates.

**Space Complexity**

* **O(1)** (excluding output and sorting space):
  + The solution uses only a few variables (i, l, r, sum), making auxiliary space O(1).
  + The output list is required for the result and not counted in space complexity unless specified.
  + Sorting may use O(log n) space for the call stack in some implementations (e.g., quicksort), but this is typically considered O(1) auxiliary space for this problem.

**Key Takeaways**

* **Two-pointer with sorting**: Sorting enables efficient two-pointer traversal and duplicate handling, reducing the complexity from O(n³) (brute force) to O(n²).
* **Edge cases**:
  + Empty array or arrays with fewer than 3 elements return an empty list (handled implicitly).
  + Arrays with all zeros (e.g., [0, 0, 0]) or duplicates are handled by skipping duplicates for i, l, and r.
  + The optimization if (nums[i] > 0) break leverages the sorted array to exit early.
* **Alternative approaches**:
  + HashMap-based solution: Fix one element and use a hash map for the other two, but this is complex and still O(n²) with worse constant factors.
  + Brute force (checking all triplets) is O(n³) and inefficient.
* **Related problems**: This problem builds on "Two Sum" (two-pointer or hash map) and relates to "4Sum" and "3Sum Closest" in the Blind 75 list, which extend the same techniques.
* **Interview tip**: Emphasize the importance of sorting for duplicate handling and the two-pointer optimization. Be ready to explain the early break for positive numbers and how duplicates are skipped to ensure uniqueness.

1. Container with Most Water (Two Pointers)

**Problem Description**

* Given an array heights where heights[i] represents the height of a vertical line at index i, find two lines that form a container with the x-axis to hold the maximum amount of water. Return the maximum area (width × height) of the container.

**Code with Comments**

class Solution {

public int maxArea(int[] heights) {

// Initialize two pointers: left at start, right at end

int left = 0, right = heights.length - 1;

// Track maximum area found

int maxA = 0;

// Continue until pointers meet

while (left < right) {

// Calculate area: min height × width (distance between indices)

int area = Math.min(heights[left], heights[right]) \* (right - left);

// Update maximum area if current area is larger

maxA = Math.max(maxA, area);

// Move the pointer of the shorter line inward to potentially increase area

if (heights[left] > heights[right]) {

right--;

} else {

left++;

}

}

return maxA;

}

}

**Approach**

* **Two-pointer technique**: Use two pointers (left and right) starting from the ends of the array and moving toward the center.
* **Calculate area**: For each pair of lines, compute the area as the minimum height (Math.min(heights[left], heights[right])) multiplied by the width (right - left).
* **Track maximum area**: Update the maximum area (maxA) if the current area is larger.
* **Move pointers**: Move the pointer corresponding to the shorter line inward (if heights[left] > heights[right], decrement right; otherwise, increment left). This maximizes the chance of finding a larger area by keeping the taller line and seeking a taller pair.
* **Continue until pointers meet**: Stop when left equals or exceeds right, as no further containers are possible.

**Time Complexity**

* **O(n)**:
  + The two pointers traverse the array once, with left and right collectively covering at most n indices, where n is the length of heights.
  + Each iteration involves constant-time operations (Math.min, Math.max, and pointer updates).
  + Total time complexity is O(n).

**Space Complexity**

* **O(1)**:
  + The solution uses only a few variables (left, right, maxA), regardless of input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Two-pointer efficiency**: The two-pointer approach avoids checking all pairs (O(n²)) by leveraging the fact that moving the shorter line inward is the only way to potentially increase the area (since width decreases with each move).
* **Edge cases**:
  + Arrays with fewer than two elements cannot form a container (problem guarantees n >= 2).
  + Arrays with equal heights or zeros are handled correctly, as the minimum height determines the area.
* **Alternative approaches**:
  + Brute force: Check all pairs of indices (O(n²) time, O(1) space), which is inefficient.
  + No other efficient approach exists without sacrificing time or space significantly.
* **Related problems**: This problem relates to "Trapping Rain Water" (two-pointer or array-based techniques) and "Valid Palindrome" (two-pointer traversal) in the Blind 75 list.
* **Interview tip**: Explain why moving the shorter line is optimal (moving the taller line cannot increase the area due to the minimum height constraint). Be prepared to discuss edge cases like arrays with all equal heights or handling negative heights (not applicable per problem constraints).

1. Best Time to Buy And Sell Stock (Sliding Window)

## **Problem Description**

* Given an array prices where prices[i] is the price of a stock on day i, find the maximum profit achievable by buying on one day and selling on a later day. Return 0 if no profit is possible.

## **Code with Comments**

class Solution {

public int maxProfit(int[] prices) {

// Initialize two pointers: left for buy day, right for sell day

int left = 0, right = 1;

// Track maximum profit

int maxP = 0;

// Continue until right pointer reaches the end

while (right < prices.length) {

// If selling price is higher than buying price, calculate profit

if (prices[right] > prices[left]) {

int profit = prices[right] - prices[left];

// Update maximum profit if current profit is larger

maxP = Math.max(maxP, profit);

} else {

// If selling price is not higher, update buy day to current day

left = right;

}

// Move sell day forward

right++;

}

return maxP;

}

}

**Approach**

* **Two-pointer technique**: Use two pointers (left for buy day, right for sell day) to track potential buy-sell pairs.
* **Compare prices**: If prices[right] > prices[left], calculate the profit (prices[right] - prices[left]) and update the maximum profit (maxP) if the current profit is larger.
* **Update buy day**: If prices[right] <= prices[left], update left to right, as buying at a lower price may lead to a higher profit later.
* **Move sell day**: Increment right to check the next potential sell day.
* **Return result**: Return the maximum profit found, or 0 if no profit is possible.

**Time Complexity**

* **O(n)**:
  + The right pointer iterates through the array once, from index 1 to prices.length - 1.
  + Each iteration involves constant-time operations (comparisons, subtraction, Math.max).
  + Total time complexity is O(n), where n is the length of prices.

**Space Complexity**

* **O(1)**:
  + The solution uses only a few variables (left, right, maxP), regardless of input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Two-pointer efficiency**: The two-pointer approach ensures a single pass through the array, achieving O(n) time while tracking the minimum buy price implicitly by updating left.
* **Edge cases**:
  + Empty array or single element: Returns 0 (no profit possible), handled implicitly as right starts at 1 and loop won’t execute.
  + Decreasing prices (e.g., [7,6,4,3,1]): Returns 0, as left moves to right when no profit is possible.
  + Increasing prices (e.g., [1,2,3,4,5]): Correctly captures maximum profit (e.g., 4 from buying at 1, selling at 5).
* **Alternative approaches**:
  + Single-pass with min-price tracking: Track the minimum price seen so far and compute max profit at each step (same O(n) time, O(1) space, slightly simpler code).
  + Brute force: Check all possible buy-sell pairs (O(n²) time), which is inefficient.
* **Related problems**: This problem relates to "Best Time to Buy and Sell Stock II" (multiple transactions) and "Maximum Subarray" (tracking maximum difference) in the Blind 75 list.
* **Interview tip**: Highlight the optimization of updating the buy day (left) only when a lower price is found, avoiding unnecessary comparisons. Be ready to discuss the single-pass min-price approach and edge cases like decreasing prices.

1. Longest Substring Without Repeating Characters (Sliding Window)

## **Problem Description**

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

## **Code with Comments**

class Solution {

public int lengthOfLongestSubstring(String s) {

// Handle null or empty string case

if (s == null || s.length() == 0) {

return 0;

}

// Handle single character case

if (s.length() == 1) {

return 1;

}

// Initialize two pointers and result

int left = 0, right = 0, ans = 0;

// HashSet to track unique characters in the current window

HashSet<Character> set = new HashSet<>();

// Slide window until right pointer reaches end

while (right < s.length()) {

char c = s.charAt(right);

// If character is already in set, shrink window from left

while (set.contains(c)) {

set.remove(s.charAt(left));

left++;

}

// Add current character to set

set.add(c);

// Update maximum length of substring without repeating characters

ans = Math.max(ans, right - left + 1);

// Move right pointer forward

right++;

}

return ans;

}

}

**Approach**

* **Handle edge cases**: Return 0 for null or empty strings and 1 for single-character strings.
* **Sliding window with HashSet**: Use two pointers (left and right) to maintain a window of unique characters and a HashSet to track characters in the current window.
* **Expand window**: Move the right pointer to include the next character (s.charAt(right)).
* **Handle duplicates**: If the current character is already in the HashSet, shrink the window from the left by removing characters (s.charAt(left)) and incrementing left until the duplicate is removed.
* **Add character**: Add the current character to the HashSet.
* **Track maximum length**: Update the maximum length (ans) by comparing the current window size (right - left + 1) with the previous maximum.
* **Continue until end**: Move right forward until it reaches the end of the string.

**Time Complexity**

* **O(n)**:
  + The right pointer iterates through the string once, covering n characters.
  + The left pointer may move multiple times per right increment, but each character is added and removed at most once (since left never moves backward).
  + HashSet operations (contains, add, remove) are O(1) on average.
  + Total time complexity is O(n), where n is the length of the string.

**Space Complexity**

* **O(min(n, m))**:
  + The HashSet stores at most min(n, m) characters, where n is the string length and m is the size of the character set (e.g., 128 for ASCII, 26 for lowercase letters).
  + Variables left, right, and ans use O(1) space.
  + Total space complexity is O(min(n, m)), often simplified to O(n) in the worst case.

**Key Takeaways**

* **Sliding window efficiency**: The sliding window with a HashSet ensures O(n) time by tracking unique characters and dynamically adjusting the window.
* **Edge cases**:
  + Null or empty string: Returns 0.
  + Single character: Returns 1.
  + Strings with all unique characters (e.g., "abcde"): Returns full length.
  + Strings with all repeated characters (e.g., "aaa"): Returns 1.
* **Alternative approaches**:
  + HashMap with character indices: Instead of removing characters, update left to the index after the last occurrence of the duplicate (still O(n) time, O(min(n, m)) space).
  + Array for character set: If the character set is small (e.g., lowercase letters), use a fixed-size array instead of a HashSet for O(1) space with small m.
* **Related problems**: This problem relates to "Longest Substring with At Most K Distinct Characters" (sliding window) and "Valid Anagram" (character counting) in the Blind 75 list.
* **Interview tip**: Highlight the sliding window technique and the role of the HashSet in maintaining uniqueness. Be prepared to discuss optimizations (e.g., using a HashMap to jump left to the last duplicate’s index + 1) and edge cases like repeated characters.

1. Longest Repeating Character Replacement (Sliding Window)

## **Problem Description**

* Given a string s and an integer k, return the length of the longest substring containing the same letter after performing at most k character replacements. The string consists of uppercase English letters.

**Code with Comments**

class Solution {

public int characterReplacement(String s, int k) {

// Initialize HashMap to store character frequencies in the current window

Map<Character, Integer> occurrence = new HashMap<>();

// Initialize two pointers and result variables

int left = 0, right = 0;

int ans = 0;

// Track the frequency of the most frequent character in the current window

int maxOccurrence = 0;

// Slide the right pointer to expand the window

for (right = 0; right < s.length(); right++) {

// Get current character and update its frequency

char c = s.charAt(right);

occurrence.put(c, occurrence.getOrDefault(c, 0) + 1);

// Update the maximum frequency of any character in the current window

maxOccurrence = Math.max(maxOccurrence, occurrence.get(c));

// Check if window is invalid (replacements needed > k)

// Window size - max frequency = number of replacements needed

if (right - left + 1 - maxOccurrence > k) {

// Shrink window: remove character at left and update frequency

char leftChar = s.charAt(left);

occurrence.put(leftChar, occurrence.get(leftChar) - 1);

left++;

}

// Update maximum valid window length

ans = Math.max(ans, right - left + 1);

}

return ans;

}

}

**Approach**

* **Sliding window with HashMap**: Use two pointers (left and right) to maintain a window and a HashMap to track character frequencies within the window.
* **Expand window**: Move the right pointer, adding each character to the HashMap and updating its frequency with getOrDefault.
* **Track max frequency**: Maintain the frequency of the most frequent character (maxOccurrence) in the current window.
* **Check window validity**: The number of replacements needed is the window size (right - left + 1) minus the frequency of the most frequent character (maxOccurrence). If this exceeds k, the window is invalid.
* **Shrink window**: If the window is invalid, move the left pointer, removing the character at left from the HashMap and decrementing its frequency.
* **Track maximum length**: Update the maximum valid window length (ans) after each step.
* **Optimization**: Avoid recomputing the maximum frequency when shrinking the window, as the new maximum cannot increase the window size significantly (it’s sufficient to update only when expanding).

**Time Complexity**

* **O(n)**:
  + The right pointer iterates through the string once, covering n characters.
  + The left pointer moves at most n times (never moves backward), and each character is added and removed at most once.
  + HashMap operations (put, getOrDefault) are O(1) on average.
  + Total time complexity is O(n), where n is the length of the string.

**Space Complexity**

* **O(m)**:
  + The HashMap stores frequencies for at most m unique characters, where m is the size of the character set (e.g., 26 for uppercase English letters).
  + Variables left, right, ans, and maxOccurrence use O(1) space.
  + Total space complexity is O(m), often considered O(1) since m is fixed at 26 for this problem.

**Key Takeaways**

* **Sliding window with max frequency**: The key insight is that the number of replacements needed is the window size minus the frequency of the most frequent character, allowing efficient window validation.
* **Edge cases**:
  + Empty string: Returns 0 (handled implicitly as right won’t iterate).
  + k >= s.length(): Allows the entire string to be valid (handled by the window logic).
  + Single character type (e.g., "AAAA"): Returns min(s.length(), k + 1) (handled correctly).
* **Alternative approaches**:
  + Brute force: Try each character as the target and check all substrings (O(n²) time).
  + Array instead of HashMap: Use a fixed-size array (e.g., size 26) for frequencies, which is equivalent in time but slightly more space-efficient for small character sets.
* **Related problems**: This problem relates to "Longest Substring Without Repeating Characters" (sliding window) and "Valid Anagram" (character counting) in the Blind 75 list.
* **Interview tip**: Explain the formula window size - max frequency <= k for validating the window. Highlight the optimization of not updating maxOccurrence when shrinking, as it doesn’t affect the result significantly. Be ready to discuss using an array for fixed character sets or handling larger character sets (e.g., Unicode).

1. Minimum Window Substring (Sliding Window)
2. Valid Parentheses (Stack)

## **Problem Description**

* Given a string s containing only parentheses characters ('(', ')', '{', '}', '[', ']'), return true if the string is valid (parentheses are properly nested and closed in the correct order) and false otherwise.

## **Code with Comments**

class Solution {

public boolean isValid(String s) {

// Initialize HashMap to map closing brackets to their corresponding opening brackets

HashMap<Character, Character> mappedBrackets = new HashMap<>();

mappedBrackets.put(')', '(');

mappedBrackets.put('}', '{');

mappedBrackets.put(']', '[');

// Initialize stack to track opening brackets

Stack<Character> stack = new Stack<>();

// Iterate through each character in the string

for (int i = 0; i < s.length(); i++) {

char c = s.charAt(i);

// If character is not a closing bracket, it’s an opening bracket; push to stack

if (!mappedBrackets.containsKey(c)) {

stack.push(c);

} else {

// If stack is empty but we have a closing bracket, return false

if (stack.empty()) {

return false;

}

// Pop the top opening bracket and check if it matches the closing bracket

char topElement = stack.pop();

if (topElement != mappedBrackets.get(c)) {

return false;

}

}

}

// Return true if stack is empty (all brackets matched), false otherwise

return stack.isEmpty();

}

}

**Approach**

* **HashMap for bracket mapping**: Create a HashMap to map each closing bracket (')', '}', ']') to its corresponding opening bracket ('(', '{', '[').
* **Use a stack**: Initialize a Stack to track opening brackets.
* **Iterate through string**: For each character:
  + If it’s not a closing bracket (i.e., not in the HashMap), it’s an opening bracket; push it onto the stack.
  + If it’s a closing bracket, check if the stack is empty (no matching opening bracket, return false). Otherwise, pop the top opening bracket and verify it matches the expected opening bracket for the current closing bracket using the HashMap. If it doesn’t match, return false.
* **Check final state**: After processing all characters, return true if the stack is empty (all brackets matched), or false if brackets remain (unmatched opening brackets).

**Time Complexity**

* **O(n)**:
  + Iterating through the string takes O(n), where n is the length of s.
  + HashMap operations (containsKey, get) and stack operations (push, pop, empty) are O(1).
  + Total time complexity is O(n).

**Space Complexity**

* **O(n)**:
  + The stack can store up to n/2 opening brackets in the worst case (e.g., "((("), leading to O(n) space.
  + The HashMap has a fixed size of 3 key-value pairs, which is O(1).
  + Total space complexity is O(n) due to the stack.

**Key Takeaways**

* **Stack for matching**: A stack is ideal for checking nested structures like parentheses, as it naturally tracks the order of opening brackets for matching with closing ones.
* **Edge cases**:
  + Empty string: Returns true (valid, as no brackets to match).
  + Single character: Returns false (handled by stack being empty for closing brackets or non-empty for opening brackets).
  + Unmatched brackets (e.g., ")(" or "((]"): Returns false due to stack mismatch or non-empty stack.
  + All matched brackets (e.g., "()[]{}"): Returns true.
* **Alternative approaches**:
  + Array-based stack: Use a char array instead of a Stack class for slight memory savings (still O(n) time and space).
  + Counter-based (for simpler cases): Not applicable here due to multiple bracket types and nesting requirements.
* **Related problems**: This problem relates to "Min Stack" (stack operations) and "Generate Parentheses" (valid bracket combinations) in the Blind 75 list.
* **Interview tip**: Highlight the stack’s role in maintaining order and the HashMap for efficient bracket matching. Be ready to discuss edge cases (e.g., empty stack on closing bracket) and why a stack is preferred over recursive or other methods.

1. Find Minimum In Rotated Sorted Array (Binary Search)

## **Problem Description**

* Given a sorted array nums that has been rotated at an unknown pivot point (e.g., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]), find the minimum element. The array has no duplicates, and you must solve it in O(log n) time.

## **Code with Comments**

class Solution {

public int findMin(int[] nums) {

// Initialize pointers and initial answer as first element

int left = 0;

int right = nums.length - 1;

int ans = nums[0];

// Handle single-element array case

if (nums.length == 1) {

return nums[0];

}

// Binary search to find the minimum

while (left <= right) {

// If left element is less than right, array is sorted; return minimum

if (nums[left] < nums[right]) {

ans = Math.min(ans, nums[left]);

break;

}

// Calculate middle index

int mid = left + (right - left) / 2;

// Update answer with minimum of current answer and middle element

ans = Math.min(ans, nums[mid]);

// If mid element is greater than right, minimum is in right half

if (nums[mid] > nums[right]) {

left = mid + 1;

} else {

// Otherwise, minimum is in left half (including mid)

right = mid - 1;

}

}

return ans;

}

}

**Approach**

* **Handle single-element case**: If the array has one element, return it as the minimum.
* **Initialize pointers and answer**: Start with left at 0, right at the end, and ans as nums[0] (a candidate for the minimum).
* **Binary search**:
  + If nums[left] < nums[right], the subarray is sorted, so the minimum is nums[left]; update ans and break.
  + Compute the middle index (mid).
  + Update ans with the minimum of the current ans and nums[mid].
  + If nums[mid] > nums[right], the minimum lies in the right half (after mid), so set left = mid + 1.
  + Otherwise, the minimum is in the left half (including mid), so set right = mid - 1.
* **Return result**: Return the minimum value found in ans.

**Time Complexity**

* **O(log n)**:
  + The binary search halves the search space in each iteration, taking O(log n) iterations, where n is the length of nums.
  + Each iteration involves constant-time operations (comparisons, Math.min).
  + Total time complexity is O(log n).

**Space Complexity**

* **O(1)**:
  + The solution uses only a few variables (left, right, ans, mid), regardless of input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Binary search for rotated arrays**: The key insight is that a rotated sorted array has a pivot point, and binary search can identify the minimum by checking if the middle element is in the sorted or unsorted half.
* **Edge cases**:
  + Single element: Returns nums[0] (handled explicitly).
  + No rotation (e.g., [1,2,3,4]): Returns first element, caught by nums[left] < nums[right].
  + Full rotation or small arrays: Correctly finds the minimum via binary search.
* **Alternative approaches**:
  + Linear scan: Check all elements to find the minimum (O(n) time), which is inefficient and doesn’t meet the O(log n) requirement.
  + Modified binary search: The provided approach is optimal, but slight variations (e.g., always comparing nums[mid] with nums[left]) are possible with similar complexity.
* **Related problems**: This problem relates to "Search in Rotated Sorted Array" (binary search on rotated arrays) and "Find Peak Element" (binary search for extrema) in the Blind 75 list.
* **Interview tip**: Explain how the condition nums[mid] > nums[right] determines the unsorted half where the minimum lies. Be ready to discuss edge cases like no rotation or single-element arrays and why ans is initialized with nums[0].

1. Search In Rotated Sorted Array (Binary Search)

**Problem Description**

* Given a sorted array nums that has been rotated at an unknown pivot point (e.g., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]) and a target value target, return the index of target if it exists in the array, or -1 if it does not. The array has no duplicates, and you must solve it in O(log n) time.

**Code with Comments**

class Solution {

public int search(int[] nums, int target) {

// Initialize two pointers: left at start, right at end

int left = 0;

int right = nums.length - 1;

// Binary search until pointers meet or cross

while (left <= right) {

// Calculate middle index

int mid = (left + right) / 2;

// If target is found at mid, return the index

if (nums[mid] == target) {

return mid;

}

// Check if left half is sorted (nums[left] <= nums[mid])

if (nums[left] <= nums[mid]) {

// If target is outside the sorted left half, search right half

if (target < nums[left] || target > nums[mid]) {

left = mid + 1;

} else {

// Otherwise, target is in the sorted left half; search left

right = mid - 1;

}

} else {

// Right half is sorted (nums[mid] < nums[right])

// If target is outside the sorted right half, search left half

if (target > nums[right] || target < nums[mid]) {

right = mid - 1;

} else {

// Otherwise, target is in the sorted right half; search right

left = mid + 1;

}

}

}

// Target not found, return -1

return -1;

}

}

**Approach**

* **Binary search with rotation handling**: Use two pointers (left and right) to perform a modified binary search on the rotated sorted array.
* **Find middle point**: Compute the middle index (mid) in the current search range.
* **Check for target**: If nums[mid] == target, return mid.
* **Determine sorted half**:
  + If nums[left] <= nums[mid], the left half (from left to mid) is sorted.
    - If target is outside this sorted range (target < nums[left] or target > nums[mid]), search the right half (left = mid + 1).
    - Otherwise, search the left half (right = mid - 1).
  + If nums[left] > nums[mid], the right half (from mid to right) is sorted.
    - If target is outside this sorted range (target > nums[right] or target < nums[mid]), search the left half (right = mid - 1).
    - Otherwise, search the right half (left = mid + 1).
* **Continue until found or not**: If left > right, the target is not in the array, so return -1.

**Time Complexity**

* **O(log n)**:
  + The binary search halves the search space in each iteration, taking O(log n) iterations, where n is the length of nums.
  + Each iteration involves constant-time operations (comparisons and pointer updates).
  + Total time complexity is O(log n).

**Space Complexity**

* **O(1)**:
  + The solution uses only a few variables (left, right, mid), regardless of input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Modified binary search**: The key insight is identifying which half of the array is sorted and determining if the target lies in that sorted portion, allowing O(log n) time despite the rotation.
* **Edge cases**:
  + Single element: Returns index if target matches, else -1 (handled by loop termination).
  + No rotation (sorted array): Works like standard binary search.
  + Target not present: Returns -1 when left > right.
  + Array with two elements: Handled correctly by checking mid and adjusting pointers.
* **Alternative approaches**:
  + Linear scan: Check each element (O(n) time), which doesn’t meet the O(log n) requirement.
  + Find pivot first: Identify the rotation point and then perform binary search, but this requires two O(log n) passes, which is less efficient than the single-pass approach.
* **Related problems**: This problem relates to "Find Minimum in Rotated Sorted Array" (binary search for minimum) and "Search in Rotated Sorted Array II" (handles duplicates) in the Blind 75 list.
* **Interview tip**: Explain how you determine the sorted half using nums[left] <= nums[mid]. Be ready to discuss edge cases (e.g., no rotation, single element) and how to extend the solution for duplicates (e.g., checking for equality in conditions).

1. Reverse Linked List (LinkedList)

## Problem Description

* Given the head of a singly linked list, reverse the list and return the new head. Each node contains an integer value and a pointer to the next node.

**Code with Comments**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode reverseList(ListNode head) {

// Initialize previous pointer as null and current pointer as head

ListNode prev = null;

ListNode curr = head;

// Iterate until current pointer reaches null (end of list)

while (curr != null) {

// Store next node to avoid losing it

ListNode temp = curr.next;

// Reverse the link: point current node's next to previous node

curr.next = prev;

// Move previous pointer to current node

prev = curr;

// Move current pointer to next node (stored in temp)

curr = temp;

}

// Return previous pointer, which is now the head of the reversed list

return prev;

}

}

**Approach**

* **Iterative reversal**: Use three pointers (prev, curr, and temp) to reverse the links of the list in-place.
* **Initialize pointers**: Set prev to null (as the new tail’s next will be null) and curr to head.
* **Iterate through list**:
  + Save the next node in temp to avoid losing it when reversing the link.
  + Reverse the current node’s link by setting curr.next to prev.
  + Move prev to curr and curr to temp (the next node).
* **Return new head**: After the loop, prev points to the new head of the reversed list.
* **Optimization**: The solution modifies the list in-place without requiring extra memory beyond pointers.

**Time Complexity**

* **O(n)**:
  + The algorithm traverses the linked list once, processing each of the n nodes exactly once.
  + Each iteration involves constant-time operations (pointer updates).
  + Total time complexity is O(n), where n is the number of nodes in the list.

**Space Complexity**

* **O(1)**:
  + The solution uses only three pointers (prev, curr, temp), regardless of the list size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **In-place reversal**: The iterative approach efficiently reverses the list by updating pointers without requiring extra space, making it optimal for linked lists.
* **Edge cases**:
  + Empty list (head == null): Returns null (handled implicitly as loop doesn’t execute).
  + Single node: Returns the same node (no reversal needed, handled correctly as curr.next is null).
  + Multiple nodes: Correctly reverses all links (e.g., 1->2->3 becomes 3->2->1).
* **Alternative approaches**:
  + Recursive solution: Uses recursion to reverse the list (O(n) time, O(n) space due to call stack), less space-efficient but more elegant for some.
  + Brute force (create new list): Copy values to a new list in reverse order (O(n) time, O(n) space), which is inefficient.
* **Related problems**: This problem relates to "Merge Two Sorted Lists" (linked list manipulation), "Palindrome Linked List" (reversing part of a list), and "Reverse Linked List II" (partial reversal) in the Blind 75 list.
* **Interview tip**: Emphasize the in-place nature of the solution and the role of the temp pointer in preserving the next node. Be ready to explain the recursive approach and discuss edge cases like empty or single-node lists.

1. Merge Two Sorted Linked Lists (LinkedList)

## Problem Description

* Given the heads of two sorted singly-linked lists list1 and list2, merge them into one sorted singly-linked list and return its head. Each node contains an integer value and a pointer to the next node.

## Code with Comments

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode mergeTwoLists(ListNode list1, ListNode list2) {

// Initialize a dummy node to simplify merging

ListNode dummy = new ListNode(0);

// Pointer to build the merged list

ListNode merge = dummy;

// While both lists have nodes, compare and merge in sorted order

while (list1 != null && list2 != null) {

if (list1.val <= list2.val) {

// Add list1 node to merged list and move list1 pointer

merge.next = list1;

list1 = list1.next;

} else {

// Add list2 node to merged list and move list2 pointer

merge.next = list2;

list2 = list2.next;

}

// Move merge pointer to the newly added node

merge = merge.next;

}

// Attach remaining nodes from list1, if any

if (list1 != null) {

merge.next = list1;

}

// Attach remaining nodes from list2, if any

else {

merge.next = list2;

}

// Return head of merged list (skip dummy node)

return dummy.next;

}

}

**Approach**

* **Dummy node**: Create a dummy node to simplify the merging process by providing a starting point for the merged list.
* **Merge pointer**: Use a merge pointer to build the result list, starting at the dummy node.
* **Compare and merge**: While both list1 and list2 are non-null, compare their current nodes’ values:
  + If list1.val <= list2.val, append list1 node to the merged list and advance list1.
  + Otherwise, append list2 node and advance list2.
  + Move the merge pointer to the newly added node.
* **Handle remaining nodes**: After one list is exhausted, append the remaining nodes from the other list (if any) to the merged list.
* **Return result**: Return dummy.next as the head of the merged list, skipping the dummy node.

**Time Complexity**

* **O(n + m)**:
  + The algorithm processes each node in list1 (length n) and list2 (length m) exactly once.
  + Each iteration involves constant-time operations (comparisons, pointer updates).
  + Total time complexity is O(n + m), where n and m are the lengths of the input lists.

**Space Complexity**

* **O(1)**:
  + The solution uses only a few pointers (dummy, merge, and the input list pointers), regardless of input size.
  + The merged list reuses existing nodes, so no additional memory is allocated beyond the output.
  + Total space complexity is O(1), excluding the output.

**Key Takeaways**

* **Dummy node for simplicity**: The dummy node eliminates the need for special handling of the head, making the code cleaner and easier to manage.
* **Edge cases**:
  + Both lists empty: Returns null (handled by dummy.next).
  + One list empty: Returns the other list (handled by attaching remaining nodes).
  + Lists of different lengths: Remaining nodes are appended correctly.
  + Single node in both lists: Merges correctly based on value comparison.
* **Alternative approaches**:
  + Recursive solution: Merge recursively by selecting the smaller head and linking to the merged tail (O(n + m) time, O(n + m) space due to call stack).
  + Brute force: Collect all values, sort, and create a new list (O((n + m) log (n + m)) time, O(n + m) space), which is inefficient.
* **Related problems**: This problem relates to "Reverse Linked List" (linked list manipulation), "Merge k Sorted Lists" (extends merging to multiple lists), and "Sort List" (sorting linked lists) in the Blind 75 list.
* **Interview tip**: Highlight the use of the dummy node for simplicity and the in-place merging for O(1) space. Be prepared to explain the recursive approach and discuss edge cases like empty lists or lists with negative values.

1. Linked List Cycle Detection (LinkedList)

**Problem Description**

* Given the head of a singly-linked list, determine if there is a cycle in the list. A cycle exists if a node can be reached again by following the next pointers. Return true if there is a cycle, false otherwise.

## **Code with Comments**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public boolean hasCycle(ListNode head) {

// Handle empty list case

if (head == null) {

return false;

}

// Initialize two pointers: fast moves two steps, slow moves one step

ListNode fast = head;

ListNode slow = head;

// Continue until fast reaches the end or a cycle is detected

while (fast != null && fast.next != null) {

// Move fast pointer by two steps

fast = fast.next.next;

// Move slow pointer by one step

slow = slow.next;

// If fast and slow meet, a cycle exists

if (fast == slow) {

return true;

}

}

// If fast reaches null, no cycle exists

return false;

}

}

**Approach**

* **Handle edge case**: If the list is empty (head == null), return false as no cycle is possible.
* **Floyd’s Cycle Detection Algorithm (Two-pointer)**: Use two pointers, fast and slow, both starting at head.
  + Move fast two steps and slow one step per iteration.
  + Check if fast and slow meet (i.e., point to the same node).
* **Cycle detection**: If fast and slow meet, a cycle exists, so return true.
* **No cycle**: If fast or fast.next becomes null, the list has an end, so return false.
* **Optimization**: The algorithm uses constant space and detects a cycle efficiently by leveraging the relative speeds of the pointers.

**Time Complexity**

* **O(n)**:
  + In a list with n nodes, if there is no cycle, the fast pointer reaches the end in O(n) steps.
  + If there is a cycle, the fast pointer catches up to the slow pointer in O(n) steps (proportional to the cycle length and the distance to the cycle start).
  + Each iteration involves constant-time pointer updates.
  + Total time complexity is O(n).

**Space Complexity**

* **O(1)**:
  + The solution uses only two pointers (fast and slow), regardless of the list size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Floyd’s Cycle Detection (Hare and Tortoise)**: This algorithm efficiently detects cycles by using two pointers moving at different speeds, making it ideal for linked list problems with O(1) space.
* **Edge cases**:
  + Empty list: Returns false (handled explicitly).
  + Single node without cycle: Returns false (fast reaches null immediately).
  + Cycle with one or more nodes: Returns true when fast and slow meet.
* **Alternative approaches**:
  + HashSet: Store visited nodes in a HashSet and check for duplicates (O(n) time, O(n) space), less space-efficient.
  + Modify list: Mark visited nodes by changing values or pointers (O(n) time, O(1) space), but destructive and not allowed in some cases.
* **Related problems**: This problem relates to "Linked List Cycle II" (finding cycle start), "Reverse Linked List" (linked list manipulation), and "Palindrome Linked List" (using fast-slow pointers) in the Blind 75 list.
* **Interview tip**: Explain how Floyd’s algorithm works (why fast and slow pointers meet in a cycle) and its O(1) space advantage. Be prepared to discuss the alternative HashSet approach and edge cases like single-node cycles.

1. Reorder List (LinkedList)

## **Problem Description**

* Given the head of a singly-linked list, reorder it in-place such that the nodes follow the pattern: L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → .... You may not modify the values in the nodes, only the pointers.

**Code with Comments**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public void reorderList(ListNode head) {

// Check if the list is empty; if so, return as no reordering is needed

if (head == null) {

return;

}

// Initialize slow and fast pointers to find the middle of the list

ListNode slow = head, fast = head;

// Move slow pointer one step and fast pointer two steps until fast reaches the end

while (fast != null && fast.next != null) {

// Move slow pointer to the next node

slow = slow.next;

// Move fast pointer two steps forward

fast = fast.next.next;

}

// Initialize pointers for reversing the second half of the list

ListNode prev = null, curr = slow, temp;

// Reverse the second half of the list

while (curr != null) {

// Store the next node to avoid losing it

temp = curr.next;

// Reverse the link by pointing current node to previous

curr.next = prev;

// Move previous pointer to current node

prev = curr;

// Move current pointer to the next node (stored in temp)

curr = temp;

}

// Initialize pointers for merging: first for the first half, second for the reversed second half

ListNode first = head, second = prev;

// Merge the two halves alternately

while (second.next != null) {

// Store the next node of the first half

temp = first.next;

// Link first node to the head of the second half

first.next = second;

// Move first pointer to the next node of the original first half

first = temp;

// Store the next node of the second half

temp = second.next;

// Link second node to the next node of the first half

second.next = first;

// Move second pointer to the next node of the reversed second half

second = temp;

}

}

}

**Approach**

* **Handle edge case**: If the list is empty (head == null), return immediately as no reordering is needed.
* **Find the middle**: Use two pointers, slow and fast, where fast moves twice as fast as slow. When fast reaches the end, slow is at the middle node (or just past it for even-length lists).
* **Reverse second half**: Reverse the second half of the list starting from slow using an iterative reversal (similar to Reverse Linked List), creating a new list starting from prev.
* **Merge halves**: Merge the first half (starting at head) and the reversed second half (starting at prev) by interleaving nodes: connect a node from the first half to a node from the second half, and vice versa, until the second half is exhausted.
* **Optimization**: The solution is in-place, modifying pointers without creating new nodes, and handles the merging carefully to avoid cycles.

**Time Complexity**

* **O(n)**:
  + Finding the middle takes O(n/2) steps, as fast traverses the list at double speed.
  + Reversing the second half takes O(n/2) for the nodes after the middle.
  + Merging the two halves takes O(n/2) to interleave the nodes.
  + Total time complexity is O(n), where n is the number of nodes in the list.

**Space Complexity**

* **O(1)**:
  + The solution uses a constant number of pointers (slow, fast, prev, curr, temp, first, second), regardless of list size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Three-step process**: The solution combines finding the middle (fast-slow pointers), reversing the second half (iterative reversal), and merging halves (pointer manipulation), all in-place.
* **Edge cases**:
  + Empty list or single node: No reordering needed (handled by early return or trivial merge).
  + Two nodes: Reorders correctly (e.g., 1->2 becomes 1->2).
  + Odd or even length: Middle node identification handles both cases, with merging stopping when the second half is exhausted.
* **Alternative approaches**:
  + Store nodes in an array: Collect nodes, reorder by index manipulation, and relink (O(n) time, O(n) space), less space-efficient.
  + Recursive merging: Possible but complex and uses O(n) space due to the call stack.
* **Related problems**: This problem relates to "Reverse Linked List" (reversal technique), "Merge Two Sorted Lists" (merging technique), and "Palindrome Linked List" (fast-slow pointers) in the Blind 75 list.
* **Interview tip**: Explain the three distinct phases (middle finding, reversal, merging) and emphasize the O(1) space efficiency. Be ready to discuss edge cases (e.g., lists with 1 or 2 nodes) and why the second.next != null condition prevents issues in the merge phase.

1. Remove Nth Node From End of List (LinkedList)

## **Problem Description**

* Given the head of a singly-linked list and an integer n, remove the nth node from the end of the list and return the head of the modified list. Assume n is valid (1 ≤ n ≤ length of list

**Code with Comments**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode removeNthFromEnd(ListNode head, int n) {

// Create a dummy node pointing to head to handle edge cases (e.g., removing first node)

ListNode dummy = new ListNode(1);

// Link dummy node to the head of the list

dummy.next = head;

// Initialize front pointer at dummy for maintaining gap

ListNode front = dummy;

// Initialize back pointer at dummy to track node before the one to remove

ListNode back = dummy;

// Move front pointer n+1 steps ahead to create a gap of n nodes

for (int i = 0; i <= n; i++) {

// Advance front pointer

front = front.next;

}

// Move both pointers until front reaches the end

while (front != null) {

// Advance front pointer

front = front.next;

// Advance back pointer, maintaining the gap

back = back.next;

}

// Remove the nth node from end by updating back's next pointer

back.next = back.next.next;

// Return the head of the modified list (skip dummy node)

return dummy.next;

}

}

**Approach**

* **Dummy node**: Create a dummy node pointing to head to simplify edge cases, such as removing the first node.
* **Two-pointer technique**: Use two pointers, front and back, both starting at the dummy node.
* **Create gap**: Move front n+1 steps ahead to create a gap of n nodes between front and back.
* **Move pointers**: Advance both front and back until front reaches the end (null). At this point, back is at the node just before the one to be removed.
* **Remove node**: Update back.next to skip the nth node from the end by linking to the node after it.
* **Return result**: Return dummy.next as the head of the modified list.

**Time Complexity**

* **O(n)**:
  + Moving the front pointer n+1 steps takes O(n) time in the worst case (if n is equal to the list length).
  + Moving both pointers until front reaches the end takes O(n - n) = O(n) steps, where n is the length of the list.
  + The removal operation (updating back.next) is O(1).
  + Total time complexity is O(n).

**Space Complexity**

* **O(1)**:
  + The solution uses only a few pointers (dummy, front, back), regardless of the list size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Two-pointer with gap**: The two-pointer technique with a fixed gap efficiently locates the node to remove in a single pass, avoiding the need to compute the list length.
* **Edge cases**:
  + Single node (n = 1): Dummy node ensures correct removal of the only node, returning null.
  + Remove first node (n = length): Dummy node simplifies handling by allowing back to point to the dummy.
  + Multiple nodes: Correctly removes the nth node from the end.
* **Alternative approaches**:
  + Two-pass solution: First pass to find list length, second pass to remove the nth node from the end (O(n) time, O(1) space), but less efficient due to extra traversal.
  + Recursive: Possible but complex and uses O(n) space due to the call stack.
* **Related problems**: This problem relates to "Linked List Cycle" (two-pointer technique), "Reverse Linked List" (linked list manipulation), and "Merge Two Sorted Lists" (pointer updates) in the Blind 75 list.
* **Interview tip**: Emphasize the role of the dummy node in handling edge cases and the two-pointer technique for O(n) single-pass efficiency. Be ready to explain why the gap is n+1 and discuss alternative approaches like the two-pass method.

1. Merge K Sorted Lists (LinkedList)
2. Invert Binary Tree (Trees)

## Problem Description

* Given the root of a binary tree, invert the tree (i.e., swap every left and right child for each node) and return its root.

**Code with Comments**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public TreeNode invertTree(TreeNode root) {

// Base case: if root is null, return null

if (root == null) {

return null;

}

// Recursively invert the right subtree

TreeNode right = invertTree(root.right);

// Recursively invert the left subtree

TreeNode left = invertTree(root.left);

// Swap the left and right children of the current node

root.left = right;

// Assign the inverted left subtree to the right child

root.right = left;

// Return the current node after inversion

return root;

}

}

**Approach**

* **Recursive inversion**: Use recursion to invert the binary tree by swapping the left and right children of each node.
* **Base case**: If the current node (root) is null, return null as there is nothing to invert.
* **Recurse on subtrees**:
  + Recursively invert the right subtree and store the result in right.
  + Recursively invert the left subtree and store the result in left.
* **Swap children**: Assign the inverted right subtree to the left child (root.left = right) and the inverted left subtree to the right child (root.right = left).
* **Return node**: Return the current node after its subtrees are inverted and children swapped.
* **Optimization**: The solution modifies the tree in-place, requiring no additional data structures beyond the recursion stack.

**Time Complexity**

* **O(n)**:
  + Each node in the tree is visited exactly once during the recursive traversal.
  + Swapping children and recursive calls are O(1) per node.
  + Total time complexity is O(n), where n is the number of nodes in the tree.

**Space Complexity**

* **O(h)**:
  + The space complexity is determined by the recursion stack, which depends on the height h of the tree.
  + In the worst case (skewed tree), h = n, so space is O(n).
  + In a balanced tree, h = log n, so space is O(log n).
  + Overall, space complexity is O(h), where h is the height of the tree.

**Key Takeaways**

* **Recursive simplicity**: The recursive approach is intuitive and mirrors the tree’s structure, making it easy to implement and understand.
* **Edge cases**:
  + Empty tree (root == null): Returns null (handled by base case).
  + Single node: No children to swap, returns the node itself.
  + Skewed or balanced trees: Works correctly for any tree structure.
* **Alternative approaches**:
  + Iterative solution: Use a queue or stack for level-order or depth-first traversal to swap children (O(n) time, O(w) space, where w is the maximum width of the tree).
  + The recursive approach is typically preferred for its clarity and similar time complexity.
* **Related problems**: This problem relates to "Binary Tree Maximum Path Sum" (tree traversal), "Symmetric Tree" (comparing mirrored subtrees), and "Maximum Depth of Binary Tree" (recursive tree processing) in the Blind 75 list.
* **Interview tip**: Highlight the recursive nature of the solution and its in-place modification. Be ready to explain the iterative approach using a queue and discuss edge cases like empty or single-node trees.

1. Maximum Depth of Binary Tree (Trees)

## Problem Description

* 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.

## Code with Comments

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public int maxDepth(TreeNode root) {

// Base case: if root is null, return 0 (empty tree has depth 0)

if (root == null) {

return 0;

}

// Recursively compute the depth of the left subtree

int leftDepth = maxDepth(root.left);

// Recursively compute the depth of the right subtree

int rightDepth = maxDepth(root.right);

// Return the maximum of left and right depths, plus 1 for the current node

return Math.max(leftDepth, rightDepth) + 1;

}

}

**Approach**

* **Recursive depth calculation**: Use recursion to compute the maximum depth of the binary tree.
* **Base case**: If the current node (root) is null, return 0, as an empty tree has no depth.
* **Recurse on subtrees**:
  + Compute the depth of the left subtree by calling maxDepth(root.left).
  + Compute the depth of the right subtree by calling maxDepth(root.right).
* **Combine results**: The depth of the current node is the maximum of the left and right subtree depths, plus 1 to account for the current node.
* **Return result**: Return the computed depth for the current subtree.

**Time Complexity**

* **O(n)**:
  + Each node in the tree is visited exactly once during the recursive traversal.
  + Operations per node (comparisons, Math.max) are O(1).
  + Total time complexity is O(n), where n is the number of nodes in the tree.

**Space Complexity**

* **O(h)**:
  + The space complexity is determined by the recursion stack, which depends on the height h of the tree.
  + In the worst case (skewed tree), h = n, so space is O(n).
  + In a balanced tree, h = log n, so space is O(log n).
  + Overall, space complexity is O(h), where h is the height of the tree.

**Key Takeaways**

* **Recursive simplicity**: The recursive approach is concise and naturally follows the tree’s structure, making it easy to compute the maximum depth.
* **Edge cases**:
  + Empty tree (root == null): Returns 0 (handled by base case).
  + Single node: Returns 1 (no children, depth is 1).
  + Skewed or balanced trees: Works correctly for any tree structure.
* **Alternative approaches**:
  + Iterative solution: Use a queue for level-order traversal (BFS), tracking the depth of each level (O(n) time, O(w) space, where w is the maximum width of the tree).
  + Iterative DFS with a stack: Mimic recursion with a stack (O(n) time, O(h) space), but less intuitive than recursion.
* **Related problems**: This problem relates to "Invert Binary Tree" (tree traversal), "Balanced Binary Tree" (height comparison), and "Binary Tree Maximum Path Sum" (recursive tree processing) in the Blind 75 list.
* **Interview tip**: Highlight the recursive solution’s clarity and its O(n) time efficiency. Be prepared to explain the iterative BFS approach and discuss edge cases like empty or skewed trees.

1. Same Tree (Trees)

## Problem Description

* Given the roots of two binary trees p and q, return true if they are structurally identical and have the same node values, and false otherwise.

## Code with Comments

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

// Base case: if both nodes are null, trees are identical at this point

if (p == null && q == null) {

return true;

}

// Base case: if one node is null and the other is not, trees differ

if (p == null || q == null) {

return false;

}

// Check if current nodes have different values

if (p.val != q.val) {

return false;

}

// Recursively check if left and right subtrees are identical

return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);

}

}

**Approach**

* **Recursive comparison**: Use recursion to compare the two trees node by node.
* **Base cases**:
  + If both nodes are null, return true (empty subtrees are identical).
  + If one node is null and the other is not, return false (structure differs).
  + If the values of the current nodes differ, return false.
* **Recursive step**: Recursively check if the left subtrees (p.left, q.left) and right subtrees (p.right, q.right) are identical. Combine results with && to ensure both subtrees match.
* **Return result**: Return true only if all checks pass for the current node and its subtrees.

**Time Complexity**

* **O(min(n, m))**:
  + Each node in the smaller tree is visited exactly once, where n and m are the number of nodes in trees p and q, respectively.
  + Comparisons (null checks, value checks) are O(1) per node.
  + Total time complexity is O(min(n, m)), as recursion stops when one tree is exhausted.
  + In the worst case (trees are identical), all nodes are visited.

**Space Complexity**

* **O(min(h\_p, h\_q))**:
  + The space complexity is determined by the recursion stack, which depends on the height of the smaller tree (h\_p for p, h\_q for q).
  + In the worst case (skewed tree), the height is O(n) or O(m), so space is O(min(n, m)).
  + In a balanced tree, the height is O(log n) or O(log m), so space is O(min(log n, log m)).
  + Overall, space complexity is O(min(h\_p, h\_q)).

**Key Takeaways**

* **Recursive elegance**: The recursive solution is concise and mirrors the tree structure, making it intuitive for comparing trees.
* **Edge cases**:
  + Both trees empty: Returns true (handled by first base case).
  + One tree empty, other not: Returns false (handled by second base case).
  + Same structure, different values: Returns false (handled by value check).
  + Identical trees: Returns true after recursive checks.
* **Alternative approaches**:
  + Iterative solution: Use a queue or stack for level-order or depth-first traversal to compare nodes (O(min(n, m)) time, O(w) space, where w is the maximum width of the smaller tree).
  + The recursive approach is typically preferred for its simplicity.
* **Related problems**: This problem relates to "Symmetric Tree" (comparing mirrored subtrees), "Invert Binary Tree" (tree manipulation), and "Maximum Depth of Binary Tree" (recursive tree traversal) in the Blind 75 list.
* **Interview tip**: Highlight the recursive approach’s clarity and how it handles structural and value comparisons. Be ready to discuss the iterative approach and edge cases like empty trees or trees with different structures.

1. Subtree of Another Tree (Trees)

## Problem Description

* Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values as subRoot, and false otherwise. A subtree is a node in root and all its descendants.

**Code with Comments**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

// Main function to check if subRoot is a subtree of root

public boolean isSubtree(TreeNode root, TreeNode subRoot) {

// Base case: if subRoot is null, an empty tree is a subtree of any tree

if (subRoot == null) {

return true;

}

// Base case: if root is null but subRoot is not, subRoot cannot be a subtree

if (root == null) {

return false;

}

// Check if the tree rooted at current node matches subRoot exactly

if (sameTree(root, subRoot)) {

return true;

}

// Recursively check if subRoot is a subtree of left or right subtree

return isSubtree(root.left, subRoot) ||

isSubtree(root.right, subRoot);

}

// Helper function to check if two trees are identical

public boolean sameTree(TreeNode root, TreeNode subRoot) {

// Base case: if both nodes are null, trees are identical

if (root == null && subRoot == null) {

return true;

}

// Check if both nodes exist and have the same value

if (root != null && subRoot != null && root.val == subRoot.val) {

// Recursively check if left and right subtrees are identical

return sameTree(root.left, subRoot.left) &&

sameTree(root.right, subRoot.right);

}

// Return false if one node is null or values differ

return false;

}

}

**Approach**

* **Two-part recursion**:
  + Use a main function isSubtree to check if subRoot is a subtree of root.
  + Use a helper function sameTree to check if two trees are identical.
* **Base cases in** isSubtree:
  + If subRoot is null, return true (an empty tree is a subtree of any tree).
  + If root is null but subRoot is not, return false (no subtree possible).
* **Check for match**: Call sameTree to check if the tree rooted at root is identical to subRoot.
* **Recurse on subtrees**: If no match is found, recursively check if subRoot is a subtree of root.left or root.right using || to combine results.
* **Helper function** sameTree:
  + If both nodes are null, return true (identical empty trees).
  + If both nodes exist and have the same value, recursively check their left and right subtrees.
  + Otherwise, return false (trees differ in structure or value).
* **Optimization**: The solution leverages recursive tree traversal to check all possible subtrees efficiently.

**Time Complexity**

* **O(n \* min(m, n))**:
  + Let n be the number of nodes in root and m be the number of nodes in subRoot.
  + The isSubtree function visits each node in root once, taking O(n) recursive calls.
  + For each node in root, sameTree may be called, which takes O(min(m, n)) to compare the subtree rooted at that node with subRoot (stops early if trees differ).
  + Total time complexity is O(n \* min(m, n)) in the worst case (e.g., when many subtrees need checking).
  + In practice, early termination in sameTree reduces the constant factor.

**Space Complexity**

* **O(max(h\_r, h\_s))**:
  + The space complexity is determined by the recursion stack.
  + For isSubtree, the maximum depth is the height of root, h\_r (O(n) for skewed, O(log n) for balanced).
  + For sameTree, the maximum depth is the height of the smaller tree, min(h\_r, h\_s).
  + Overall, space complexity is O(max(h\_r, h\_s)), where h\_r and h\_s are the heights of root and subRoot, respectively.

**Key Takeaways**

* **Recursive subtree checking**: The solution combines checking for identical trees with recursive traversal to test all possible subtrees, making it intuitive and effective.
* **Edge cases**:
  + Empty subRoot: Returns true (handled by base case).
  + Empty root, non-empty subRoot: Returns false (handled by base case).
  + Single node trees: Handled by sameTree checking values and structure.
  + Non-matching trees: Correctly returns false via recursive checks.
* **Alternative approaches**:
  + Iterative solution: Use BFS or DFS with a queue/stack to check subtrees (same time complexity, O(w) space where w is the tree width).
  + Tree serialization: Convert both trees to strings (e.g., pre-order traversal) and check if subRoot’s string is a substring of root’s string (O(n + m) time, O(n + m) space), but less efficient for large trees.
* **Related problems**: This problem relates to "Same Tree" (identical tree checking), "Symmetric Tree" (mirrored tree comparison), and "Binary Tree Maximum Path Sum" (tree traversal) in the Blind 75 list.
* **Interview tip**: Explain the role of the sameTree helper function and the recursive traversal in isSubtree. Be ready to discuss the time complexity (why it’s O(n \* min(m, n))) and alternative approaches like serialization or iterative traversal.

1. Lowest Common Ancestor of a Binary Search Tree (Trees)

## Problem Description

* Given a binary search tree (BST) with root node root and two nodes p and q in the tree, find the lowest common ancestor (LCA) of p and q. The LCA is the lowest node in the tree that has both p and q as descendants (a node can be a descendant of itself). The BST guarantees that for any node, all values in its left subtree are less than the node’s value, and all values in its right subtree are greater.

**Code with Comments**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

// Function to find the lowest common ancestor of nodes p and q in a BST

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

// Base case: if root, p, or q is null, return null

if (root == null || p == null || q == null) {

return null;

}

// If both p and q values are less than root, LCA is in the left subtree

if (Math.max(p.val, q.val) < root.val) {

// Recursively search in the left subtree

return lowestCommonAncestor(root.left, p, q);

}

// If both p and q values are greater than root, LCA is in the right subtree

else if (Math.min(p.val, q.val) > root.val) {

// Recursively search in the right subtree

return lowestCommonAncestor(root.right, p, q);

}

// If one value is less and the other is greater (or equal), root is the LCA

else {

return root;

}

}

}

**Approach**

* **Leverage BST property**: Use the BST property that all nodes in the left subtree have values less than the current node, and all nodes in the right subtree have values greater.
* **Base case**: If root, p, or q is null, return null (no LCA possible).
* **Compare values**:
  + If both p.val and q.val are less than root.val (using Math.max(p.val, q.val) < root.val), both nodes lie in the left subtree, so recurse on root.left.
  + If both p.val and q.val are greater than root.val (using Math.min(p.val, q.val) > root.val), both nodes lie in the right subtree, so recurse on root.right.
  + Otherwise, the nodes are split (one in the left subtree or equal to root, one in the right subtree or equal to root), or one of the nodes is the root itself, making root the LCA.
* **Return result**: Return the LCA node (either root or the result of recursive calls).

**Time Complexity**

* **O(h)**:
  + The algorithm traverses a single path from the root to the LCA, where h is the height of the tree.
  + Each recursive call performs O(1) operations (comparisons).
  + In a balanced BST, h = O(log n), where n is the number of nodes.
  + In a skewed BST, h = O(n).
  + Overall, time complexity is O(h), typically O(log n) for balanced trees.

**Space Complexity**

* **O(h)**:
  + The space complexity is determined by the recursion stack, which depends on the height h of the tree.
  + In a balanced BST, h = O(log n), so space is O(log n).
  + In a skewed BST, h = O(n), so space is O(n).
  + Overall, space complexity is O(h).

**Key Takeaways**

* **BST property exploitation**: The solution efficiently uses the BST property to direct the search to the left or right subtree, reducing the search space in each step.
* **Edge cases**:
  + Empty tree (root == null): Returns null (handled by base case).
  + One of p or q is null: Returns null (handled by base case).
  + p or q is the root: Correctly returns the root if it’s the LCA.
  + p and q are the same node: Returns that node (handled when one value equals root.val).
* **Alternative approaches**:
  + Iterative solution: Use a loop instead of recursion to traverse the tree based on value comparisons (O(h) time, O(1) space).
  + General binary tree approach: For non-BSTs, find paths to p and q and identify the last common node (O(n) time, O(n) space), less efficient for BSTs.
* **Related problems**: This problem relates to "Lowest Common Ancestor of a Binary Tree" (non-BST version), "Binary Search Tree Iterator" (BST traversal), and "Validate Binary Search Tree" (BST properties) in the Blind 75 list.
* **Interview tip**: Highlight how the BST property simplifies the solution compared to a general binary tree. Be ready to explain the iterative approach and edge cases like when p or q is the root or identical.

1. Binary Tree Level Order Traversal (Trees)

## Problem Description

* 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) as a list of lists, where each inner list contains the node values at a specific level.

**Code with Comments**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

// Initialize a list to store the result (list of lists for each level)

private List<List<Integer>> res = new ArrayList<>();

// Main function to perform level-order traversal

public List<List<Integer>> levelOrder(TreeNode root) {

// If the tree is empty, return the empty result list

if (root == null) {

return res;

}

// Start recursive traversal from the root at level 0

order(root, 0);

// Return the completed result list

return res;

}

// Helper function to perform recursive level-order traversal

private void order(TreeNode node, int level) {

// If the result list doesn't have a list for the current level, create one

if (res.size() == level) {

res.add(new ArrayList<Integer>());

}

// Add the current node's value to the list for its level

res.get(level).add(node.val);

// Recursively process the left child at the next level

if (node.left != null) {

order(node.left, level + 1);

}

// Recursively process the right child at the next level

if (node.right != null) {

order(node.right, level + 1);

}

}

}

**Approach**

* **Recursive level-order traversal**: Use a recursive depth-first search (DFS) approach to traverse the tree, tracking the level of each node.
* **Initialize result**: Create a global List<List<Integer>> (res) to store the node values for each level.
* **Base case**: If the input root is null, return the empty result list.
* **Helper function (**order**)**:
  + Take the current node and its level as parameters.
  + If the result list doesn’t have a list for the current level (res.size() == level), create a new ArrayList for that level.
  + Add the current node’s value to the list for its level.
  + Recursively process the left child at level + 1 if it exists.
  + Recursively process the right child at level + 1 if it exists.
* **Return result**: After the recursive traversal, return the result list containing the level-order traversal.

**Time Complexity**

* **O(n)**:
  + Each node in the tree is visited exactly once during the recursive traversal.
  + Operations per node (adding to list, checking level) are O(1).
  + Total time complexity is O(n), where n is the number of nodes in the tree.

**Space Complexity**

* **O(h) for recursion stack + O(n) for output**:
  + The recursion stack depends on the height h of the tree:
    - In a balanced tree, h = O(log n), so recursion stack is O(log n).
    - In a skewed tree, h = O(n), so recursion stack is O(n).
  + The output list res stores all node values, which is O(n) but typically not counted as auxiliary space.
  + Auxiliary space (excluding output) is O(h), where h is the tree height.
  + Total space complexity is O(h) for recursion, often simplified to O(log n) for balanced trees.

**Key Takeaways**

* **Recursive DFS for level-order**: Using DFS with a level parameter is an elegant alternative to the more common BFS approach, achieving the same result with different traversal mechanics.
* **Edge cases**:
  + Empty tree (root == null): Returns empty list (handled by base case).
  + Single node: Returns a list with one inner list containing the node’s value.
  + Skewed or balanced trees: Works correctly, as levels are tracked via the level parameter.
* **Alternative approaches**:
  + Iterative BFS: Use a queue to process nodes level by level (O(n) time, O(w) space, where w is the maximum width of the tree, typically O(n) in the worst case).
  + The BFS approach is more common but uses more space for wide trees.
* **Related problems**: This problem relates to "Binary Tree Zigzag Level Order Traversal" (modified level-order), "Maximum Depth of Binary Tree" (level tracking), and "Binary Tree Right Side View" (level-based processing) in the Blind 75 list.
* **Interview tip**: Highlight the recursive DFS approach’s simplicity and how it tracks levels. Be prepared to explain the BFS approach (using a queue) and discuss trade-offs, such as space complexity for wide vs. deep trees.

1. Validate Binary Search Tree (Trees)

## Problem Description

* Given the root of a binary tree, determine if it is a valid binary search tree (BST). A valid BST is defined as follows:
  + The left subtree of a node contains only nodes with values strictly less than the node's value.
  + The right subtree of a node contains only nodes with values strictly greater than the node's value.
  + Both the left and right subtrees must also be valid BSTs.

## Code with Comments

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

public class Solution {

// Main function to check if the tree is a valid BST

public boolean isValidBST(TreeNode root) {

// Call helper function with initial range of Long.MIN\_VALUE and Long.MAX\_VALUE

return valid(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

}

// Helper function to validate BST with range constraints

public boolean valid(TreeNode node, long left, long right) {

// Base case: if node is null, it is a valid BST

if (node == null) {

return true;

}

// Check if current node's value is within the valid range (left, right)

if (!(left < node.val && node.val < right)) {

return false;

}

// Recursively validate left subtree with updated upper bound (node.val)

// and right subtree with updated lower bound (node.val)

return valid(node.left, left, node.val) &&

valid(node.right, node.val, right);

}

}

**Approach**

* **Recursive range checking**: Use recursion to validate each node in the tree against a valid range of values.
* **Main function**: Start the validation by calling the helper function valid with the root node and the widest possible range (Long.MIN\_VALUE to Long.MAX\_VALUE).
* **Helper function (**valid**)**:
  + **Base case**: If the current node is null, return true (empty tree is a valid BST).
  + **Range check**: Verify that the current node’s value is strictly within the range (left, right). If not, return false.
  + **Recurse**:
    - For the left subtree, recurse with the same lower bound but update the upper bound to node.val (all left subtree values must be less than node.val).
    - For the right subtree, recurse with the same upper bound but update the lower bound to node.val (all right subtree values must be greater than node.val).
  + Combine results with && to ensure both subtrees are valid BSTs.
* **Use** long **for bounds**: Use long instead of int for range boundaries to handle edge cases where node values are Integer.MIN\_VALUE or Integer.MAX\_VALUE.

**Time Complexity**

* **O(n)**:
  + Each node in the tree is visited exactly once during the recursive traversal.
  + Operations per node (comparisons, recursive calls) are O(1).
  + Total time complexity is O(n), where n is the number of nodes in the tree.

**Space Complexity**

* **O(h)**:
  + The space complexity is determined by the recursion stack, which depends on the height h of the tree.
  + In a balanced tree, h = O(log n), so space is O(log n).
  + In a skewed tree, h = O(n), so space is O(n).
  + Overall, space complexity is O(h), where h is the height of the tree.

**Key Takeaways**

* **Range-based validation**: The solution uses dynamic range checking to ensure each node’s value satisfies BST properties, making it robust and efficient.
* **Edge cases**:
  + Empty tree: Returns true (handled by base case).
  + Single node: Returns true (valid BST if within initial range).
  + Invalid BST due to equal values: Handled by strict inequalities (left < node.val < right).
  + Extreme values (Integer.MIN\_VALUE, Integer.MAX\_VALUE): Using long for bounds prevents overflow issues.
* **Alternative approaches**:
  + Inorder traversal: Perform an inorder traversal and check if values are strictly increasing (O(n) time, O(h) space). Less intuitive for handling edge cases.
  + Iterative solution: Use a stack for DFS or queue for BFS with range checking (O(n) time, O(h) or O(w) space). More complex but avoids recursion.
* **Related problems**: This problem relates to "Lowest Common Ancestor of a Binary Search Tree" (BST properties), "Binary Search Tree Iterator" (inorder traversal), and "Search in a Binary Search Tree" (BST navigation) in the Blind 75 list.
* **Interview tip**: Highlight the use of long to handle edge cases and the recursive range-checking approach for clarity. Be ready to discuss the inorder traversal method and why strict inequalities are necessary for BST validation.

1. Kth Smallest Element In a Bst (Trees)

**Problem Description**

* Given the root of a binary search tree (BST) and an integer k, return the kth smallest element in the tree (1-indexed). Assume k is valid (1 ≤ k ≤ number of nodes in the tree).

**Code with Comments**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

// Helper function to perform inorder traversal and collect node values

public ArrayList<Integer> inOrder(TreeNode root, ArrayList<Integer> arr) {

// Base case: if root is null, return the current array

if (root == null) {

return arr;

}

// Recursively traverse the left subtree

inOrder(root.left, arr);

// Add current node's value to the array (inorder: left, root, right)

arr.add(root.val);

// Recursively traverse the right subtree

inOrder(root.right, arr);

// Return the updated array

return arr;

}

// Main function to find the kth smallest element in the BST

public int kthSmallest(TreeNode root, int k) {

// Perform inorder traversal to get all values in sorted order

ArrayList<Integer> nums = inOrder(root, new ArrayList<Integer>());

// Return the (k-1)th element (0-based index) from the sorted array

return nums.get(k - 1);

}

}

**Approach**

* **Leverage BST property**: In a BST, an inorder traversal (left, root, right) visits nodes in ascending order.
* **Inorder traversal**:
  + Use a recursive helper function inOrder to traverse the BST and collect node values in an ArrayList.
  + Base case: If the current node is null, return the current array.
  + Recursively traverse the left subtree, add the current node’s value, then traverse the right subtree.
* **Find kth element**: After obtaining the sorted list of node values, return the element at index k-1 (since k is 1-indexed).
* **Optimization**: The solution is straightforward but uses extra space to store all node values. An iterative or early-stopping approach could optimize space usage.

**Time Complexity**

* **O(n)**:
  + The inorder traversal visits each node exactly once to build the array, taking O(n) time, where n is the number of nodes.
  + Accessing the kth element in the array is O(1).
  + Total time complexity is O(n).

**Space Complexity**

* **O(n + h)**:
  + The ArrayList stores all n node values, contributing O(n) space.
  + The recursion stack for inorder traversal depends on the tree height h:
    - In a balanced BST, h = O(log n).
    - In a skewed BST, h = O(n).
  + Total space complexity is O(n + h), dominated by O(n) for the array.
  + Excluding the output array (if considered auxiliary space), the recursion stack gives O(h).

**Key Takeaways**

* **Inorder traversal for BST**: The solution leverages the BST property that inorder traversal yields sorted values, making it simple to find the kth smallest element.
* **Edge cases**:
  + Empty tree: Not applicable since k is guaranteed valid.
  + Single node (k = 1): Returns the node’s value (handled by inorder).
  + k equals number of nodes: Returns the largest value (last element in the array).
* **Alternative approaches**:
  + Iterative inorder with early stopping: Use a stack to perform inorder traversal and stop after finding the kth node (O(h + k) time, O(h) space), more space-efficient.
  + Recursive with counter: Track the count of visited nodes during inorder traversal and stop at the kth node (O(h + k) time, O(h) space).
* **Related problems**: This problem relates to "Validate Binary Search Tree" (BST properties), "Binary Search Tree Iterator" (inorder traversal), and "Lowest Common Ancestor of a BST" (BST navigation) in the Blind 75 list.
* **Interview tip**: Highlight the simplicity of using inorder traversal for BSTs and explain why the result is sorted. Be ready to discuss the space-efficient iterative approach and edge cases like k = 1 or maximum k.

1. Construct Binary Tree From Preorder And Inorder Traversal (Trees)
2. Binary Tree Maximum Path Sum (Trees)
3. Serialize And Deserialize Binary Tree (Trees)
4. Find Median From Data Stream (Heaps)
5. Combination Sum (Backtracking)
6. Word Search (Backtracking)
7. Implement Trie Prefix Tree (Tries)
8. Design Add And Search Words Data Structure (Tries)
9. Word Search II (Tries)
10. Number of Islands (Graphs)

## Problem Description

* Given an m x n 2D grid map of '1's (land) and '0's (water), return the number of islands. An island is a group of '1's connected vertically or horizontally, surrounded by '0's or the grid boundaries.

## Code with Comments

public class Solution {

// Define directions for exploring adjacent cells: down, up, right, left

private static final int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

// Function to count the number of islands in the grid

public int numIslands(char[][] grid) {

// Get grid dimensions

int ROWS = grid.length, COLS = grid[0].length;

// Initialize counter for number of islands

int islands = 0;

// Iterate through each cell in the grid

for (int r = 0; r < ROWS; r++) {

for (int c = 0; c < COLS; c++) {

// If a land cell ('1') is found, explore it with DFS

if (grid[r][c] == '1') {

// Perform DFS to mark all connected land cells as visited

dfs(grid, r, c);

// Increment island count after exploring the connected component

islands++;

}

}

}

// Return the total number of islands

return islands;

}

// Helper function to perform DFS and mark connected land cells

private void dfs(char[][] grid, int r, int c) {

// Base case: if out of bounds or cell is water/visited ('0'), return

if (r < 0 || c < 0 || r >= grid.length ||

c >= grid[0].length || grid[r][c] == '0') {

return;

}

// Mark current cell as visited by setting it to '0'

grid[r][c] = '0';

// Explore all four adjacent cells (down, up, right, left)

for (int[] dir : directions) {

dfs(grid, r + dir[0], c + dir[1]);

}

}

}

**Approach**

* **DFS with in-place marking**: Use depth-first search (DFS) to explore and count connected components of land cells ('1's) in the grid.
* **Initialize variables**:
  + Define directions for the four possible movements: down ({1, 0}), up ({-1, 0}), right ({0, 1}), left ({0, -1}).
  + Store grid dimensions (ROWS, COLS) for bounds checking.
  + Initialize islands counter to track the number of islands.
* **Iterate through grid**:
  + For each cell (r, c), if it’s a land cell (grid[r][c] == '1'), use DFS to mark all connected land cells as visited and increment the islands counter.
* **DFS helper**:
  + Base case: If the current cell is out of bounds or is water/visited ('0'), return.
  + Mark the current cell as visited by setting it to '0'.
  + Recursively explore all four adjacent cells using the directions array.
* **Return result**: The islands counter gives the total number of islands.
* **Optimization**: Marking cells in-place by changing '1' to '0' avoids the need for a separate visited array, satisfying the in-place requirement.

**Time Complexity**

* **O(m \* n)**:
  + Each cell in the m x n grid is visited at most once, either during the outer loop or via DFS.
  + DFS explores each connected land cell exactly once, and operations per cell (checking bounds, marking, recursion) are O(1).
  + Total time complexity is O(m \* n), where m is the number of rows and n is the number of columns.

**Space Complexity**

* **O(m \* n)**:
  + The recursion stack for DFS can go as deep as O(m \* n) in the worst case (e.g., a grid filled with '1's, forming a single island).
  + The directions array is O(1) (fixed size of 4).
  + No additional data structures are used, as marking is done in-place.
  + Total space complexity is O(m \* n) due to the recursion stack.

**Key Takeaways**

* **DFS for connected components**: The solution efficiently counts islands by using DFS to explore and mark connected land cells, treating each unvisited '1' as the start of a new island.
* **Edge cases**:
  + Empty grid: Returns 0 (handled by loop not running if ROWS = 0).
  + Single cell: Returns 1 if '1', 0 if '0' (handled by DFS and loop).
  + All water ('0's): Returns 0 (no islands found).
  + All land ('1's): Returns 1 (single island).
* **Alternative approaches**:
  + BFS: Use a queue to explore connected land cells (O(m \* n) time, O(min(m, n)) space for queue), equally effective but may use less stack space for wide grids.
  + Union-Find: Group connected '1's using a disjoint-set data structure (O(m \* n) time with path compression, O(m \* n) space), more complex but useful for related problems.
* **Related problems**: This problem relates to "Flood Fill" (DFS/BFS on grid), "Surrounded Regions" (grid traversal), and "Word Search" (grid DFS) in the Blind 75 list.
* **Interview tip**: Highlight the in-place marking strategy to save space and the DFS approach to explore connected components. Be ready to discuss BFS or Union-Find alternatives and edge cases like all-water or all-land grids.

1. Clone Graph (Graphs)

# Clone Graph

## Problem Description

* Given a reference to a node in a connected undirected graph, return a deep copy (clone) of the graph. Each node contains an integer value and a list of its neighbors. The graph may contain cycles, and all nodes must be cloned exactly once with the same structure.

## Code with Comments

/\*

Definition for a Node.

class Node {

public int val;

public List<Node> neighbors;

public Node() {

val = 0;

neighbors = new ArrayList<Node>();

}

public Node(int \_val) {

val = \_val;

neighbors = new ArrayList<Node>();

}

public Node(int \_val, ArrayList<Node> \_neighbors) {

val = \_val;

neighbors = \_neighbors;

}

}

\*/

class Solution {

// HashMap to store mapping of original nodes to their clones

private HashMap<Node, Node> visited = new HashMap<>();

// Function to clone the graph starting from the given node

public Node cloneGraph(Node node) {

// Base case: if node is null, return null

if (node == null) {

return null;

}

// If node has already been cloned, return its clone from the map

if (visited.containsKey(node)) {

return visited.get(node);

}

// Create a new node with the same value and an empty neighbor list

Node cloneNode = new Node(node.val, new ArrayList<>());

// Store the mapping of original node to its clone

visited.put(node, cloneNode);

// Recursively clone all neighbors and add them to the clone's neighbor list

for (Node neighbor : node.neighbors) {

cloneNode.neighbors.add(cloneGraph(neighbor));

}

// Return the cloned node

return cloneNode;

}

}

**Approach**

* **DFS with memoization**: Use depth-first search (DFS) to traverse the graph and create a deep copy, with a HashMap to track already cloned nodes to handle cycles.
* **Initialize HashMap**: Create a HashMap (visited) to map original nodes to their clones, preventing duplicate cloning in cyclic graphs.
* **Base cases**:
  + If the input node is null, return null.
  + If the node has already been cloned (visited.containsKey(node)), return its clone from the HashMap.
* **Clone node**:
  + Create a new Node with the same value as the input node and an empty neighbor list.
  + Store the mapping in visited.
* **Clone neighbors**: Recursively clone each neighbor using cloneGraph and add the cloned neighbors to the current node’s neighbor list.
* **Return result**: Return the cloned node.
* **Optimization**: The HashMap ensures each node is cloned exactly once, handling cycles efficiently and avoiding infinite recursion.

**Time Complexity**

* **O(V + E)**:
  + Each vertex (node) is visited and cloned exactly once, contributing O(V), where V is the number of nodes.
  + Each edge (neighbor connection) is processed once when adding cloned neighbors, contributing O(E), where E is the number of edges.
  + Operations like HashMap lookups and insertions are O(1) on average.
  + Total time complexity is O(V + E).

**Space Complexity**

* **O(V)**:
  + The HashMap stores a mapping for each node, using O(V) space.
  + The recursion stack can go as deep as O(V) in the worst case (e.g., a linear graph or deep DFS path).
  + The output (cloned graph) is not counted as auxiliary space.
  + Total space complexity is O(V) due to the HashMap and recursion stack.

**Key Takeaways**

* **Handling cycles**: The HashMap is critical for preventing duplicate cloning and infinite recursion in graphs with cycles.
* **Edge cases**:
  + Empty graph (node == null): Returns null.
  + Single node with no neighbors: Returns a single cloned node with empty neighbor list.
  + Cyclic graph: Correctly handled by checking visited before cloning.
  + Fully connected graph: All nodes and edges cloned exactly once.
* **Alternative approaches**:
  + BFS: Use a queue to clone nodes level by level, with a HashMap to track clones (O(V + E) time, O(V) space for queue and map), equally efficient but iterative.
  + Iterative DFS: Use a stack instead of recursion (O(V + E) time, O(V) space), avoids recursion stack but similar complexity.
* **Related problems**: This problem relates to "Number of Islands" (DFS on grid), "Course Schedule" (graph traversal), and "Copy List with Random Pointer" (deep copy with pointers) in the Blind 75 list.
* **Interview tip**: Highlight the role of the HashMap in handling cycles and ensuring each node is cloned once. Be ready to explain the BFS alternative and edge cases like a single node or cyclic graphs.

1. Pacific Atlantic Water Flow (Graphs)

**Problem Description**

* Given an m x n matrix heights where heights[r][c] represents the height of a cell at position (r, c), water can flow from a cell to another with height less than or equal to the current cell’s height, or to the Pacific Ocean (top or left edge) or Atlantic Ocean (bottom or right edge). Return a list of coordinates [r, c] where water can flow to both the Pacific and Atlantic Oceans.

**Code with Comments**

public class Solution {

// Define directions for exploring adjacent cells: down, up, right, left

private int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

// Function to find cells where water can flow to both Pacific and Atlantic Oceans

public List<List<Integer>> pacificAtlantic(int[][] heights) {

// Get matrix dimensions

int ROWS = heights.length, COLS = heights[0].length;

// Initialize boolean arrays to track cells reachable from Pacific and Atlantic

boolean[][] pac = new boolean[ROWS][COLS];

boolean[][] atl = new boolean[ROWS][COLS];

// Perform DFS from Pacific Ocean boundaries (top row and left column)

for (int c = 0; c < COLS; c++) {

// Start DFS from top row (Pacific)

dfs(0, c, pac, heights);

// Start DFS from bottom row (Atlantic)

dfs(ROWS - 1, c, atl, heights);

}

for (int r = 0; r < ROWS; r++) {

// Start DFS from left column (Pacific)

dfs(r, 0, pac, heights);

// Start DFS from right column (Atlantic)

dfs(r, COLS - 1, atl, heights);

}

// Collect cells that can reach both Pacific and Atlantic Oceans

List<List<Integer>> res = new ArrayList<>();

for (int r = 0; r < ROWS; r++) {

for (int c = 0; c < COLS; c++) {

// If cell is reachable from both oceans, add its coordinates to result

if (pac[r][c] && atl[r][c]) {

res.add(Arrays.asList(r, c));

}

}

}

// Return the list of coordinates

return res;

}

// Helper function to perform DFS from a given cell

private void dfs(int r, int c, boolean[][] ocean, int[][] heights) {

// Mark current cell as reachable

ocean[r][c] = true;

// Explore all four adjacent cells

for (int[] d : directions) {

int nr = r + d[0], nc = c + d[1];

// Check if the adjacent cell is within bounds, not visited, and has height >= current cell

if (nr >= 0 && nr < heights.length &&

nc >= 0 && nc < heights[0].length &&

!ocean[nr][nc] && heights[nr][nc] >= heights[r][c]) {

// Recursively explore the valid adjacent cell

dfs(nr, nc, ocean, heights);

}

}

}

}

**Approach**

* **Reverse flow with DFS**: Instead of simulating water flowing from cells to oceans, simulate water flowing from the Pacific and Atlantic Oceans to cells (reverse flow) to determine which cells can reach both oceans.
* **Initialize tracking arrays**:
  + Create pac and atl boolean arrays to mark cells reachable from the Pacific and Atlantic Oceans, respectively.
* **DFS from ocean boundaries**:
  + For Pacific: Start DFS from the top row (r = 0) and left column (c = 0).
  + For Atlantic: Start DFS from the bottom row (r = ROWS - 1) and right column (c = COLS - 1).
  + In DFS, explore adjacent cells where water can flow (i.e., heights[nr][nc] >= heights[r][c]), marking them as reachable in the respective ocean array.
* **Collect results**: Iterate through the grid and collect coordinates where both pac[r][c] and atl[r][c] are true, indicating cells reachable from both oceans.
* **Optimization**: Using reverse flow (from oceans to cells) simplifies the problem, as boundary cells are directly accessible, and DFS efficiently marks all reachable cells.

**Time Complexity**

* **O(m \* n)**:
  + DFS is performed from boundary cells: 2 \* (m + n) starting points (top/bottom rows, left/right columns).
  + Each cell is visited at most once per ocean (Pacific and Atlantic), as marked cells are skipped.
  + Total cells visited across both DFS traversals is O(m \* n), with O(1) operations per cell (bounds checking, marking).
  + Iterating to collect results takes O(m \* n).
  + Total time complexity is O(m \* n), where m is the number of rows and n is the number of columns.

**Space Complexity**

* **O(m \* n)**:
  + The pac and atl arrays each use O(m \* n) space to track reachable cells.
  + The recursion stack for DFS can go as deep as O(m \* n) in the worst case (e.g., all cells reachable).
  + The output list stores up to O(m \* n) coordinates but is not counted as auxiliary space.
  + The directions array is O(1).
  + Total space complexity is O(m \* n) due to the tracking arrays and recursion stack.

**Key Takeaways**

* **Reverse flow insight**: Simulating flow from oceans to cells (instead of cells to oceans) simplifies the problem by leveraging boundary conditions and DFS traversal.
* **Edge cases**:
  + Empty grid: Returns empty list (handled by ROWS = 0).
  + Single cell: Returns [[0, 0]] if reachable by both oceans (handled by DFS).
  + No cells reachable by both: Returns empty list.
  + All cells reachable: Returns all coordinates (e.g., uniform height grid).
* **Alternative approaches**:
  + BFS: Use queues instead of DFS to explore from boundaries (O(m \* n) time, O(min(m, n)) space for queues), equally effective but iterative.
  + Forward flow: Simulate flow from each cell to oceans (O(m \* n \* (m + n)) time, O(m \* n) space), less efficient due to repeated checks.
* **Related problems**: This problem relates to "Number of Islands" (DFS on grid), "Surrounded Regions" (grid traversal), and "Flood Fill" (DFS/BFS on grid) in the Blind 75 list.
* **Interview tip**: Highlight the reverse flow approach and why it’s efficient. Be ready to explain BFS as an alternative and discuss edge cases like single-cell or uniform-height grids.

1. Course Schedule (Graphs)

## Problem Description

* Given an integer numCourses representing the number of courses (labeled from 0 to numCourses - 1) and an array prerequisites where prerequisites[i] = [ai, bi] indicates that course bi must be taken before course ai, determine if it is possible to finish all courses. Return true if possible, false if there is a cycle in the prerequisite graph (indicating a conflict).

## Code with Comments

class Solution {

// Function to determine if all courses can be completed

public boolean canFinish(int numCourses, int[][] prerequisites) {

// Initialize adjacency list to represent the graph of course prerequisites

HashMap<Integer, List<Integer>> courseGraph = new HashMap<>();

// Build the graph: for each prerequisite pair [ai, bi], add ai to bi's adjacency list

for (int[] pre : prerequisites) {

// If bi already has an entry, add ai to its list of dependent courses

if (courseGraph.containsKey(pre[1])) {

courseGraph.get(pre[1]).add(pre[0]);

} else {

// Create a new list for bi, add ai, and put it in the graph

List<Integer> nextCourses = new LinkedList<>();

nextCourses.add(pre[0]);

courseGraph.put(pre[1], nextCourses);

}

}

// Initialize set to track visited nodes during DFS

HashSet<Integer> visited = new HashSet<>();

// Check each course to ensure no cycles exist

for (int currentCourse = 0; currentCourse < numCourses; currentCourse++) {

// If DFS detects a cycle starting from currentCourse, return false

if (courseSchedule(currentCourse, visited, courseGraph) == false) {

return false;

}

}

// If no cycles are found, return true

return true;

}

// Helper function to perform DFS and detect cycles

public boolean courseSchedule(int course, HashSet<Integer> visited,

HashMap<Integer, List<Integer>> courseGraph) {

// If course is already in the current DFS path, a cycle is detected

if (visited.contains(course)) {

return false;

}

// If course has no prerequisites or has been fully processed, no cycle

if (courseGraph.get(course) == null) {

return true;

}

// Mark current course as visited in the current DFS path

visited.add(course);

// Recursively check all dependent courses

for (int pre : courseGraph.get(course)) {

// If a cycle is detected in any dependent course, return false

if (courseSchedule(pre, visited, courseGraph) == false) {

return false;

}

}

// Remove course from visited set after exploring all its dependencies

visited.remove(course);

// Mark course as fully processed by setting its adjacency list to null

courseGraph.put(course, null);

// No cycle detected, return true

return true;

}

}

**Approach**

* **DFS with cycle detection**: Model the problem as a directed graph where nodes are courses and edges represent prerequisites, and use depth-first search (DFS) to detect cycles.
* **Build adjacency list**:
  + Create a HashMap (courseGraph) where each key is a course, and its value is a list of courses that depend on it (i.e., for [ai, bi], add ai to bi’s list).
* **DFS for cycle detection**:
  + For each course (0 to numCourses - 1), perform DFS to check for cycles.
  + In the courseSchedule helper:
    - If the current course is in the visited set, a cycle is detected (return false).
    - If the course has no prerequisites or has been fully processed (courseGraph.get(course) == null), return true.
    - Mark the course as visited, explore its prerequisites recursively, and remove it from visited after exploration.
    - Mark the course as fully processed by setting its adjacency list to null to avoid redundant checks.
* **Return result**: If no cycles are found after checking all courses, return true; otherwise, return false.
* **Optimization**: Setting processed courses’ adjacency lists to null and using a single visited set for cycle detection optimizes the DFS by avoiding re-exploration of completed courses.

**Time Complexity**

* **O(V + E)**:
  + Building the adjacency list takes O(E), where E is the number of prerequisites (edges).
  + DFS visits each vertex (course) and edge at most once:
    - Each course is processed once (O(V), where V = numCourses).
    - Each prerequisite edge is explored once during DFS (O(E)).
  + Operations like HashMap and HashSet lookups/insertions are O(1) on average.
  + Total time complexity is O(V + E), where V is the number of courses and E is the number of prerequisites.

**Space Complexity**

* **O(V + E)**:
  + The courseGraph HashMap stores O(V + E) space: O(V) for keys (courses) and O(E) for the adjacency lists.
  + The visited HashSet stores up to O(V) courses in the worst case (during a deep DFS path).
  + The recursion stack can go as deep as O(V) in the worst case (e.g., a linear dependency chain).
  + Total space complexity is O(V + E) due to the graph and recursion stack.

**Key Takeaways**

* **Cycle detection**: The solution uses DFS to detect cycles in a directed graph, which indicates whether all courses can be completed without conflicts.
* **Edge cases**:
  + No prerequisites: Returns true (no cycles possible).
  + Single course: Returns true (handled by loop and DFS base case).
  + Cyclic dependencies: Returns false (detected by visited set).
  + Disconnected graph: All courses checked via the outer loop.
* **Alternative approaches**:
  + BFS with topological sort: Use Kahn’s algorithm to check if a valid topological order exists (O(V + E) time, O(V) space), equally efficient but iterative.
  + DFS with two sets: Use separate sets for current path and globally visited nodes (O(V + E) time, O(V) space), slightly more verbose.
* **Related problems**: This problem relates to "Course Schedule II" (topological sort), "Clone Graph" (graph traversal), and "Pacific Atlantic Water Flow" (DFS on grid) in the Blind 75 list.
* **Interview tip**: Highlight the DFS cycle detection mechanism and the optimization of marking processed courses with null. Be ready to explain the BFS topological sort approach and edge cases like cycles or empty prerequisites.

1. Graph Valid Tree (Graphs)

## Problem Description

* Given n nodes labeled from 0 to n-1 and a list of undirected edges edges (where edges[i] = [ai, bi] represents an edge between nodes ai and bi), determine if the input forms a valid tree. A valid tree is a connected graph with no cycles and exactly n-1 edges.

## Code with Comments

class Solution {

// Function to check if the given nodes and edges form a valid tree

public boolean validTree(int n, int[][] edges) {

// Check if the number of edges is exactly n-1 (required for a tree)

if (edges.length != n - 1) {

return false;

}

// Initialize adjacency list for undirected graph

List<List<Integer>> adjacencyList = new ArrayList<>();

for (int i = 0; i < n; i++) {

// Create an empty list for each node

adjacencyList.add(new ArrayList<>());

}

// Build the adjacency list for undirected edges

for (int[] edge : edges) {

// Add bi to ai's list and ai to bi's list (undirected)

adjacencyList.get(edge[0]).add(edge[1]);

adjacencyList.get(edge[1]).add(edge[0]);

}

// Initialize stack for iterative DFS and set for visited nodes

Stack<Integer> stack = new Stack<>();

HashSet<Integer> visited = new HashSet<>();

// Start DFS from node 0

stack.push(0);

visited.add(0);

// Perform iterative DFS to explore all connected nodes

while (!stack.isEmpty()) {

// Pop the current node from the stack

int node = stack.pop();

// Explore all neighbors of the current node

for (int neighbour : adjacencyList.get(node)) {

// Skip if neighbor has already been visited

if (visited.contains(neighbour)) {

continue;

}

// Mark neighbor as visited and push to stack

visited.add(neighbour);

stack.push(neighbour);

}

}

// Check if all nodes were visited (graph is connected)

if (visited.size() == n) {

return true;

}

return false;

}

}

**Approach**

* **Edge count check**: A valid tree with n nodes must have exactly n-1 edges. If not, return false.
* **Build adjacency list**: Represent the undirected graph using an adjacency list where each node maps to a list of its neighbors.
* **Iterative DFS for connectivity**:
  + Use a stack to perform depth-first search (DFS) starting from node 0.
  + Track visited nodes with a HashSet to avoid revisiting and to implicitly check for cycles (no explicit cycle detection needed since n-1 edges ensure no cycles if connected).
  + For each node, explore unvisited neighbors, mark them as visited, and push them to the stack.
* **Check connectivity**: After DFS, verify that the number of visited nodes equals n, ensuring the graph is fully connected.
* **Optimization**: The edge count check eliminates the need for explicit cycle detection, as a connected graph with n-1 edges cannot have cycles. Iterative DFS avoids recursion stack space.

**Time Complexity**

* **O(V + E)**:
  + Building the adjacency list takes O(E), where E is the number of edges (equal to n-1 for a valid tree).
  + Iterative DFS visits each node and edge at most once:
    - Each node is processed once (O(V), where V = n).
    - Each edge is explored twice (once for each direction in an undirected graph), O(E).
  + Operations like HashSet lookups/insertions and stack operations are O(1) on average.
  + Total time complexity is O(V + E) = O(n + (n-1)) = O(n).
* Note: Since E = n-1 for a valid tree, the complexity simplifies to O(n).

**Space Complexity**

* **O(n)**:
  + The adjacency list stores O(n + E) = O(n + (n-1)) = O(n) space (each node has a list, and total edges are n-1).
  + The visited HashSet stores up to O(n) nodes.
  + The stack stores up to O(n) nodes in the worst case (e.g., a linear graph).
  + Total space complexity is O(n).

**Key Takeaways**

* **Tree properties**: A valid tree must have n-1 edges and be connected with no cycles. The edge count check simplifies cycle detection, as connectivity is sufficient given n-1 edges.
* **Edge cases**:
  + n = 1, no edges: Returns true (single node is a valid tree).
  + edges.length != n-1: Returns false (invalid edge count for a tree).
  + Disconnected graph: Returns false (detected by visited.size() != n).
  + Cyclic graph: Implicitly handled by edge count check (cannot have n-1 edges with a cycle).
* **Alternative approaches**:
  + Recursive DFS: Use recursive DFS instead of a stack (O(n) time, O(n) space due to recursion stack), similar but less explicit.
  + BFS: Use a queue for breadth-first search (O(n) time, O(n) space), equally effective.
  + Union-Find: Use a disjoint-set data structure to check for cycles and connectivity (O(n \* α(n)) time, O(n) space), more complex but generalizable.
* **Related problems**: This problem relates to "Course Schedule" (cycle detection), "Clone Graph" (graph traversal), and "Number of Islands" (DFS/BFS connectivity) in the Blind 75 list.
* **Interview tip**: Highlight the edge count check (n-1) and how it eliminates explicit cycle detection. Explain the iterative DFS and be ready to discuss BFS or Union-Find approaches and edge cases like a single node or disconnected graphs.

1. Number of Connected Components In An Undirected Graph (Graphs)

## Problem Description

* Given n nodes labeled from 0 to n-1 and a list of undirected edges edges (where edges[i] = [ai, bi] represents an edge between nodes ai and bi), return the number of connected components in the undirected graph. A connected component is a group of nodes connected by edges, with no connections to nodes outside the group.

## Code with Comments

class Solution {

// Function to count the number of connected components in the graph

public int countComponents(int n, int[][] edges) {

// Initialize counter for the number of connected components

int counter = 0;

// Initialize array to track visited nodes (0 = unvisited, 1 = visited)

int[] visited = new int[n];

// Initialize adjacency list for the undirected graph

List<Integer>[] adjList = new ArrayList[n];

for (int i = 0; i < n; i++) {

// Create an empty list for each node

adjList[i] = new ArrayList<Integer>();

}

// Build the adjacency list for undirected edges

for (int i = 0; i < edges.length; i++) {

// Add bi to ai's list and ai to bi's list (undirected)

adjList[edges[i][0]].add(edges[i][1]);

adjList[edges[i][1]].add(edges[i][0]);

}

// Iterate through each node to find unvisited components

for (int i = 0; i < n; i++) {

// If node is unvisited, it starts a new component

if (visited[i] == 0) {

// Increment component counter

counter++;

// Perform DFS to mark all nodes in the component as visited

dfs(adjList, visited, i);

}

}

// Return the number of connected components

return counter;

}

// Helper function to perform DFS and mark connected nodes as visited

public void dfs(List<Integer>[] adjList, int[] visited, int node) {

// Mark current node as visited

visited[node] = 1;

// Explore all neighbors of the current node

for (int i = 0; i < adjList[node].size(); i++) {

// If neighbor is unvisited, recursively explore it

if (visited[adjList[node].get(i)] == 0) {

dfs(adjList, visited, adjList[node].get(i));

}

}

}

}

**Approach**

* **DFS for connected components**: Use depth-first search (DFS) to explore and count connected components in the undirected graph.
* **Build adjacency list**:
  + Create an array of ArrayLists (adjList) where each node maps to a list of its neighbors.
  + For each edge [ai, bi], add bi to ai’s list and ai to bi’s list (undirected graph).
* **Track visited nodes**: Use an array visited to mark nodes as visited (1) or unvisited (0).
* **Count components**:
  + Iterate through all nodes (0 to n-1).
  + For each unvisited node, increment the component counter and perform DFS to mark all nodes in its connected component as visited.
* **DFS helper**:
  + Mark the current node as visited.
  + Recursively explore all unvisited neighbors.
* **Return result**: The counter gives the number of connected components.
* **Optimization**: The solution efficiently explores each node and edge once, and the adjacency list representation is suitable for sparse graphs.

**Time Complexity**

* **O(V + E)**:
  + Building the adjacency list takes O(E), where E is the number of edges.
  + DFS visits each node and edge at most once:
    - Each node is processed once (O(V), where V = n).
    - Each edge is explored twice (once for each direction in an undirected graph), O(E).
  + Operations like array access and list operations are O(1) on average.
  + Total time complexity is O(V + E), where V is the number of nodes and E is the number of edges.

**Space Complexity**

* **O(V + E)**:
  + The adjacency list stores O(V + E) space: O(V) for the array of lists and O(E) for the neighbor lists.
  + The visited array uses O(V) space.
  + The recursion stack for DFS can go as deep as O(V) in the worst case (e.g., a linear graph).
  + Total space complexity is O(V + E).

**Key Takeaways**

* **Connected components**: The solution uses DFS to identify and count connected components by exploring all nodes reachable from an unvisited starting node.
* **Edge cases**:
  + n = 1, no edges: Returns 1 (single node is one component).
  + No edges: Returns n (each node is its own component).
  + Single component: Returns 1 (all nodes connected).
  + Empty graph (n = 0): Returns 0 (handled implicitly by loop not running).
* **Alternative approaches**:
  + BFS: Use a queue instead of DFS to explore components (O(V + E) time, O(V) space for queue), equally efficient but iterative.
  + Union-Find: Use a disjoint-set data structure to group connected nodes (O(E \* α(V)) time with path compression, O(V) space), more complex but efficient for dynamic graphs.
* **Related problems**: This problem relates to "Number of Islands" (DFS for components), "Valid Tree" (connectivity and cycle check), and "Course Schedule" (graph traversal) in the Blind 75 list.
* **Interview tip**: Highlight the DFS approach for counting components and the role of the adjacency list. Be ready to discuss BFS or Union-Find alternatives and edge cases like isolated nodes or a single component.

1. Alien Dictionary (Advanced Graphs)
2. Climbing Stairs (1-D Dynamic Programming)
3. House Robber (1-D Dynamic Programming)
4. House Robber II (1-D Dynamic Programming)
5. Longest Palindromic Substring (1-D Dynamic Programming)
6. Palindromic Substrings (1-D Dynamic Programming)
7. Decode Ways (1-D Dynamic Programming)
8. Coin Change (1-D Dynamic Programming)
9. Maximum Product Subarray (1-D Dynamic Programming)
10. Word Break (1-D Dynamic Programming)
11. Longest Increasing Subsequence (1-D Dynamic Programming)
12. Unique Paths (2-D Dynamic Programming)
13. Longest Common Subsequence (2-D Dynamic Programming)
14. Maximum Subarray (Greedy)

## Problem Description

* Given an integer array nums, find the contiguous subarray with the largest sum and return its sum. The array contains at least one element.

## Code with Comments

class Solution {

// Function to find the maximum subarray sum

public int maxSubArray(int[] nums) {

// Initialize current sum to track the sum of the current subarray

int currSum = 0;

// Initialize maximum subarray sum with the first element

int maxSub = nums[0];

// Iterate through each number in the array

for (int num : nums) {

// If current sum is negative, reset it to 0 (start a new subarray)

if (currSum < 0) {

currSum = 0;

}

// Add the current number to the current subarray sum

currSum += num;

// Update the maximum subarray sum if current sum is larger

maxSub = Math.max(maxSub, currSum);

}

// Return the maximum subarray sum

return maxSub;

}

}

**Approach**

* **Kadane’s Algorithm**: Use a single pass through the array to find the maximum subarray sum.
* **Initialize variables**:
  + currSum: Tracks the sum of the current subarray ending at the current index.
  + maxSub: Tracks the maximum subarray sum seen so far, initialized with nums[0] to handle the case of a single-element array.
* **Iterate through array**:
  + For each number, check if currSum is negative. If so, reset currSum to 0, as starting a new subarray from the current number is better.
  + Add the current number to currSum.
  + Update maxSub with the maximum of maxSub and currSum.
* **Return result**: After the loop, maxSub contains the maximum subarray sum.
* **Optimization**: The algorithm processes the array in one pass and uses constant space, making it highly efficient.

**Time Complexity**

* **O(n)**:
  + The algorithm iterates through the array exactly once, where n is the length of nums.
  + Each iteration involves O(1) operations (comparisons, addition, Math.max).
  + Total time complexity is O(n).

**Space Complexity**

* **O(1)**:
  + The solution uses only two variables (currSum, maxSub), regardless of the input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Kadane’s Algorithm**: This is an optimal solution for finding the maximum subarray sum, efficiently handling both positive and negative numbers in a single pass.
* **Edge cases**:
  + Single element: Returns nums[0] (handled by initializing maxSub to nums[0]).
  + All negative numbers: Returns the largest single element (handled by currSum reset and maxSub updates).
  + All positive numbers: Likely includes all elements (handled naturally by the algorithm).
* **Alternative approaches**:
  + Brute force: Check all possible subarrays (O(n²) time, O(1) space), inefficient for large arrays.
  + Divide and conquer: Split the array and compute the maximum subarray sum across midpoints (O(n log n) time, O(log n) space), less efficient than Kadane’s.
* **Related problems**: This problem relates to "Best Time to Buy and Sell Stock" (tracking maximum difference), "Maximum Product Subarray" (similar dynamic programming), and "Longest Turbulent Subarray" (subarray properties) in the Blind 75 list.
* **Interview tip**: Highlight the elegance of Kadane’s Algorithm and its O(n) time and O(1) space efficiency. Be ready to explain why resetting currSum to 0 works and discuss edge cases like all negative numbers.

1. Jump Game (Greedy)

**Problem Description**

* Given an array of non-negative integers nums, where each element represents the maximum jump length from that index, determine if you can reach the last index starting from the first index.

**Code with Comments**

class Solution {

// Function to determine if the last index can be reached

public boolean canJump(int[] nums) {

// Initialize goal as the last index of the array

int goal = nums.length - 1;

// Iterate backwards from the second-to-last index to the start

for (int i = nums.length - 2; i >= 0; i--) {

// Check if the current index can reach or surpass the current goal

if (i + nums[i] >= goal) {

// If reachable, update goal to the current index

goal = i;

}

}

// Return true if the goal has been moved to index 0, false otherwise

return goal == 0;

}

}

**Approach**

* **Greedy backward approach**: Start from the end of the array and work backwards to determine if the first index can reach the last index.
* **Initialize goal**: Set goal to the last index (nums.length - 1), representing the target to reach.
* **Iterate backwards**: From the second-to-last index to the first (index 0):
  + Check if the current index i can reach or surpass the current goal by jumping nums[i] steps (i + nums[i] >= goal).
  + If it can, update goal to the current index i, as reaching i is now sufficient to reach the end.
* **Check result**: After the loop, return true if goal == 0 (indicating the first index can reach the last index), otherwise return false.
* **Optimization**: The greedy approach ensures a single pass through the array, making it efficient by updating the earliest index that can reach the end.

**Time Complexity**

* **O(n)**:
  + The algorithm iterates through the array once, from index nums.length - 2 to 0, which is O(n) where n is the length of nums.
  + Each iteration involves O(1) operations (comparison, addition, assignment).
  + Total time complexity is O(n).

**Space Complexity**

* **O(1)**:
  + The solution uses only a single variable (goal), regardless of the input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Greedy efficiency**: The backward greedy approach efficiently determines reachability by tracking the earliest index that can reach the last index, avoiding unnecessary exploration.
* **Edge cases**:
  + Single element: Returns true (no jumps needed, handled implicitly as loop doesn’t run).
  + All zeros except last: Returns false unless starting at the last index (handled by goal not moving to 0).
  + Large jumps: Correctly handles cases where jumps can skip multiple indices.
* **Alternative approaches**:
  + Forward greedy: Track the maximum reachable index starting from index 0 (O(n) time, O(1) space), equally efficient but processes forward.
  + Dynamic programming: Use a boolean array to track reachable indices (O(n) time, O(n) space), less space-efficient.
* **Related problems**: This problem relates to "Jump Game II" (minimum jumps to reach end), "Minimum Number of Taps to Open to Water a Garden" (greedy reachability), and "Maximum Subarray" (greedy optimization) in the Blind 75 list.
* **Interview tip**: Highlight the greedy nature of the solution and why moving the goal backwards works. Be ready to explain the forward greedy approach and discuss edge cases like arrays with all zeros or single elements.

1. Insert Interval (Intervals)

## Problem Description

* Given a list of non-overlapping intervals intervals sorted in ascending order by start time, and a new interval newInterval, insert newInterval into intervals such that the resulting list is still sorted and contains no overlapping intervals. Merge intervals if necessary.

## Code with Comments

public class Solution {

// Function to insert a new interval into a sorted list of non-overlapping intervals

public int[][] insert(int[][] intervals, int[] newInterval) {

// Initialize a list to store the result intervals

List<int[]> res = new ArrayList<>();

// Iterate through each interval in the input array

for (int[] interval : intervals) {

// Case 1: Current interval ends before newInterval starts, add it to result

if (newInterval == null || interval[1] < newInterval[0]) {

res.add(interval);

}

// Case 2: Current interval starts after newInterval ends, add newInterval then current interval

else if (interval[0] > newInterval[1]) {

res.add(newInterval);

res.add(interval);

newInterval = null; // Mark newInterval as processed

}

// Case 3: Overlap exists, merge by updating newInterval's bounds

else {

newInterval[0] = Math.min(interval[0], newInterval[0]); // Update start to minimum

newInterval[1] = Math.max(interval[1], newInterval[1]); // Update end to maximum

}

}

// If newInterval hasn't been added, append it to the result

if (newInterval != null) res.add(newInterval);

// Convert the result list to an array and return

return res.toArray(new int[res.size()][]);

}

}

**Approach**

* **Single-pass interval merging**: Iterate through the sorted intervals and handle newInterval by comparing it with each interval to determine whether to add, merge, or skip.
* **Initialize result**: Create an ArrayList to store the resulting intervals, allowing dynamic resizing.
* **Process intervals**:
  + **Case 1: No overlap, interval before** newInterval: If the current interval ends before newInterval starts (interval[1] < newInterval[0]), add the current interval to the result.
  + **Case 2: No overlap, interval after** newInterval: If the current interval starts after newInterval ends (interval[0] > newInterval[1]), add newInterval to the result, then add the current interval, and mark newInterval as processed (null).
  + **Case 3: Overlap**: Merge the current interval with newInterval by updating newInterval’s start to the minimum of both starts and end to the maximum of both ends.
* **Handle remaining** newInterval: If newInterval hasn’t been added (i.e., it’s not null), append it to the result (e.g., when it belongs at the end or the input list is empty).
* **Return result**: Convert the ArrayList to a 2D array and return.
* **Optimization**: The solution processes the intervals in one pass, merging as needed, and uses newInterval = null to avoid adding it multiple times.

**Time Complexity**

* **O(n)**:
  + The algorithm iterates through the intervals array once, where n is the number of intervals.
  + Each iteration involves O(1) operations (comparisons, Math.min, Math.max, list additions).
  + Converting the ArrayList to an array is O(n) in the worst case.
  + Total time complexity is O(n).

**Space Complexity**

* **O(n)**:
  + The ArrayList res stores up to n + 1 intervals (including newInterval), contributing O(n) space.
  + Excluding the output space (as it’s required for the result), auxiliary space is O(1) since only a few variables are used.
  + Total space complexity is O(n) for the output.

**Key Takeaways**

* **Interval processing**: The solution efficiently handles interval insertion by categorizing each interval relative to newInterval (before, after, or overlapping) and merging as needed.
* **Edge cases**:
  + Empty intervals array: Returns [newInterval] (handled by final check).
  + newInterval before/after all intervals: Added correctly in Case 1 or final check.
  + Full overlap: Correctly merges multiple overlapping intervals into newInterval.
  + Single interval: Handled correctly by comparison and merging logic.
* **Alternative approaches**:
  + Two-pass solution: First add non-overlapping intervals before newInterval, then merge overlapping intervals, then add remaining intervals (O(n) time, O(n) space), less concise.
  + Sort-based: Add newInterval to intervals, sort by start time, then merge all intervals (O(n log n) time, O(n) space), less efficient.
* **Related problems**: This problem relates to "Merge Intervals" (interval merging), "Meeting Rooms II" (interval scheduling), and "Non-overlapping Intervals" (interval manipulation) in the Blind 75 list.
* **Interview tip**: Explain the three cases (before, after, overlap) and how newInterval = null prevents duplicate additions. Be ready to discuss the alternative sort-based approach and edge cases like an empty input array.

1. Merge Intervals (Intervals)

## Problem Description

* Given an array of intervals intervals where intervals[i] = [start\_i, end\_i], merge all overlapping intervals and return an array of non-overlapping intervals that cover all the intervals in the input. The intervals are not guaranteed to be sorted.

## Code with Comments

class Solution {

// Function to merge overlapping intervals

public int[][] merge(int[][] intervals) {

// Sort intervals by start time to ensure sequential processing

Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

// Initialize a list to store merged intervals

List<int[]> ans = new ArrayList<>();

// Iterate through each interval in the sorted array

for (int[] interval : intervals) {

// If the result list is empty or the last merged interval doesn't overlap with the current one

if (ans.isEmpty() || ans.get(ans.size() - 1)[1] < interval[0]) {

// Add the current interval to the result list

ans.add(interval);

} else {

// Merge by updating the end time of the last interval to the maximum end time

ans.get(ans.size() - 1)[1] = Math.max(ans.get(ans.size() - 1)[1], interval[1]);

}

}

// Convert the result list to a 2D array and return

return ans.toArray(new int[ans.size()][]);

}

}

**Approach**

* **Sort intervals**: Sort the input intervals array by start time to ensure overlapping intervals are adjacent.
* **Initialize result**: Create an ArrayList to store the merged intervals, allowing dynamic resizing.
* **Process intervals**:
  + For each interval, check if the result list is empty or if the current interval does not overlap with the last merged interval (ans.get(ans.size() - 1)[1] < interval[0]).
    - If true, add the current interval to the result list as a new interval.
    - If false (overlap exists), merge by updating the end time of the last interval in the result list to the maximum of its current end time and the current interval’s end time.
* **Return result**: Convert the ArrayList to a 2D array and return.
* **Optimization**: Sorting ensures a single pass through the intervals is sufficient, and merging is done in-place within the result list.

**Time Complexity**

* **O(n log n)**:
  + Sorting the intervals takes O(n log n), where n is the number of intervals.
  + The single pass through the sorted intervals takes O(n), with O(1) operations per interval (comparisons, Math.max, list additions).
  + Converting the ArrayList to an array is O(n).
  + Total time complexity is dominated by sorting, so O(n log n).

**Space Complexity**

* **O(n)**:
  + The ArrayList ans stores up to n intervals in the worst case (no merging), contributing O(n) space.
  + The sorting operation may require O(log n) space for the recursion stack in some implementations (e.g., Timsort in Java), but this is often considered O(1) auxiliary space in practice.
  + Excluding the output, auxiliary space is O(1) for the iteration variables.
  + Total space complexity is O(n) for the output.

**Key Takeaways**

* **Sorting for efficiency**: Sorting by start time simplifies the merging process by ensuring overlapping intervals are consecutive, allowing a single-pass solution.
* **Edge cases**:
  + Empty array: Returns an empty array (handled by empty ans).
  + Single interval: Returns the interval itself (no merging needed).
  + No overlaps: Returns all intervals as is (handled by adding non-overlapping intervals).
  + All overlapping: Merges into a single interval (handled by updating end times).
* **Alternative approaches**:
  + Brute force: Check each interval against all others for overlaps (O(n²) time, O(n) space), highly inefficient.
  + Sweep line: Use a sweep line algorithm to process start and end points (O(n log n) time, O(n) space), more complex but viable for related problems.
* **Related problems**: This problem relates to "Insert Interval" (interval insertion and merging), "Meeting Rooms II" (interval scheduling), and "Non-overlapping Intervals" (interval manipulation) in the Blind 75 list.
* **Interview tip**: Highlight the importance of sorting to achieve O(n log n) time and the simplicity of the single-pass merging logic. Be ready to discuss edge cases (e.g., empty input) and alternative approaches like sweep line.

1. Non Overlapping Intervals (Intervals)

## Problem Description

* Given an array of intervals intervals where intervals[i] = [start\_i, end\_i], return the minimum number of intervals that must be removed to make the remaining intervals non-overlapping. The intervals are not guaranteed to be sorted.

## Code with Comments

class Solution {

// Function to find the minimum number of intervals to remove to make them non-overlapping

public int eraseOverlapIntervals(int[][] intervals) {

// Handle edge case: if the array is empty, no intervals to remove

if (intervals.length == 0) {

return 0;

}

// Sort intervals by start time to process them sequentially

Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

// Initialize index of previous non-overlapping interval

int prev = 0;

// Initialize counter for intervals to remove

int count = 0;

// Iterate through intervals starting from the second one

for (int i = 1; i < intervals.length; i++) {

// Check if current interval overlaps with the previous selected interval

if (intervals[prev][1] > intervals[i][0]) {

// If overlap exists, choose the interval with the earlier end time

if (intervals[prev][1] > intervals[i][1]) {

// Update prev to current interval (it ends earlier)

prev = i;

}

// Increment count of intervals to remove

count++;

} else {

// No overlap, update prev to current interval

prev = i;

}

}

// Return the minimum number of intervals to remove

return count;

}

}

**Approach**

* **Greedy by earliest end time**: Sort intervals by start time and greedily select intervals with the earliest end time to maximize the number of non-overlapping intervals.
* **Handle edge case**: If the input array is empty, return 0 (no intervals to remove).
* **Sort intervals**: Sort intervals by start time to ensure sequential processing.
* **Initialize variables**:
  + prev: Index of the last selected non-overlapping interval, starting at 0.
  + count: Tracks the number of intervals to remove.
* **Iterate through intervals**:
  + For each interval starting from index 1:
    - If it overlaps with the previous selected interval (intervals[prev][1] > intervals[i][0]):
      * Increment count to mark an interval for removal.
      * Choose the interval with the earlier end time by updating prev to i if intervals[i][1] < intervals[prev][1].
    - If no overlap, update prev to the current interval i.
* **Return result**: The count represents the minimum number of intervals to remove.
* **Optimization**: Sorting allows a single pass, and choosing the earlier end time ensures the maximum number of non-overlapping intervals are kept.

**Time Complexity**

* **O(n log n)**:
  + Sorting the intervals takes O(n log n), where n is the number of intervals.
  + The single pass through the sorted intervals takes O(n), with O(1) operations per interval (comparisons, assignments).
  + Total time complexity is dominated by sorting, so O(n log n).

**Space Complexity**

* **O(1)**:
  + The solution uses only a few variables (prev, count), and sorting is typically done in-place (or uses O(log n) recursion stack space for Timsort in Java, often considered O(1) auxiliary space).
  + No additional data structures are used, making the auxiliary space complexity O(1).
  + Note: The output (number of intervals to remove) is a single integer, not counted in auxiliary space.

**Key Takeaways**

* **Greedy strategy**: Sorting by start time and choosing intervals with the earliest end time minimizes the number of intervals to remove, similar to the activity selection problem.
* **Edge cases**:
  + Empty array: Returns 0 (handled by base case).
  + Single interval: Returns 0 (no overlaps possible, handled implicitly).
  + No overlaps: Returns 0 (all intervals kept).
  + All overlapping: Removes all but the interval with the earliest end time.
* **Alternative approaches**:
  + Sort by end time: Sort intervals by end time and greedily select non-overlapping intervals (O(n log n) time, O(1) space), equally efficient but processes end times first.
  + Dynamic programming: Compute the maximum number of non-overlapping intervals using DP (O(n²) time, O(n) space), less efficient.
* **Related problems**: This problem relates to "Merge Intervals" (interval merging), "Insert Interval" (interval insertion), and "Meeting Rooms II" (interval scheduling) in the Blind 75 list.
* **Interview tip**: Explain the greedy choice of keeping the interval with the earlier end time to maximize non-overlapping intervals. Be ready to discuss sorting by end time and edge cases like empty or non-overlapping inputs.

1. Meeting Rooms (Intervals)

## Problem Description

* Given a list of meeting time intervals intervals where each interval has a start and end time (start, end), determine if a person can attend all meetings without any conflicts. A conflict occurs if any two meetings overlap (i.e., one meeting’s end time is greater than the next meeting’s start time).

## Code with Comments

/\*\*

\* Definition of Interval:

\* public class Interval {

\* public int start, end;

\* public Interval(int start, int end) {

\* this.start = start;

\* this.end = end;

\* }

\* }

\*/

class Solution {

// Function to check if all meetings can be attended without conflicts

public boolean canAttendMeetings(List<Interval> intervals) {

// Sort intervals by start time to check for overlaps sequentially

intervals.sort((a, b) -> Integer.compare(a.start, b.start));

// Iterate through the sorted intervals up to the second-to-last one

for (int i = 0; i < intervals.size() - 1; i++) {

// Check if the current meeting's end time overlaps with the next meeting's start time

if (intervals.get(i).end > intervals.get(i + 1).start) {

// If overlap exists, return false as meetings conflict

return false;

}

}

// If no overlaps are found, return true as all meetings can be attended

return true;

}

}

**Approach**

* **Sort by start time**: Sort the list of intervals by their start times to ensure adjacent intervals are checked for overlaps.
* **Check for overlaps**: Iterate through the sorted intervals and compare each interval’s end time with the next interval’s start time.
  + If the current interval’s end time (intervals[i].end) is greater than the next interval’s start time (intervals[i + 1].start), there is an overlap, so return false.
* **No conflicts**: If the loop completes without finding any overlaps, return true, indicating all meetings can be attended.
* **Optimization**: Sorting ensures a single pass through the intervals is sufficient to detect conflicts, leveraging the fact that overlaps can only occur between adjacent intervals in a sorted list.

**Time Complexity**

* **O(n log n)**:
  + Sorting the intervals takes O(n log n), where n is the number of intervals.
  + The single pass through the sorted intervals takes O(n), with O(1) operations per iteration (comparison).
  + Total time complexity is dominated by sorting, so O(n log n).

**Space Complexity**

* **O(1)**:
  + The solution performs in-place sorting (or uses O(log n) space for the sorting algorithm’s recursion stack in Java’s Timsort, often considered O(1) auxiliary space).
  + No additional data structures are used beyond the loop counter, making auxiliary space O(1).
  + Note: The input list is modified in-place, and the output is a boolean, not counted in auxiliary space.

**Key Takeaways**

* **Sorting for conflict detection**: Sorting by start time simplifies overlap detection to a single pass, as conflicts only occur between consecutive intervals in a sorted list.
* **Edge cases**:
  + Empty list: Returns true (no meetings, no conflicts, handled by loop not running).
  + Single meeting: Returns true (no overlaps possible, handled by loop not running).
  + Non-overlapping intervals: Returns true (handled by loop finding no overlaps).
  + Overlapping intervals: Returns false (detected when end > start of next interval).
* **Alternative approaches**:
  + Sweep line: Process start and end times as events in a sorted order (O(n log n) time, O(n) space), more complex but useful for related problems.
  + Brute force: Check every pair of intervals for overlap (O(n²) time, O(1) space), inefficient for large inputs.
* **Related problems**: This problem relates to "Merge Intervals" (interval merging), "Non-overlapping Intervals" (interval removal), and "Meeting Rooms II" (scheduling with overlaps) in the Blind 75 list.
* **Interview tip**: Highlight the efficiency of sorting followed by a single pass to detect conflicts. Be ready to discuss the sweep line approach and edge cases like empty or single-interval inputs.

1. Meeting Rooms II (Intervals)

## Problem Description

* Given a list of meeting time intervals intervals where each interval has a start and end time (start, end), return the minimum number of meeting rooms required to accommodate all meetings without conflicts. Intervals may overlap, and each meeting requires a separate room if it overlaps with others.

## Code with Comments

/\*\*

\* Definition of Interval:

\* public class Interval {

\* public int start, end;

\* public Interval(int start, int end) {

\* this.start = start;

\* this.end = end;

\* }

\* }

\*/

public class Solution {

// Function to find the minimum number of meeting rooms required

public int minMeetingRooms(List<Interval> intervals) {

// Get the number of intervals

int n = intervals.size();

// Initialize arrays to store start and end times

int[] start = new int[n];

int[] end = new int[n];

// Populate start and end arrays from intervals

for (int i = 0; i < n; i++) {

// Store the start time of the i-th interval

start[i] = intervals.get(i).start;

// Store the end time of the i-th interval

end[i] = intervals.get(i).end;

}

// Sort start times in ascending order

Arrays.sort(start);

// Sort end times in ascending order

Arrays.sort(end);

// Initialize result (max rooms needed) and current room count

int res = 0, count = 0, s = 0, e = 0;

// Process events until all start times are handled

while (s < n) {

// If next event is a meeting start (start time < next end time)

if (start[s] < end[e]) {

// Increment start pointer to process next start time

s++;

// Increment room count as a new meeting starts

count++;

} else {

// Increment end pointer to process next end time

e++;

// Decrement room count as a meeting ends

count--;

}

// Update maximum rooms needed so far

res = Math.max(res, count);

}

// Return the minimum number of rooms required

return res;

}

}

**Approach**

* **Sweep line with two-pointer technique**: Process meeting start and end times as events to track the maximum number of concurrent meetings.
* **Extract start and end times**: Create two arrays, start and end, to store the start and end times of all intervals.
* **Sort arrays**: Sort start and end arrays separately in ascending order to process events chronologically.
* **Process events**:
  + Use two pointers, s for start times and e for end times.
  + While s < n (not all start times processed):
    - If the next start time (start[s]) is less than the next end time (end[e]), a meeting starts, so increment count (rooms needed) and move s.
    - Otherwise, a meeting ends, so decrement count and move e.
    - Update res with the maximum count seen so far, representing the peak number of concurrent meetings.
* **Return result**: res gives the minimum number of rooms required.
* **Optimization**: Sorting start and end times separately allows efficient tracking of concurrent meetings using a single pass through the sorted arrays.

**Time Complexity**

* **O(n log n)**:
  + Extracting start and end times takes O(n), where n is the number of intervals.
  + Sorting the start and end arrays takes O(n log n).
  + The two-pointer traversal processes each start and end time exactly once, taking O(n).
  + Total time complexity is dominated by sorting, so O(n log n).

**Space Complexity**

* **O(n)**:
  + The start and end arrays each require O(n) space.
  + Additional variables (res, count, s, e) use O(1) space.
  + Total space complexity is O(n) for the arrays.
  + Note: If sorting is considered to use O(log n) space for the recursion stack (e.g., Timsort), it’s typically not dominant.

**Key Takeaways**

* **Sweep line efficiency**: Separating and sorting start and end times allows tracking of concurrent meetings in a single pass, making the solution both intuitive and efficient.
* **Edge cases**:
  + Empty list: Returns 0 (no meetings, no rooms needed, handled by n = 0).
  + Single meeting: Returns 1 (one room needed, handled by processing one start/end pair).
  + No overlaps: Returns 1 (each meeting ends before the next starts).
  + All concurrent meetings: Returns n (maximum rooms needed).
* **Alternative approaches**:
  + Priority queue (min-heap): Store end times of ongoing meetings, adding new meetings and removing ended ones (O(n log n) time, O(n) space), more complex but intuitive for scheduling.
  + Sort by start time: Process intervals and track overlapping meetings (O(n log n) time, O(1) space if modifying input), less generalizable.
* **Related problems**: This problem relates to "Meeting Rooms" (conflict detection), "Merge Intervals" (interval merging), and "Non-overlapping Intervals" (interval removal) in the Blind 75 list.
* **Interview tip**: Highlight the sweep line approach and how sorting start/end times tracks concurrent meetings. Be ready to discuss the priority queue approach and edge cases like empty or single-meeting inputs.

1. Rotate Image (Math & Geometry)

## Problem Description

* Given an n x n 2D matrix representing an image, rotate the image by 90 degrees clockwise in-place. You cannot use extra space for another matrix; you must modify the input matrix directly.

## Code with Comments

class Solution {

// Function to rotate an n x n matrix by 90 degrees clockwise in-place

public void rotate(int[][] matrix) {

// Get the size of the matrix (n x n)

int n = matrix.length;

// Iterate over the layers of the matrix (up to (n+1)/2 for rows)

for (int i = 0; i < (n + 1) / 2; i++) {

// Iterate over columns up to n/2 for each layer

for (int j = 0; j < n / 2; j++) {

// Store the top-left element of the current group in a temporary variable

int temp = matrix[n - 1 - j][i];

// Move bottom-left to top-left

matrix[n - 1 - j][i] = matrix[n - 1 - i][n - j - 1];

// Move bottom-right to bottom-left

matrix[n - 1 - i][n - j - 1] = matrix[j][n - 1 - i];

// Move top-right to bottom-right

matrix[j][n - 1 - i] = matrix[i][j];

// Move top-left (stored in temp) to top-right

matrix[i][j] = temp;

}

}

}

}

**Approach**

* **In-place 90-degree rotation**: Rotate the matrix by swapping elements in a specific pattern to achieve a 90-degree clockwise rotation without using extra space.
* **Layer-by-layer processing**:
  + Treat the matrix as a set of concentric layers (like an onion).
  + Iterate over the outer layers to the inner layers, covering rows i from 0 to (n + 1) / 2 and columns j from 0 to n / 2.
  + For each position (i, j) in a layer, perform a four-way swap to rotate a group of four elements:
    - Top-left: (i, j)
    - Top-right: (j, n - 1 - i)
    - Bottom-right: (n - 1 - i, n - 1 - j)
    - Bottom-left: (n - 1 - j, i)
* **Four-way swap**:
  + Store the top-left element in temp.
  + Move bottom-left to top-left.
  + Move bottom-right to bottom-left.
  + Move top-right to bottom-right.
  + Move top-left (from temp) to top-right.
* **Optimization**: The in-place swapping ensures no extra matrix is used, and the loop bounds (n + 1) / 2 and n / 2 handle both odd and even n, ensuring all elements are rotated correctly.

**Time Complexity**

* **O(n²)**:
  + The matrix is n x n, and the nested loops iterate over approximately n/2 \* n/2 = n²/4 elements (one quadrant of the matrix).
  + Each iteration performs a constant number of swaps (O(1)).
  + Total time complexity is O(n²), as all elements in the matrix are processed.
* Note: This is optimal, as every element must be moved to achieve the rotation.

**Space Complexity**

* **O(1)**:
  + The solution modifies the matrix in-place, using only a single temporary variable (temp) for swapping.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **In-place rotation**: The four-way swap for each group of elements ensures the matrix is rotated in-place, meeting the problem’s space constraint.
* **Edge cases**:
  + n = 1: No rotation needed (single element, loop doesn’t run).
  + n = 2: Rotates four elements in one iteration.
  + Odd n (e.g., n = 3): Middle element stays in place, handled by (n + 1) / 2 loop bound.
  + Even n (e.g., n = 4): All elements swapped in layers.
* **Alternative approaches**:
  + Transpose and reverse: First transpose the matrix (swap across diagonal, O(n²) time), then reverse each row (O(n²) time, O(1) space). Equally efficient but involves two distinct steps.
  + Extra matrix: Copy to a new matrix with rotated indices (O(n²) time, O(n²) space), not allowed by the problem’s constraints.
* **Related problems**: This problem relates to "Spiral Matrix" (matrix traversal), "Set Matrix Zeroes" (in-place matrix manipulation), and "Rotate Array" (rotation logic) in the Blind 75 list.
* **Interview tip**: Explain the four-way swap process and how the loop bounds cover all necessary elements. Be ready to discuss the transpose-and-reverse approach and edge cases like n = 1 or odd n.

1. Spiral Matrix (Math & Geometry)

## Problem Description

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

## Code with Comments

public class Solution {

// Function to return matrix elements in spiral order

public List<Integer> spiralOrder(int[][] matrix) {

// Initialize result list to store spiral order elements

List<Integer> res = new ArrayList<>();

// Define directions for traversal: right, down, left, up

int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

// Initialize steps for horizontal and vertical movements

int[] steps = {matrix[0].length, matrix.length - 1};

// Initialize starting position (r, c) and direction index d

int r = 0, c = -1, d = 0;

// Continue while there are steps to take in the current direction

while (steps[d % 2] > 0) {

// Take steps[d % 2] steps in the current direction

for (int i = 0; i < steps[d % 2]; i++) {

// Update row and column based on current direction

r += directions[d][0];

c += directions[d][1];

// Add current element to result

res.add(matrix[r][c]);

}

// Decrease steps for the current direction (horizontal or vertical)

steps[d % 2]--;

// Move to the next direction (right -> down -> left -> up)

d = (d + 1) % 4;

}

// Return the list of elements in spiral order

return res;

}

}

**Approach**

* **Spiral traversal with directions**: Traverse the matrix in a spiral (clockwise) order by moving right, down, left, and up, adjusting boundaries after each direction.
* **Initialize variables**:
  + res: An ArrayList to store the spiral order of elements.
  + directions: A 2D array defining movements: right ({0, 1}), down ({1, 0}), left ({0, -1}), up ({-1, 0}).
  + steps: An array tracking remaining steps for horizontal (width: matrix[0].length) and vertical (height - 1: matrix.length - 1) movements.
  + r, c: Current row and column, starting at (0, -1) to simplify the first rightward move.
  + d: Direction index (0: right, 1: down, 2: left, 3: up).
* **Traverse in spiral**:
  + While there are steps to take in the current direction (steps[d % 2] > 0):
    - Move steps[d % 2] times in the current direction, updating r and c and adding elements to res.
    - Decrease steps[d % 2] (horizontal for d = 0, 2, vertical for d = 1, 3).
    - Switch to the next direction by incrementing d modulo 4.
* **Return result**: Return the res list containing all elements in spiral order.
* **Optimization**: Using steps and d % 2 alternates between horizontal and vertical steps efficiently, avoiding explicit boundary tracking.

**Time Complexity**

* **O(m \* n)**:
  + Each element in the m x n matrix is visited exactly once and added to the result.
  + The loop iterates over all elements, with O(1) operations per element (updates to r, c, and list addition).
  + Total time complexity is O(m \* n), where m is the number of rows and n is the number of columns.

**Space Complexity**

* **O(m \* n)**:
  + The output list res stores all m \* n elements, contributing O(m \* n) space.
  + Auxiliary space includes directions (O(1), fixed size 4), steps (O(1), size 2), and variables (r, c, d, i), all O(1).
  + Excluding the output, auxiliary space is O(1).
  + Total space complexity is O(m \* n) due to the output list.

**Key Takeaways**

* **Spiral traversal**: The solution uses a direction-based approach with dynamic step counts to traverse the matrix in a clockwise spiral, making it concise and efficient.
* **Edge cases**:
  + Single row (m = 1): Traverses left to right (handled by initial steps[0]).
  + Single column (n = 1): Traverses top to bottom (handled by steps[1]).
  + Empty matrix: Not applicable (problem assumes non-empty matrix).
  + Square or rectangular matrix: Works for any m x n dimensions.
* **Alternative approaches**:
  + Boundary tracking: Use four boundaries (top, bottom, left, right) and shrink them after each direction (O(m \* n) time, O(1) auxiliary space), more common but slightly more verbose.
  + Recursive layer peeling: Process outer layer and recurse on inner matrix (O(m \* n) time, O(m \* n) space due to recursion), less space-efficient.
* **Related problems**: This problem relates to "Rotate Image" (matrix manipulation), "Set Matrix Zeroes" (in-place matrix processing), and "Matrix Diagonal Traverse" (matrix traversal patterns) in the Blind 75 list.
* **Interview tip**: Explain the direction array and how steps[d % 2] alternates between horizontal and vertical movements. Be ready to discuss the boundary-tracking approach and edge cases like single row/column matrices.

1. Set Matrix Zeroes (Math & Geometry)

## Problem Description

* Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0. The operation must be done in-place, without using extra space proportional to the matrix size.

## Code with Comments

class Solution {

// Function to set rows and columns to zero where a zero is found, in-place

public void setZeroes(int[][] matrix) {

// Flag to track if the first column needs to be zeroed

Boolean firstCol = false;

// Get matrix dimensions: rows (r) and columns (c)

int r = matrix.length;

int c = matrix[0].length;

// First pass: mark rows and columns for zeroing

for (int i = 0; i < r; i++) {

// Check if first column has a zero

if (matrix[i][0] == 0) {

firstCol = true;

}

// Check remaining columns for zeros

for (int j = 1; j < c; j++) {

// If a zero is found, mark the first cell of the row and column

if (matrix[i][j] == 0) {

matrix[i][0] = 0; // Mark first cell of row i

matrix[0][j] = 0; // Mark first cell of column j

}

}

}

// Second pass: set elements to zero based on marks, excluding first row/column

for (int i = 1; i < r; i++) {

for (int j = 1; j < c; j++) {

// If first cell of row i or column j is marked, set element to zero

if (matrix[i][0] == 0 || matrix[0][j] == 0) {

matrix[i][j] = 0;

}

}

}

// Handle first row separately if it was marked

if (matrix[0][0] == 0) {

for (int j = 0; j < c; j++) {

matrix[0][j] = 0; // Set entire first row to zero

}

}

// Handle first column if it was marked

if (firstCol) {

for (int i = 0; i < r; i++) {

matrix[i][0] = 0; // Set entire first column to zero

}

}

}

}

**Approach**

* **In-place marking with first row/column**: Use the first row and first column of the matrix as markers to indicate which rows and columns should be set to zero, minimizing extra space.
* **Track first column**: Use a boolean firstCol to track if the first column contains a zero, as the first cell matrix[0][0] is shared between the first row and column.
* **First pass (marking)**:
  + Check if the first column has any zeros and set firstCol accordingly.
  + For each cell (i, j) (excluding first column), if matrix[i][j] == 0, mark matrix[i][0] and matrix[0][j] as 0 to indicate row i and column j should be zeroed.
* **Second pass (zeroing)**:
  + For cells (i, j) (excluding first row and column), set matrix[i][j] = 0 if either matrix[i][0] or matrix[0][j] is 0.
* **Handle first row and column**:
  + If matrix[0][0] == 0, zero the entire first row.
  + If firstCol is true, zero the entire first column.
* **Optimization**: Using the matrix’s first row and column as markers eliminates the need for extra space, satisfying the in-place requirement.

**Time Complexity**

* **O(m \* n)**:
  + First pass: Iterate over all m \* n elements (skipping first column for inner loop), O(m \* n).
  + Second pass: Iterate over (m - 1) \* (n - 1) elements, O(m \* n).
  + Zeroing first row and column: O(n) and O(m), respectively.
  + Total time complexity is O(m \* n), where m is the number of rows and n is the number of columns.

**Space Complexity**

* **O(1)**:
  + The solution uses only a single boolean variable (firstCol) and modifies the matrix in-place.
  + No additional data structures proportional to the matrix size are used.
  + Total space complexity is O(1), meeting the problem’s in-place requirement.

**Key Takeaways**

* **In-place efficiency**: Using the first row and column as markers cleverly avoids extra space while handling the zeroing logic correctly.
* **Edge cases**:
  + Single row or column: Handled by checking first column and row separately.
  + No zeros: No changes made to the matrix.
  + All zeros: Entire matrix set to zero.
  + Zeros in first row/column: firstCol and matrix[0][0] ensure correct handling.
* **Alternative approaches**:
  + Extra space: Use separate arrays to mark rows and columns (O(m \* n) time, O(m + n) space), simpler but violates in-place constraint.
  + Brute force: Copy matrix and zero rows/columns (O(m \* n) time, O(m \* n) space), not allowed.
* **Related problems**: This problem relates to "Rotate Image" (in-place matrix manipulation), "Spiral Matrix" (matrix traversal), and "Game of Life" (in-place state updates) in the Blind 75 list.
* **Interview tip**: Explain the use of the first row/column as markers and why firstCol is needed to avoid ambiguity with matrix[0][0]. Be ready to discuss trade-offs of using extra space and edge cases like zeros in the first row/column.

1. Number of 1 Bits (Bit Manipulation)

## Problem Description

* Given a 32-bit unsigned integer n, return the number of 1 bits (Hamming weight) in its binary representation.

class Solution {

// Function to calculate the number of 1 bits in the binary representation of n

public int hammingWeight(int n) {

// Initialize result to count the number of 1 bits

int res = 0;

// Continue until n becomes 0

while (n != 0) {

// Use bitwise AND with (n-1) to clear the least significant 1 bit

n = n & (n - 1);

// Increment the count of 1 bits

res++;

}

// Return the total count of 1 bits

return res;

}

}

**Approach**

* **Bit manipulation with Brian Kernighan’s algorithm**: Use a bitwise operation to efficiently count the number of 1 bits in n.
* **Initialize counter**: Set res to 0 to track the count of 1 bits.
* **Clear least significant 1 bit**: In each iteration:
  + Perform n & (n - 1) to clear the rightmost 1 bit in n. For example, if n = 1011, then n - 1 = 1010, and n & (n - 1) = 1010.
  + Increment res to count the cleared 1 bit.
* **Loop until** n **is 0**: Repeat until all 1 bits are cleared (n == 0).
* **Return result**: Return the count of 1 bits stored in res.
* **Optimization**: This approach is highly efficient because it iterates only as many times as there are 1 bits, not the full 32 bits of the integer.

**Time Complexity**

* **O(k)**:
  + The loop runs k times, where k is the number of 1 bits in n.
  + In the worst case (e.g., n = 0xFFFFFFFF, all 1s), k = 32, which is constant since n is a 32-bit integer.
  + Each iteration involves O(1) operations (bitwise AND, increment).
  + Total time complexity is O(k), effectively O(1) for a 32-bit integer.

**Space Complexity**

* **O(1)**:
  + The solution uses only one variable (res), regardless of the input.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Brian Kernighan’s algorithm**: The n & (n - 1) trick is an efficient way to count 1 bits by clearing the least significant 1 bit in each iteration, reducing the number of iterations to the number of 1 bits.
* **Edge cases**:
  + n = 0: Returns 0 (no 1 bits, handled by loop not running).
  + n = 0xFFFFFFFF: Returns 32 (all bits are 1, handled by counting each bit).
  + Negative numbers: Not applicable, as the problem assumes an unsigned 32-bit integer (in Java, treated as a signed int but interpreted as unsigned for bitwise operations).
* **Alternative approaches**:
  + Bit-by-bit check: Shift n right 32 times and check each bit with n & 1 (O(1) time for 32 iterations, O(1) space), less efficient due to fixed 32 iterations.
  + Lookup table: Precompute bit counts for smaller chunks (e.g., bytes) and sum them (O(1) time, O(1) space), but more complex.
* **Related problems**: This problem relates to "Sum of Two Integers" (bit manipulation), "Reverse Bits" (bitwise operations), and "Single Number" (XOR operations) in the Blind 75 list.
* **Interview tip**: Highlight the efficiency of Brian Kernighan’s algorithm compared to checking each bit. Be ready to explain how n & (n - 1) works (e.g., 1011 & 1010 = 1010) and discuss edge cases like n = 0 or all 1s.

1. Counting Bits (Bit Manipulation)

## Problem Description

* Given an integer n, return an array ans of length n + 1 where ans[i] is the number of 1 bits (Hamming weight) in the binary representation of i for each i from 0 to n.

**Code with Comments**

class Solution {

// Function to compute the number of 1 bits for each integer from 0 to n

public int[] countBits(int n) {

// Initialize result array of size n+1 to store bit counts

int[] ans = new int[n + 1];

// Set base case: 0 has zero 1 bits

ans[0] = 0;

// Iterate through each number from 1 to n

for (int i = 1; i <= n; i++) {

// Compute bit count using the formula: ans[i] = ans[i & (i-1)] + 1

ans[i] = ans[i & (i - 1)] + 1;

}

// Return the result array

return ans;

}

}

**Approach**

* **Dynamic programming with bit manipulation**: Use the fact that i & (i - 1) clears the least significant 1 bit in i to compute the number of 1 bits efficiently.
* **Initialize result array**: Create an array ans of size n + 1 to store the count of 1 bits for each integer from 0 to n.
* **Base case**: Set ans[0] = 0, as the binary representation of 0 has no 1 bits.
* **Iterate and compute**:
  + For each i from 1 to n, compute ans[i] using the formula ans[i & (i - 1)] + 1.
  + The operation i & (i - 1) produces a number with one fewer 1 bit than i (by clearing the rightmost 1 bit). Thus, ans[i] = ans[i & (i - 1)] + 1 accounts for the additional 1 bit in i.
* **Return result**: Return the ans array containing the bit counts.
* **Optimization**: This approach avoids recomputing bit counts for each number by leveraging previously computed results, making it more efficient than counting bits individually.

**Time Complexity**

* **O(n)**:
  + The loop iterates from 1 to n, performing n iterations.
  + Each iteration involves O(1) operations: a bitwise AND (i & (i - 1)), array access, and addition.
  + Total time complexity is O(n).

**Space Complexity**

* **O(n)**:
  + The solution uses an array ans of size n + 1 to store the bit counts.
  + No additional data structures are used, making the space complexity O(n) (excluding the output array, auxiliary space is O(1)).

**Key Takeaways**

* **Bit manipulation with DP**: The solution efficiently uses i & (i - 1) to relate the bit count of i to a smaller number, combining bit manipulation with dynamic programming to avoid redundant calculations.
* **Edge cases**:
  + n = 0: Returns [0] (handled by initializing ans[0] = 0).
  + n = 1: Returns [0, 1] (handled by loop setting ans[1] = ans[0] + 1).
  + Large n: Works efficiently due to constant-time operations per iteration.
* **Alternative approaches**:
  + Naive bit counting: For each i, count 1 bits using i & (i - 1) or right shifts (O(n \* log n) time, O(n) space), less efficient due to repeated bit counting.
  + Pattern-based DP: Use the observation that ans[i] = ans[i >> 1] + (i & 1) (O(n) time, O(n) space), equally efficient but different logic.
* **Related problems**: This problem relates to "Number of 1 Bits" (bit counting), "Reverse Bits" (bit manipulation), and "Single Number" (bitwise operations) in the Blind 75 list.
* **Interview tip**: Explain how i & (i - 1) reduces the problem to a previously solved subproblem and why it’s efficient. Be ready to discuss the alternative pattern-based approach and edge cases like n = 0.

1. Reverse Bits (Bit Manipulation)

## Problem Description

* Given a 32-bit unsigned integer n, reverse its bits and return the result as an unsigned integer. For example, if the input is 00000010100101000001111010011100, the output is 00111001011110000010100101000000.

1. Missing Number (Bit Manipulation)

## Problem Description

* 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.

**Code with Comments**

class Solution {

// Function to find the missing number in the range [0, n]

public int missingNumber(int[] nums) {

// Get the length of the array (n)

int n = nums.length;

// Calculate the expected sum of numbers from 0 to n using the formula n\*(n+1)/2

int expectedSum = n \* (n + 1) / 2;

// Initialize variable to store the actual sum of numbers in the array

int actualSum = 0;

// Iterate through each number in the array

for (int num : nums) {

// Add the current number to the actual sum

actualSum += num;

}

// Return the missing number by subtracting actual sum from expected sum

return expectedSum - actualSum;

}

}

**Approach**

* **Mathematical sum formula**: Use the fact that the sum of the first n + 1 numbers (from 0 to n) can be calculated using the formula n \* (n + 1) / 2.
* **Get array length**: Store the length of nums as n, which represents the range [0, n].
* **Calculate expected sum**: Compute the expected sum of all numbers from 0 to n using the formula.
* **Calculate actual sum**: Iterate through nums to compute the sum of all numbers in the array.
* **Find missing number**: Subtract the actual sum from the expected sum to find the missing number.
* **Optimization**: This approach is efficient as it requires only one pass through the array and uses a mathematical formula to avoid sorting or searching.

**Time Complexity**

* **O(n)**:
  + The loop to compute the actual sum iterates through the array once, taking O(n) time, where n is the length of nums.
  + All other operations (calculating expectedSum, subtraction) are O(1).
  + Total time complexity is O(n).

**Space Complexity**

* **O(1)**:
  + The solution uses only a few variables (n, expectedSum, actualSum), regardless of the input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Mathematical elegance**: The solution leverages the arithmetic series sum formula to compute the expected sum, making it both intuitive and efficient.
* **Edge cases**:
  + n = 0 (empty array): Not applicable, as the problem guarantees n >= 1.
  + Missing number is 0: Handled correctly, as expectedSum includes 0, and actualSum will be short by 0.
  + Missing number is n: Handled correctly, as expectedSum includes n, and actualSum will be short by n.
* **Alternative approaches**:
  + XOR method: XOR all numbers from 0 to n with all numbers in nums (O(n) time, O(1) space). The result is the missing number due to XOR’s self-cancellation property.
  + Sorting: Sort the array and find the missing number by checking gaps (O(n log n) time, O(1) space), less efficient.
  + HashSet: Store numbers in a set and check for missing number (O(n) time, O(n) space), less space-efficient.
* **Related problems**: This problem relates to "Single Number" (XOR-based solution), "Find All Numbers Disappeared in an Array" (array manipulation), and "Number of 1 Bits" (bit manipulation) in the Blind 75 list.
* **Interview tip**: Highlight the simplicity of the sum-based approach and its O(1) space efficiency. Be ready to explain the XOR-based alternative and discuss edge cases like missing 0 or n.

1. Sum of Two Integers (Bit Manipulation)

## Problem Description

* Given two integers a and b, return their sum without using the + or - operators. You must use bitwise operations to compute the sum.

## Code with Comments

public class Solution {

// Function to compute the sum of two integers using bitwise operations

public int getSum(int a, int b) {

// Continue until there is no carry

while (b != 0) {

// Compute the carry using AND and left shift by 1

int carry = (a & b) << 1;

// Compute the sum without carry using XOR

a ^= b;

// Assign the carry to b for the next iteration

b = carry;

}

// Return the final sum stored in a

return a;

}

}

**Approach**

* **Bitwise simulation of addition**: Use bitwise operations to mimic the process of adding two numbers as done in binary arithmetic.
* **Iterative process**:
  + **XOR for sum without carry**: The XOR operation (a ^ b) computes the sum of a and b for each bit position without considering the carry (e.g., 1 ^ 1 = 0, 1 ^ 0 = 1).
  + **AND and shift for carry**: The AND operation (a & b) identifies bits where both numbers have 1s, generating a carry. Left-shifting by 1 (<< 1) moves the carry to the correct position.
  + **Update variables**: Assign the XOR result to a (sum without carry) and the carry to b.
  + **Loop until no carry**: Repeat until b (the carry) becomes 0, indicating no further carries are needed.
* **Return result**: Once b == 0, a contains the final sum.
* **Optimization**: The solution avoids using arithmetic operators, relying solely on bitwise operations, and handles both positive and negative integers correctly due to Java’s two’s complement representation.

**Time Complexity**

* **O(log m)**:
  + The number of iterations depends on the number of bits in the carry, which is at most the number of bits in the larger number m = max(|a|, |b|).
  + In the worst case (e.g., a = 0xFFFFFFFF, b = 1), the carry propagates through all 32 bits, requiring O(log m) iterations.
  + Each iteration involves O(1) bitwise operations (AND, XOR, shift).
  + Total time complexity is O(log m), effectively O(1) for 32-bit integers.

**Space Complexity**

* **O(1)**:
  + The solution uses only a few variables (carry, a, b), regardless of the input size.
  + No additional data structures are used, making the space complexity O(1).

**Key Takeaways**

* **Bitwise addition**: The solution elegantly simulates binary addition using XOR for the sum and AND with left shift for the carry, avoiding arithmetic operators as required.
* **Edge cases**:
  + Both numbers are 0: Returns 0 (handled as b = 0 immediately).
  + Negative numbers: Works correctly due to two’s complement representation in Java.
  + Large numbers: Handles full 32-bit integers, including cases with significant carries.
* **Alternative approaches**:
  + Recursive solution: Use recursion instead of a loop to compute a ^ b and (a & b) << 1 (O(log m) time, O(log m) space due to recursion stack), less space-efficient.
  + Lookup table: Not practical for arbitrary integers but possible for small ranges (O(1) time, O(1) space with precomputation).
* **Related problems**: This problem relates to "Number of 1 Bits" (bit counting), "Reverse Bits" (bit manipulation), and "Counting Bits" (bitwise operations with DP) in the Blind 75 list.
* **Interview tip**: Explain how XOR and AND simulate addition (XOR for sum, AND for carry) and why the loop terminates. Be ready to discuss handling negative numbers and the recursive approach.