**NeetCode 150 - Complete Topic-wise List**

1. Arrays & Hashing (9 problems)

* Contains Duplicate

Commented Code

class Solution {

*// Method to check if the input array contains any duplicate elements*

public boolean hasDuplicate(int[] nums) {

*// Initialize a HashSet to store unique elements*

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

*// Iterate through each element in the input array*

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

*// If the current element is already in the HashSet, a duplicate is found*

if (seen.contains(nums[i])) {

return true; *// Return true as soon as a duplicate is detected*

}

*// Add the current element to the HashSet*

seen.add(nums[i]);

}

*// If no duplicates are found after the loop, return false*

return false;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The code iterates through the array of length n exactly once using a single for loop.
  + For each element, it performs:
    - A contains operation on the HashSet, which has an average time complexity of O(1).
    - An add operation on the HashSet, which also has an average time complexity of O(1).
  + Thus, the total time complexity is O(n) \* O(1) = **O(n)**, where n is the length of the input array.
  + **Note**: In the worst case (rare due to hash collisions), contains and add could degrade to O(n), but this is unlikely with a good hash function.

**Space Complexity**

* **Space Complexity: O(n)**
  + The HashSet stores at most n unique elements from the input array (in the case where there are no duplicates until the end).
  + Therefore, the space required by the HashSet is proportional to the input size, leading to a space complexity of **O(n)**.
  + No additional significant data structures are used, so the space complexity remains O(n).

**Concise Summary of the Approach**

The solution checks for duplicates in an array by utilizing a HashSet to track unique elements. It iterates through the array once, checking if each element exists in the HashSet. If an element is already present, a duplicate is found, and the method returns true. Otherwise, the element is added to the HashSet. If the loop completes without finding duplicates, the method returns false. This approach is efficient with **O(n)** time complexity and **O(n)** space complexity, leveraging the constant-time operations of a HashSet for lookups and insertions.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: The use of a HashSet ensures O(n) time complexity, which is optimal for this problem compared to naive approaches like nested loops (O(n²)).
  + **Simplicity**: The code is concise, readable, and easy to explain in an interview setting.
  + **Robustness**: The solution handles edge cases well, such as empty arrays or arrays with a single element (both return false implicitly).
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it without extra space (i.e., O(1) space complexity)?
    - **Response**: If the array can be modified, sort the array (O(n log n) time, O(1) space) and check adjacent elements for duplicates. If modification is not allowed, the HashSet approach is optimal for time efficiency.
  + **Interviewer might ask**: What if the array contains negative numbers or very large numbers?
    - **Response**: The HashSet approach works for any integer values, as HashSet in handles all integer types effectively.
  + **Interviewer might ask**: How would you handle hash collisions affecting performance?
    - **Response**: Hash collisions are rare with ’s HashSet due to its robust hashing mechanism, but in pathological cases, performance could degrade. Using a balanced binary search tree (e.g., TreeSet) could provide O(n log n) time with guaranteed performance but is typically unnecessary.
* **Edge Cases Handled**:
  + Empty array: Returns false (no duplicates possible).
  + Single-element array: Returns false (no duplicates possible).
  + Array with all identical elements: Returns true on the second occurrence.
  + Array with negative or large numbers: Handled correctly by HashSet.
* **Alternative Approaches**:
  + **Sorting**: Sort the array and check for adjacent duplicates. Time: O(n log n), Space: O(1) if in-place sorting is allowed.
  + **Brute Force**: Compare each element with all others. Time: O(n²), Space: O(1).
  + The HashSet approach is typically preferred in interviews due to its balance of simplicity and efficiency.

**Suggestions for Improvement**

* **Code Clarity**: The code is already clear, but adding a brief comment about the purpose of the HashSet (e.g., "to track unique elements") enhances readability.
* **Edge Case Explicit Handling**: While not necessary, explicitly checking for nums.length <= 1 and returning false could make the intent clearer, though it’s redundant given the current logic.
* **Alternative Data Structures**: In rare cases where memory is a concern, discuss trade-offs with sorting or other approaches during the interview.
* Valid Anagram

Commented Code

class Solution {

*// Method to check if two strings s and t are anagrams*

public boolean isAnagram(String s, String t) {

*// If lengths differ, they cannot be anagrams*

if (s.length() != t.length()) {

return false;

}

*// Array to store character frequency counts (for lowercase letters a-z)*

int[] charCounts = new int[26];

*// Iterate through both strings simultaneously*

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

*// Increment count for character in s*

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

*// Decrement count for character in t*

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

}

*// Check if all character counts are zero (indicating equal frequencies)*

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

if (charCounts[i] != 0) {

return false; *// Non-zero count means strings are not anagrams*

}

}

*// If all counts are zero, strings are anagrams*

return true;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The initial length check is O(1).
  + The first loop iterates through both strings of length n, performing constant-time operations (array increment/decrement) for each character, resulting in O(n).
  + The second loop iterates over the fixed-size array of 26 elements (for lowercase letters), which is O(1) since it’s constant regardless of input size.
  + Overall, the dominant term is O(n), where n is the length of the input strings.
* **Note**: The time complexity assumes that string character access (charAt) is O(1), which is true in for standard strings.

**Space Complexity**

* **Space Complexity: O(1)**
  + The charCounts array has a fixed size of 26 (for lowercase English letters), which is constant regardless of input size.
  + No additional data structures scale with input size, so the space complexity is O(1).
  + **Note**: If the problem allowed for a larger character set (e.g., Unicode), a larger array or a different data structure like a HashMap might be needed, which could increase space complexity to O(k), where k is the size of the character set.

**Concise Summary of the Approach**

The solution determines if two strings are anagrams by first checking if their lengths are equal (if not, they cannot be anagrams). It then uses a fixed-size array of 26 elements to track the frequency of each lowercase letter. For each character in both strings, it increments the count for the character in the first string and decrements it for the second string. Finally, it checks if all counts in the array are zero, indicating identical character frequencies. The approach is efficient with **O(n)** time complexity and **O(1)** space complexity, leveraging a single array for constant space.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: Achieves O(n) time complexity and O(1) space complexity, making it optimal for the problem when restricted to lowercase English letters.
  + **Simplicity**: The code is concise, intuitive, and avoids complex data structures like hash maps, which is ideal for interview settings.
  + **Clever Optimization**: Simultaneously processing both strings in one loop reduces the number of passes through the data.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: What if the strings contain Unicode characters or uppercase letters?
    - **Response**: For Unicode, use a HashMap to store character frequencies, increasing space complexity to O(k), where k is the number of unique characters. For uppercase letters, either convert strings to lowercase first (O(n) time) or use a larger array (e.g., size 52 for both cases).
  + **Interviewer might ask**: Can you solve it using sorting?
    - **Response**: Yes, convert both strings to char arrays, sort them (O(n log n)), and compare for equality. This uses O(n) space (or O(log n) for sorting in some cases) but is less efficient in time compared to the array-based approach.
  + **Interviewer might ask**: How does this handle edge cases?
    - **Response**: The length check handles empty strings or strings of different lengths. The array-based approach assumes lowercase letters, which is a common constraint (e.g., LeetCode’s "Valid Anagram" problem). If constraints differ, clarify and adjust (e.g., use a HashMap).
* **Edge Cases Handled**:
  + Empty strings: If both are empty, returns true (valid anagram). If one is empty and the other isn’t, the length check returns false.
  + Strings of different lengths: Returns false immediately.
  + Strings with same characters but different frequencies: The charCounts array will have non-zero values, returning false.
  + Case sensitivity: Assumes lowercase (per problem constraints). If not specified, clarify with the interviewer.
* **Alternative Approaches**:
  + **Sorting**: Convert strings to char arrays, sort, and compare. Time: O(n log n), Space: O(n) or O(1) if sorting in-place.
  + **HashMap**: Use a HashMap to count frequencies in s, then decrement for t. Time: O(n), Space: O(k), where k is the number of unique characters.
  + The array-based approach is preferred for lowercase English letters due to its O(1) space complexity.
* **Assumptions**:
  + The code assumes input strings contain only lowercase English letters (a-z), as implied by the use of a 26-element array and the 'a' offset.
  + If the problem allows other characters, clarify with the interviewer or use a HashMap.

**Suggestions for Improvement**

* **Input Validation**: Add explicit checks for null strings (e.g., if (s == null || t == null) return false;) if not guaranteed by the problem.
* **Clarity**: The code is clear, but a comment noting the assumption of lowercase letters (e.g., “Assumes lowercase a-z”) would improve readability.
* **Generalization**: If the problem allows for uppercase letters or Unicode, mention in the interview that you can adapt the solution using a HashMap or larger array, and discuss trade-offs.
* **Optimization**: For very short strings, sorting might be competitive due to cache efficiency, but the array approach is generally better for larger inputs.
* Two Sum

Commented Code

class Solution {

*// Method to find two indices in nums whose elements sum to target*

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 element*

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

}

*// Store the current number and its index in the HashMap*

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

}

*// Return empty array if no solution is found*

return new int[] {};

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The code iterates through the array of length n exactly once using a single for loop.
  + For each element, it performs:
    - A containsKey operation on the HashMap, which has an average time complexity of O(1).
    - A put operation on the HashMap, which also has an average time complexity of O(1).
  + Thus, the total time complexity is O(n) \* O(1) = **O(n)**, where n is the length of the input array.
  + **Note**: In the worst case (rare due to hash collisions), containsKey and put could degrade to O(n), but this is unlikely with a good hash function.

**Space Complexity**

* **Space Complexity: O(n)**
  + The HashMap stores at most n key-value pairs (number-to-index mappings), where each key is an integer from the input array and each value is its corresponding index.
  + The output array is of fixed size (2 elements), which is O(1) and does not affect the overall space complexity.
  + Therefore, the space complexity is **O(n)** due to the HashMap.

**Concise Summary of the Approach**

The solution finds two indices in an array whose elements sum to a given target using a HashMap to store number-to-index mappings. It iterates through the array once, calculating the complement (target - nums[i]) for each element. If the complement exists in the HashMap, it returns the indices of the complement and the current element. Otherwise, it adds the current number and its index to the HashMap. If no solution is found, it returns an empty array. The approach is efficient with **O(n)** time complexity and **O(n)** space complexity, leveraging constant-time HashMap operations.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: Achieves O(n) time complexity, which is optimal compared to the naive O(n²) approach using nested loops.
  + **Simplicity**: The code is concise, readable, and easy to explain, making it ideal for interviews.
  + **Robustness**: Handles the problem constraints well, assuming exactly one valid solution exists (as per the standard "Two Sum" problem, e.g., LeetCode).
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it with O(1) space complexity?
    - **Response**: If the array can be modified, sorting and using two pointers achieves O(n log n) time and O(1) space (excluding the output array). However, this is less efficient in time compared to the HashMap approach. Without modification, O(n) space is typically necessary for O(n) time.
  + **Interviewer might ask**: What if there are multiple valid pairs?
    - **Response**: The current solution returns the first valid pair found. If all pairs are needed, modify the code to collect all valid index pairs in a list, still maintaining O(n) time.
  + **Interviewer might ask**: How does this handle negative numbers or duplicates?
    - **Response**: The HashMap approach works for any integer values (positive, negative, or zero) and handles duplicates correctly since it stores the most recent index for each number, which is sufficient for finding one valid pair.
* **Edge Cases Handled**:
  + Empty array or array with fewer than 2 elements: Returns an empty array (no solution possible).
  + No solution exists: Returns an empty array as required.
  + Duplicate numbers: The HashMap overwrites indices, but since we check the complement before adding the current number, it correctly identifies valid pairs.
  + Negative numbers or zero: Handled seamlessly by the HashMap.
* **Assumptions**:
  + The problem assumes exactly one valid solution exists (standard for "Two Sum").
  + The input array has at least two elements, and the target is achievable.
  + If these assumptions don’t hold, clarify with the interviewer (e.g., handle no solution, multiple solutions, or invalid inputs).
* **Alternative Approaches**:
  + **Brute Force**: Use nested loops to check all pairs. Time: O(n²), Space: O(1). Inefficient but simple.
  + **Sorting + Two Pointers**: Sort the array (with indices tracked) and use two pointers to find the pair. Time: O(n log n), Space: O(n) for storing indices or O(1) if in-place sorting is allowed. Less efficient in time but reduces space if modification is permitted.
  + The HashMap approach is typically preferred in interviews due to its optimal time complexity and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null arrays or arrays with fewer than 2 elements (e.g., if (nums == null || nums.length < 2) return new int[] {};) if not guaranteed by the problem.
  + **Clarity**: The code is clear, but a comment explaining the HashMap’s role (e.g., “Stores number-to-index mappings”) enhances readability.
  + **Edge Case Discussion**: In an interview, proactively mention edge cases (e.g., duplicates, negative numbers) and confirm problem constraints to demonstrate thoroughness.
* Group Anagrams

Commented Code

class Solution {

*// Method to group anagrams from an array of strings*

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

*// Initialize a HashMap to map sorted strings to lists of anagrams*

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

*// Iterate through each string in the input array*

for (String word : strs) {

*// Convert the string to a char array and sort it*

char[] chars = word.toCharArray();

Arrays.sort(chars);

*// Create a string from the sorted char array (key for anagram group)*

String sortedWords = new String(chars);

*// If the sorted string is not in the map, add a new empty list*

if (!map.containsKey(sortedWords)) {

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

}

*// Add the original word to the list corresponding to its sorted key*

map.get(sortedWords).add(word);

}

*// Return the list of anagram groups (values of the HashMap)*

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

}

}

**Time Complexity**

* **Time Complexity: O(n \* k \* log k)**
  + The code iterates through the input array of n strings once.
  + For each string of maximum length k:
    - Converting the string to a char array is O(k).
    - Sorting the char array using Arrays.sort (which uses Timsort) is O(k log k).
    - Creating a new string from the sorted array is O(k).
    - HashMap operations (containsKey, put, get) are O(1) on average.
  + For n strings, the total time is O(n \* (k + k log k + k)) ≈ O(n \* k log k), as the sorting term dominates.
  + **Note**: The final map.values() and new ArrayList<>(map.values()) operations are O(n) in practice, as they involve copying the grouped lists, but this is typically negligible compared to the sorting cost.

**Space Complexity**

* **Space Complexity: O(n \* k)**
  + The HashMap stores at most n strings, grouped by their sorted versions. Each string (of length up to k) is stored in the map’s value lists, leading to O(n \* k) storage for the strings themselves.
  + The keys in the HashMap (sorted strings) also take O(k) per unique anagram group, but the number of groups is at most n, so this is bounded by O(n \* k).
  + Temporary variables (e.g., char[] chars, sortedWords) use O(k) space per iteration, which is reused and does not scale with n.
  + The output list contains all input strings grouped, which is O(n \* k).
  + Overall, the space complexity is **O(n \* k)**, dominated by the storage of the input strings in the HashMap and output.

**Concise Summary of the Approach**

The solution groups anagrams from an array of strings by using a HashMap to map sorted versions of strings to lists of their anagrams. For each string, it sorts its characters to create a key, then adds the original string to the corresponding list in the HashMap. If the key doesn’t exist, a new list is created. Finally, it returns all lists of anagrams (the HashMap’s values). The approach is efficient with **O(n \* k log k)** time complexity (due to sorting) and **O(n \* k)** space complexity, leveraging the HashMap for grouping.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: The sorting-based key generation ensures anagrams are grouped correctly, with O(n \* k log k) being near-optimal for this problem.
  + **Clarity**: The code is concise, readable, and easy to explain, making it ideal for interviews.
  + **Flexibility**: The HashMap approach naturally handles all edge cases and varying string lengths.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you optimize the time complexity?
    - **Response**: Instead of sorting, use a character frequency array (or string) as the key (e.g., a count of each letter). This reduces the time to O(n \* k) by avoiding sorting, but requires O(k) to create the key (e.g., a string of counts). For lowercase letters, use a 26-element array; for Unicode, use a HashMap. Discuss trade-offs (e.g., increased space for frequency arrays).
  + **Interviewer might ask**: How does this handle edge cases like empty strings or null inputs?
    - **Response**: Empty strings are handled (sorted as empty, grouped together). For null inputs, clarify if the input array or individual strings can be null and add checks (e.g., if (strs == null) or skip null strings).
  + **Interviewer might ask**: What if the strings contain Unicode characters?
    - **Response**: The sorting approach still works, but for the frequency-based optimization, use a HashMap for character counts instead of a fixed-size array, increasing space complexity to O(n \* k + k \* c), where c is the size of the character set.
* **Edge Cases Handled**:
  + Empty array: Returns an empty list of lists.
  + Array with one string: Returns a list containing a single list with that string.
  + Empty strings: Treated as valid anagrams of each other (sorted as empty).
  + Strings with same characters but different frequencies: Sorted keys differ, so they’re grouped separately.
  + **Note**: The code assumes no null strings or null array; add checks if required (e.g., if (strs == null) return new ArrayList<>();).
* **Assumptions**:
  + The input array is non-null, and strings are valid (non-null, possibly empty).
  + Strings may contain any characters (though sorting works for any character set, including Unicode).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Character Frequency Key**: Instead of sorting, create a key based on character frequencies (e.g., a string like "a:2,b:1" or a 26-element array for lowercase letters). Time: O(n \* k), Space: O(n \* k). Preferred for small character sets (e.g., lowercase letters).
  + **Brute Force**: Compare each string with all others to check for anagrams. Time: O(n² \* k), Space: O(n \* k). Highly inefficient.
  + The sorting-based approach is a good balance of simplicity and efficiency, but mention the frequency-based approach in interviews to show optimization awareness.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null array or null strings (e.g., if (strs == null) return new ArrayList<>(); or skip null strings in the loop).
  + **Optimization**: If the problem is constrained to lowercase letters, use a frequency array as the key to reduce time to O(n \* k). Example key: "1,2,0,..." (counts of a-z).
  + **Clarity**: Add comments explaining the sorting step (e.g., “Sort characters to create a key for anagram grouping”).
  + **Memory Efficiency**: For very large strings, discuss reusing a single char array or StringBuilder to reduce temporary object creation.
* Top K Frequent Elements

Commented Code

class Solution {

*// Method to find the k most frequent elements in an array*

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

*// If k equals the array length, return the entire array*

if (k == nums.length) {

return nums;

}

*// Initialize a HashMap to store the frequency of each number*

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

*// Count the frequency of each number in the array*

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

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

}

*// Initialize a min-heap to keep track of the k most frequent elements*

*// Compare elements based on their frequencies (ascending order)*

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

*// Add each number to the heap, maintaining size <= k*

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

heap.add(n);

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

if (heap.size() > k) {

heap.poll();

}

}

*// Build the 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 the k most frequent elements*

return ans;

}

}

**Time Complexity**

* **Time Complexity: O(n log n)**
  + Building the frequency map: Iterating through the array of length n and performing put and getOrDefault operations on the HashMap (O(1) average) takes O(n).
  + Building the heap: There are at most n unique elements in the HashMap. Adding each element to the heap is O(log m), where m is the current heap size (at most k). Since k ≤ n, inserting all unique elements (up to n) into the heap, with removals to maintain size k, takes O(n log k). In the worst case (many unique elements), this is O(n log n).
  + Extracting k elements from the heap: Polling k times from a heap of size at most k is O(k log k).
  + Overall, the dominant term is O(n log k) for heap operations, but since k ≤ n, we conservatively state O(n log n) to account for the worst case with many unique elements.
* **Note**: If k is much smaller than n, the heap operations are effectively O(n log k), which is more precise for typical cases.

**Space Complexity**

* **Space Complexity: O(n)**
  + The HashMap stores at most n unique elements and their frequencies, requiring O(n) space.
  + The PriorityQueue (min-heap) stores at most k elements, which is O(k) space, where k ≤ n.
  + The output array is of size k, which is O(k).
  + Overall, the space complexity is dominated by the HashMap, resulting in **O(n)**.

**Concise Summary of the Approach**

The solution finds the k most frequent elements in an array by first counting the frequency of each number using a HashMap. It then uses a min-heap to maintain the k elements with the highest frequencies, comparing elements based on their counts. For each unique number, it adds it to the heap and removes the least frequent if the heap size exceeds k. Finally, it extracts the k elements from the heap into the result array. The approach is efficient with **O(n log n)** time complexity and **O(n)** space complexity, leveraging a HashMap for counting and a min-heap for selecting the top k elements.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: Using a min-heap ensures O(n log k) for heap operations (or O(n log n) in the worst case), which is efficient compared to sorting all frequencies (O(n log n)).
  + **Clarity**: The code is straightforward, with clear separation of counting, heap processing, and result extraction, making it easy to explain in an interview.
  + **Correctness**: The min-heap guarantees the top k frequent elements are retained.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: Yes, use bucket sort: Create an array of lists where the index represents frequency (up to n). Place each number in the bucket corresponding to its frequency, then collect the top k elements from the highest-frequency buckets. Time: O(n), Space: O(n). This is faster for large n but requires more space for the bucket array.
  + **Interviewer might ask**: What if k is invalid (e.g., k > n or k <= 0)?
    - **Response**: Add validation (e.g., if (k <= 0 || k > nums.length) throw new IllegalArgumentException();). The current code assumes k is valid per problem constraints.
  + **Interviewer might ask**: How does this handle ties in frequency?
    - **Response**: The heap returns any k elements among those with the highest frequencies. If specific tie-breaking rules are needed (e.g., lexicographic order), modify the heap comparator to include the element value.
* **Edge Cases Handled**:
  + k == nums.length: Returns the entire array (handled explicitly).
  + Single element: If k == 1, returns the only element (heap handles this correctly).
  + All elements have the same frequency: Returns any k elements (heap selects arbitrarily).
  + Empty array: Not handled explicitly (assumes non-empty per problem constraints).
  + **Note**: Add checks for null or empty arrays if required (e.g., if (nums == null || nums.length == 0) return new int[] {};).
* **Assumptions**:
  + The input array is non-null and non-empty.
  + k is valid (1 ≤ k ≤ number of unique elements ≤ n).
  + The problem does not specify tie-breaking rules for equal frequencies.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Bucket Sort**: Use an array of lists indexed by frequency (0 to n). Place numbers in buckets based on their counts, then collect the top k from the highest buckets. Time: O(n), Space: O(n). Preferred for large n when O(n) time is critical.
  + **Sorting**: Sort the HashMap entries by frequency (descending) and take the top k. Time: O(n log n), Space: O(n). Simpler but less efficient than the heap approach.
  + The heap-based approach is a good balance of efficiency and simplicity for most cases.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null/empty arrays or invalid k (e.g., if (nums == null || nums.length == 0 || k <= 0 || k > nums.length) return new int[] {};).
  + **Order of Output**: The current code returns elements in ascending frequency order due to polling the min-heap. If descending frequency is desired, reverse the comparator or collect elements in reverse order into the result array (e.g., fill ans from index k-1 to 0).
  + **Clarity**: Add comments explaining the min-heap’s role (e.g., “Min-heap to maintain k elements with highest frequencies”).
  + **Optimization**: If the number of unique elements is small, the heap operations are faster (O(m log k), where m is the number of unique elements). Highlight this in interviews.
* **Note on Output Order**: The problem typically does not guarantee the order of elements with equal frequencies. The current code returns elements in ascending frequency order (due to min-heap polling). If descending order or specific tie-breaking is required, clarify with the interviewer and adjust the comparator or result collection.
* Product of Array Except Self

Commented Code

class Solution {

*// Method to compute the product of all elements in nums except self*

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

*// Initialize result array with the same length as input*

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

*// Fill result array with 1s to handle multiplication*

Arrays.fill(result, 1);

*// Initialize variables to track prefix and postfix products*

int pre = 1, post = 1;

*// First pass: Compute prefix products (product of all elements before i)*

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

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

pre = nums[i] \* pre; *// Update prefix product for next iteration*

}

*// Second pass: Multiply by postfix products (product of all elements after i)*

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

result[i] = result[i] \* post; *// Multiply prefix product by postfix product*

post = post \* nums[i]; *// Update postfix product for next iteration*

}

*// Return the result array containing products except self*

return result;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The code performs two separate passes through the array of length n:
    - First pass: Computes prefix products, iterating n times with O(1) operations per iteration.
    - Second pass: Computes postfix products and updates the result, iterating n times with O(1) operations per iteration.
  + Initializing the result array with Arrays.fill is O(n).
  + Total time complexity is O(n) + O(n) + O(n) = **O(n)**, where n is the length of the input array.

**Space Complexity**

* **Space Complexity: O(1)**
  + The result array of length n is required for output and is not counted as extra space per problem constraints (e.g., LeetCode’s standard).
  + The variables pre and post use O(1) space.
  + No additional data structures scale with input size, so the space complexity is **O(1)** (excluding the output array).

**Concise Summary of the Approach**

The solution computes the product of all array elements except the element at each index without using division. It uses two passes: the first pass computes the product of all elements before each index (prefix products) and stores them in the result array. The second pass multiplies each result by the product of all elements after the index (postfix products). By maintaining running prefix and postfix products, it achieves **O(n)** time complexity and **O(1)** space complexity (excluding the output array), efficiently avoiding division and extra space.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: Achieves O(n) time and O(1) space (excluding output), which is optimal given the problem’s constraints against division.
  + **Simplicity**: The two-pass approach is intuitive and avoids complex data structures, making it easy to explain.
  + **Compliance**: Meets the common problem constraint of not using division, which could fail with zeros in the input.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: How would you solve it using division?
    - **Response**: Compute the total product of non-zero elements and count zeros. For each index, if there’s no zero, divide the total product by nums[i]. If there’s one zero, the result is non-zero only at the zero’s index. If there are multiple zeros, the result is all zeros. Time: O(n), Space: O(1) (excluding output). Discuss division’s issues (e.g., handling zeros, overflow).
  + **Interviewer might ask**: How does this handle edge cases like zeros?
    - **Response**: The current approach works with zeros since it avoids division, correctly computing products of all other elements. For example, if nums[i] = 0, the result for other indices includes the zero, and for index i, the product excludes the zero.
  + **Interviewer might ask**: Can you explain why O(1) space is achieved?
    - **Response**: The result array is required for output, so it’s not counted. Only two scalar variables (pre, post) are used, and no additional data structures scale with input size, yielding O(1) extra space.
* **Edge Cases Handled**:
  + Array with one element: Not applicable, as the problem typically assumes nums.length >= 2.
  + Array with zeros: Handled correctly, as the approach avoids division.
  + Array with all identical elements: Correctly computes the product of n-1 identical elements for each index.
  + **Note**: Add checks for null or invalid input if required (e.g., if (nums == null || nums.length < 2) return new int[] {};).
* **Assumptions**:
  + The input array is non-null and has at least 2 elements.
  + The product of any prefix or suffix fits within a 32-bit integer (per problem constraints).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Division-Based**: Compute the total product and divide by each element. Time: O(n), Space: O(1). Fails with zeros or requires special handling.
  + **Two Arrays**: Use two arrays to store prefix and postfix products separately, then combine them. Time: O(n), Space: O(n). Less space-efficient but simpler to understand.
  + The current approach is preferred for its O(1) space efficiency and robustness with zeros.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or small arrays (e.g., if (nums == null || nums.length < 2) return new int[] {};) if not guaranteed.
  + **Clarity**: Add comments explaining the prefix/postfix logic (e.g., “First pass stores product of all elements before i”).
  + **Edge Case Discussion**: In an interview, proactively mention handling of zeros and integer overflow to show thoroughness.
  + **Optimization**: The current approach is already optimal, but discuss the division-based approach as a contrast, highlighting why it’s less robust.
* **Note on Output Order**: The result array stores products in the same order as the input indices, which is correct per the problem. The order of multiplication (prefix then postfix) ensures correctness without affecting the result.
* Valid Sudoku

Commented Code

class Solution {

*// Method to check if a 9x9 Sudoku board is valid*

public boolean isValidSudoku(char[][] board) {

int N = 9; *// Size of the Sudoku board (9x9)*

*// Initialize HashSets for each row, column, and 3x3 box to track digits*

HashSet<Character>[] rows = new HashSet[N];

HashSet<Character>[] cols = new HashSet[N];

HashSet<Character>[] boxes = new HashSet[N];

*// Initialize each HashSet*

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

rows[i] = new HashSet<>();

cols[i] = new HashSet<>();

boxes[i] = new HashSet<>();

}

*// Iterate through each cell in the 9x9 board*

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

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

char val = board[i][j];

*// Skip empty cells (denoted by '.')*

if (val == '.') {

continue;

}

*// Check row for duplicate digit*

if (rows[i].contains(val)) {

return false; *// Duplicate found in row*

}

rows[i].add(val);

*// Check column for duplicate digit*

if (cols[j].contains(val)) {

return false; *// Duplicate found in column*

}

cols[j].add(val);

*// Check 3x3 box for duplicate digit*

*// Box index calculated as (row/3)\*3 + (col/3)*

int idx = (i / 3) \* 3 + (j / 3);

if (boxes[idx].contains(val)) {

return false; *// Duplicate found in box*

}

boxes[idx].add(val);

}

}

*// If no duplicates are found, the board is valid*

return true;

}

}

**Time Complexity**

* **Time Complexity: O(1)**
  + The board is fixed at 9x9 (81 cells), so the nested loops iterate a constant number of times (81 iterations).
  + For each cell:
    - Checking and adding to HashSet (contains and add) are O(1) on average.
    - Other operations (e.g., calculating box index) are O(1).
  + Since the board size is fixed, the total time is O(81) = **O(1)**.
  + **Note**: If the board size were variable (e.g., NxN), the time complexity would be O(N²), but for standard Sudoku, N=9 is constant.

**Space Complexity**

* **Space Complexity: O(1)**
  + The code uses three arrays of HashSet<Character>, each of size 9 (for rows, columns, and boxes).
  + Each HashSet stores at most 9 digits (characters '1' to '9'), so each set uses O(1) space.
  + Total space: 3 arrays \* 9 sets \* O(1) per set = O(1).
  + The input board is not counted as extra space, as it’s provided.
  + **Note**: If the board size were variable (NxN), the space complexity would be O(N), but for standard Sudoku, it’s **O(1)**.

**Concise Summary of the Approach**

The solution checks if a 9x9 Sudoku board is valid by ensuring no digit ('1' to '9') is repeated in any row, column, or 3x3 box. It uses three arrays of HashSets to track digits in each row, column, and box. For each cell, it skips empty cells ('.') and checks if the current digit exists in the corresponding row, column, or box HashSet. If a duplicate is found, it returns false. Otherwise, it adds the digit to the appropriate sets. If all cells are processed without duplicates, it returns true. The approach achieves **O(1)** time and space complexity for a fixed 9x9 board, leveraging efficient HashSet operations.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(1) time and space for a fixed 9x9 board, optimal for the standard Sudoku problem.
  + **Clarity**: The use of HashSets for rows, columns, and boxes is intuitive and easy to explain.
  + **Robustness**: Handles all valid Sudoku digits and empty cells correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it without extra space?
    - **Response**: Use the board itself to mark duplicates (e.g., modify values temporarily), but this is complex and risks altering input. Alternatively, use bitsets (one bit per digit 1-9) instead of HashSets to reduce memory, still O(1) for 9x9. The current approach is practical and clear.
  + **Interviewer might ask**: How would you handle a variable-sized board (NxN)?
    - **Response**: The same logic applies, but time becomes O(N²) for iterating the board, and space becomes O(N) for storing N HashSets, each holding up to N digits.
  + **Interviewer might ask**: What if the board contains invalid characters?
    - **Response**: Add validation to check if val is between '1' and '9' (e.g., if (val < '1' || val > '9') return false;). The current code assumes valid digits per problem constraints.
* **Edge Cases Handled**:
  + Empty cells ('.'): Skipped correctly via the if (val == '.') check.
  + Single row/column/box: Valid if no duplicates (handled by HashSets).
  + Invalid board (duplicates): Returns false when a duplicate is found.
  + **Note**: Assumes the input is a 9x9 board with valid characters ('.' or '1' to '9'). Add checks for null or invalid dimensions if required.
* **Assumptions**:
  + The board is 9x9, and cells contain either '.' or digits '1' to '9'.
  + The input array is non-null and properly formatted.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Bitsets**: Use 9-bit bitsets instead of HashSets for each row, column, and box. Time: O(1), Space: O(1). Saves memory but is less readable.
  + **Boolean Arrays**: Use boolean arrays (size 10 for digits 1-9) instead of HashSets. Time: O(1), Space: O(1). Similar efficiency but slightly more explicit.
  + **Single Pass with Bit Manipulation**: Track digits using bit manipulation in a single array. More complex but minimizes memory. The HashSet approach is preferred for clarity and maintainability.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null board or invalid dimensions (e.g., if (board == null || board.length != 9 || board[0].length != 9) return false;) and invalid characters.
  + **Clarity**: Add comments explaining the box index calculation (e.g., “(i/3)\*3 + (j/3) maps to one of 9 boxes”).
  + **Optimization**: For a 9x9 board, consider boolean arrays (size 10) instead of HashSets to reduce overhead, though the impact is minimal for O(1) space.
  + **Edge Case Discussion**: In an interview, mention handling of invalid characters or board sizes to demonstrate thoroughness.
* **Note on Box Index Calculation**: The formula (i / 3) \* 3 + (j / 3) maps each cell to one of the 9 boxes (0 to 8). For example, cell (4,5) maps to box (4/3)\*3 + (5/3) = 1\*3 + 1 = 4. Explain this clearly in an interview if asked.
* Encode and Decode Strings

Commented Code

class Solution {

*// Encodes a list of strings into a single string*

public String encode(List<String> strs) {

StringBuilder sb = new StringBuilder();

*// Iterate through each string in the 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();

}

*// Decodes a single string back into a list of strings*

public List<String> decode(String s) {

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

int i = 0; *// Pointer to track current position in string*

*// Process the string until the end*

while (i < s.length()) {

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

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

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

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

*// Extract the string of specified length after the delimiter*

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

*// Move the pointer to the start of the next length (after the string)*

i = j + 1 + length;

}

*// Return the decoded list of strings*

return result;

}

}

**Time Complexity**

* **Encode: O(n)**
  + Iterates through the list of n strings once.
  + For each string of length k\_i, appending its length (O(1)), delimiter (O(1)), and the string itself (O(k\_i)) to the StringBuilder takes O(k\_i).
  + Total time for all strings is O(k\_1 + k\_2 + ... + k\_n) = O(K), where K is the total length of all strings. Since we iterate over n strings, we can approximate this as O(n) in terms of the number of strings, assuming average string length is bounded.
  + **Note**: StringBuilder append operations are amortized O(1) per character, so the total time is effectively O(K).
* **Decode: O(n)**
  + The while loop processes the input string of length K (total length of encoded string, including lengths and delimiters).
  + For each string:
    - indexOf('#', i) scans from i to find the delimiter, taking O(K) in the worst case, but typically O(k\_i) for each string due to the structured format.
    - substring(i, j) and Integer.parseInt take O(k\_i) to extract and parse the length.
    - substring(j + 1, j + 1 + length) takes O(k\_i) to extract the string.
    - Updating i is O(1).
  + Since the loop processes each character of the input string once across all iterations, the total time is O(K), or O(n) in terms of the number of strings, assuming bounded average string length.
  + **Note**: The indexOf operation’s worst-case cost is mitigated by the fact that the pointer i advances past each processed string, ensuring each character is visited a constant number of times.
* **Overall**: Both methods are O(n) in terms of the number of strings or O(K) in terms of total string length.

**Space Complexity**

* **Encode: O(K)**
  + The StringBuilder stores the encoded string, which includes each string’s length, a '#' delimiter, and the string itself. This is O(K), where K is the total length of all strings plus delimiters and lengths (proportional to K).
  + The output string is required and counted as part of the space complexity.
  + No additional data structures scale significantly, so space is **O(K)**.
* **Decode: O(K)**
  + The output ArrayList stores the decoded strings, which collectively have total length K, so O(K) for the output.
  + Temporary variables (e.g., substrings, parsed integers) use O(k\_i) space per iteration, where k\_i is the length of the current string, but these are reused and don’t accumulate.
  + The input string is not counted as extra space.
  + Total space is **O(K)** for the output list.
* **Overall**: Both methods use **O(K)** space, where K is the total length of all strings.

**Concise Summary of the Approach**

The solution encodes a list of strings into a single string by appending each string’s length, a '#' delimiter, and the string itself using a StringBuilder. To decode, it iterates through the encoded string, extracting each string’s length before the '#' delimiter, then retrieving the string of that length after the delimiter, adding it to the result list. The approach is efficient with **O(n)** time complexity (or O(K) for total string length) and **O(K)** space complexity for both encoding and decoding, effectively handling variable-length strings and special characters.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time (or O(K) for total string length) is optimal for processing each character once.
  + **Robustness**: The length-delimiter format handles empty strings, special characters, and variable lengths without ambiguity.
  + **Clarity**: The code is straightforward and easy to explain, making it interview-friendly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: What if the strings contain '#' or numbers?
    - **Response**: The length-delimiter format ensures unambiguous decoding, as the length before '#' is parsed as an integer, and the exact number of characters after '#' is extracted, regardless of content (e.g., '#' in the string is included in the length count).
  + **Interviewer might ask**: Can you use a different delimiter or format?
    - **Response**: Yes, any unique delimiter not confused with length digits (e.g., ':', '/'), or a fixed-length field for the length, could work. The current '#' is simple and effective. Alternatively, serialize lengths in binary format to save space, but this complicates parsing.
  + **Interviewer might ask**: How would you handle invalid input in decoding?
    - **Response**: Add validation for malformed strings (e.g., missing '#', invalid length format). For example, check if indexOf('#') returns -1 or if Integer.parseInt throws an exception.
* **Edge Cases Handled**:
  + Empty list of strings: Encode returns an empty string; decode returns an empty list.
  + Empty strings: Handled correctly (e.g., encoded as "0#").
  + Strings with special characters (e.g., '#'): Included in the length count, so decoded correctly.
  + Single string: Encoded and decoded as expected.
  + **Note**: Assumes the input string for decoding is well-formed (from a valid encode). Add error handling if required.
* **Assumptions**:
  + The input list for encoding is non-null, and strings may be empty or contain any characters.
  + The input string for decoding is a valid encoded string (e.g., from the encode method).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Delimiter with Escape Characters**: Use a delimiter and escape any occurrences of it in the strings (e.g., escape '#' with '#'). More complex and error-prone.
  + **Fixed-Length Prefix**: Use a fixed number of bytes for the length (e.g., 4 digits). Limits maximum string length but avoids parsing issues.
  + **Binary Encoding**: Serialize lengths and strings in binary format. Saves space but requires complex handling for cross-language compatibility.
  + The length-delimiter approach is preferred for simplicity and robustness.
* **Suggestions for Improvement**: –

System: It looks like your message was cut off. Would you like to continue with suggestions for improvement or any other specific details about the "Encode and Decode Strings" problem? Below, I’ll complete the analysis by providing suggestions for improvement and any additional interview preparation notes, assuming that’s what you intended. If you have a different request, please clarify!

**Suggestions for Improvement (Continued)**

* **Input Validation**:
  + For encode: Add a check for a null input list (e.g., if28if (strs == null) return "";).
  + For decode: Add validation for malformed input, such as checking if indexOf('#') returns -1 or catching exceptions from Integer.parseInt (e.g., try { length = Integer.parseInt(s.substring(i, j)); } catch (NumberFormatException e) { return new ArrayList<>(); }).
* **Clarity**: Add comments explaining the encoding format (e.g., “Format: length#string for each string”) and the decoding process (e.g., “Extract length before '#', then take substring of that length”).
* **Error Handling**: For decoding, handle edge cases like an empty string (if (s.isEmpty()) return new ArrayList<>();) or invalid formats by returning an empty list or throwing an exception, depending on problem requirements.
* **Optimization**: For very large strings, consider using a StringBuilder for substring operations in decode to avoid creating temporary strings, though the impact is minimal given the single-pass nature.
* **Robustness**: Ensure the decoding logic handles edge cases like consecutive '#' characters or non-numeric length prefixes by validating the format explicitly.

**Additional Interview Preparation Notes**

* **Edge Case Discussion**: In an interview, proactively mention edge cases like empty strings, special characters, or malformed encoded strings to demonstrate thoroughness. For example, explain how "0#" correctly encodes an empty string and how the length prefix prevents ambiguity with special characters like '#'.
* **Alternative Delimiters**: If asked about other delimiters, suggest options like ':' or ';', but note that '#' is effective due to its rarity in typical string inputs. Discuss trade-offs, such as needing escape characters for delimiters that appear in strings.
* **Performance Considerations**: Highlight that StringBuilder in encode ensures efficient string concatenation (O(K) vs. O(K²) for string addition). For decode, emphasize that the single-pass approach minimizes character scans, making it efficient.
* **Cross-Language Compatibility**: If the encoded string is used across languages (e.g., in a system design context), mention that the length-delimiter format is simple and portable, but binary encoding could be considered for space efficiency in specific scenarios.
* **Testing Edge Cases**: Suggest test cases like:
  + [""] → Encodes to "0#" → Decodes to [""].
  + ["hello#world"] → Encodes to "11#hello#world" → Decodes correctly.
  + ["", "abc", ""] → Encodes to "0#3#abc0#" → Decodes to ["", "abc", ""].
  + Invalid decode input like "abc#def" or "3#x" → Handle with error or empty list.

**Example Walkthrough for Interview**

To explain in an interview:

1. **Encode**: “For each string, I append its length, a '#' delimiter, and the string itself to a StringBuilder. This format ensures we can decode unambiguously, even with special characters.”
2. **Decode**: “I scan the string, finding each '#' to extract the length, parse it to an integer, then take the substring of that length after the '#'. I update the pointer to skip the processed string and repeat until done.”
3. **Complexity**: “Both methods are O(K) for total string length K, or O(n) for n strings with bounded length. Space is O(K) for the output string or list.”
4. **Edge Cases**: “Empty strings encode as '0#', special characters are included in the length, and invalid decode inputs can be handled with validation.”

* Longest Consecutive Sequence

Commented Code

class Solution {

*// Method to find the length of the longest consecutive sequence in an array*

public int longestConsecutive(int[] nums) {

*// If the array is empty, return 0*

if (nums.length == 0) {

return 0;

}

*// Initialize a HashSet to store all numbers for O(1) lookup*

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

*// Add all numbers to the HashSet*

for (int num : nums) {

numSet.add(num);

}

int longest = 0; *// Track the longest sequence found*

*// Iterate through each number in the HashSet*

for (int num : numSet) {

*// Only process numbers that start a sequence (no num-1 exists)*

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

int length = 1; *// Start counting the sequence length*

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

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

length++;

}

*// Update the longest sequence if current is longer*

longest = Math.max(longest, length);

}

}

*// Return the length of the longest consecutive sequence*

return longest;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + Building the HashSet: Iterating through the array of length n and adding each number to the HashSet (O(1) per insertion on average) takes O(n).
  + Main loop: Iterates through each number in the HashSet (at most n unique numbers).
    - For each number, checking numSet.contains(num - 1) is O(1) on average.
    - If the number starts a sequence (num - 1 is not present), the while loop checks for consecutive numbers (num + 1, num + 2, ...).
    - Each number in the array is checked at most twice: once as a potential sequence start (in the for loop) and once as part of a sequence (in the while loop). Since contains is O(1), the total work for all sequence checks is O(n).
  + Overall, the time complexity is O(n) for building the HashSet + O(n) for processing sequences = **O(n)**.
  + **Note**: The HashSet operations are O(1) on average, but in rare cases of hash collisions, they could degrade. This is unlikely with a good hash function.

**Space Complexity**

* **Space Complexity: O(n)**
  + The HashSet stores at most n unique numbers from the input array, requiring O(n) space.
  + Other variables (longest, length, loop counters) use O(1) space.
  + The total space complexity is **O(n)**.

**Concise Summary of the Approach**

The solution finds the length of the longest consecutive sequence in an array by using a HashSet for O(1) lookups. It first adds all numbers to the HashSet. Then, for each number, it checks if it’s the start of a sequence (i.e., num - 1 is not present). If so, it counts consecutive numbers (num + 1, num + 2, ...) using the HashSet and tracks the longest sequence found. The approach achieves **O(n)** time complexity and **O(n)** space complexity, efficiently handling duplicates and unsorted input.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each number is processed a constant number of times, leveraging HashSet for fast lookups.
  + **Clarity**: The logic is intuitive—start sequences only at the smallest number and count consecutive numbers—making it easy to explain.
  + **Robustness**: Handles duplicates (via HashSet), empty arrays, and negative numbers correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it with O(1) space?
    - **Response**: Sorting the array (O(n log n) time, O(1) space if in-place) and iterating to find the longest consecutive sequence is possible, but it’s less efficient in time. The HashSet approach is preferred for O(n) time unless space is a strict constraint.
  + **Interviewer might ask**: How does this handle duplicates?
    - **Response**: Duplicates are automatically handled by the HashSet, which stores each number once, ensuring sequences are counted correctly without redundant checks.
  + **Interviewer might ask**: What if the array is very large with few unique elements?
    - **Response**: The HashSet size is bounded by the number of unique elements, so if there are few unique elements, space and time are reduced (still O(n) worst-case). The approach remains efficient.
* **Edge Cases Handled**:
  + Empty array: Returns 0 (explicitly checked).
  + Single element: Returns 1 (sequence of length 1).
  + Duplicates: Handled by HashSet, which stores unique numbers.
  + No consecutive sequences: Returns 1 (each number is its own sequence).
  + Negative numbers: Handled correctly, as the HashSet and comparisons work for all integers.
  + **Note**: Add a null check (e.g., if (nums == null) return 0;) if required by the problem.
* **Assumptions**:
  + The input array is non-null.
  + Numbers can be positive, negative, or zero, and duplicates are possible.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Sorting-Based**: Sort the array and iterate to find the longest sequence of consecutive numbers. Time: O(n log n), Space: O(1) if in-place sorting is allowed. Less efficient but saves space.
  + **Union-Find**: Treat numbers as nodes and connect consecutive numbers. Time: O(n) with path compression, Space: O(n). More complex and not typically necessary.
  + The HashSet approach is preferred for its balance of simplicity, efficiency, and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add a check for a null array (e.g., if (nums == null) return 0;) to handle invalid input.
  + **Clarity**: Add comments explaining why only sequence starts are checked (e.g., “Skip numbers with num-1 to avoid redundant sequence checks”).
  + **Optimization**: The current approach is near-optimal, but mention in an interview that checking only sequence starts avoids redundant work, ensuring O(n) time.
  + **Edge Case Discussion**: Proactively mention handling of duplicates, negative numbers, and empty arrays to show thoroughness.
* **Note on Sequence Start Check**: The condition !numSet.contains(num - 1) ensures we only process the start of each sequence, avoiding redundant checks for numbers in the middle of a sequence. This is key to maintaining O(n) time, as each number is visited at most twice (once in the for loop, once in the while loop).

2. Two Pointers (5 problems)

* Valid Palindrome

Commented Code

class Solution {

*// Method to check if a string is a palindrome, ignoring non-alphanumeric characters*

public boolean isPalindrome(String s) {

int left = 0, right = s.length() - 1; *// Initialize two pointers: left at start, right at end*

*// Continue until pointers meet*

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)*

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

return false; *// Return false if characters don't match*

}

*// Move pointers inward*

left++;

right--;

}

*// Return true if all characters match (or string is empty)*

return true;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The two pointers (left and right) traverse the string of length n toward the center, visiting each character at most once.
  + The inner while loops skip non-alphanumeric characters, and in the worst case (e.g., all characters are non-alphanumeric), each character is checked exactly once.
  + The operations Character.isLetterOrDigit and Character.toLowerCase are O(1).
  + The total work is proportional to the string length, resulting in **O(n)** time complexity, where n is the length of the string.
  + **Note**: Each character is visited at most a constant number of times (once by each pointer or during skipping), ensuring linear time.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses only two integer variables (left and right) and no additional data structures.
  + The input string is not modified, and no extra space scales with input size.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution checks if a string is a palindrome by ignoring non-alphanumeric characters and considering case-insensitive comparison. It uses two pointers (left and right) starting from the string’s ends. Each pointer skips non-alphanumeric characters, and the characters at the pointers are compared (converted to lowercase). If any pair doesn’t match, it returns false. If the pointers meet or cross, it returns true. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently handling special characters and case sensitivity.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time and O(1) space make it optimal, as it processes each character a constant number of times without extra storage.
  + **Clarity**: The two-pointer technique is intuitive and easy to explain, making it ideal for interviews.
  + **Robustness**: Handles all edge cases (empty strings, non-alphanumeric characters, mixed case) correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it by preprocessing the string?
    - **Response**: Yes, convert the string to lowercase and filter out non-alphanumeric characters into a new string, then check if it’s a palindrome. Time: O(n), Space: O(n). The two-pointer approach is better for space efficiency.
  + **Interviewer might ask**: How does this handle Unicode or special characters?
    - **Response**: Character.isLetterOrDigit works for ASCII letters and digits. For Unicode, it supports a broader range, but if specific character sets are required, clarify with the interviewer or use a custom check (e.g., regex or specific Unicode ranges).
  + **Interviewer might ask**: What if we only consider letters (not digits)?
    - **Response**: Modify the condition to use Character.isLetter instead of Character.isLetterOrDigit. The logic remains the same.
* **Edge Cases Handled**:
  + Empty string: Returns true (empty string is a palindrome).
  + Single character: Returns true (single character is a palindrome).
  + Non-alphanumeric characters only (e.g., “,,,”): Returns true (all skipped, effectively empty).
  + Mixed case (e.g., “RaCeCaR”): Handled by Character.toLowerCase.
  + Spaces and punctuation (e.g., “A man, a plan, a canal: Panama”): Skipped correctly.
  + **Note**: Assumes the input is non-null. Add a null check if required.
* **Assumptions**:
  + The input string is non-null.
  + Only alphanumeric characters (letters and digits) are considered, and comparison is case-insensitive.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Preprocessing**: Convert the string to lowercase and filter out non-alphanumeric characters into a new string or array, then use two pointers or reverse-and-compare. Time: O(n), Space: O(n). Less space-efficient but simpler to read.
  + **Reverse String**: Build a cleaned string (alphanumeric, lowercase) and compare it with its reverse. Time: O(n), Space: O(n). Similar to preprocessing but more explicit.
  + The two-pointer approach is preferred for its O(1) space efficiency and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add a check for null input (e.g., if (s == null) return true;) if not guaranteed by the problem.
  + **Clarity**: Add comments explaining the two-pointer logic (e.g., “Skip non-alphanumeric characters and compare case-insensitive”).
  + **Optimization**: The current approach is optimal, but mention in an interview that checking left < right in the inner loops prevents unnecessary checks when pointers meet.
  + **Edge Case Discussion**: Proactively mention handling of spaces, punctuation, and case sensitivity to show thoroughness.
* **Note on Character Methods**: The use of Character.isLetterOrDigit and Character.toLowerCase ensures robust handling of ASCII (and some Unicode) characters. If the problem specifies a different character set, discuss with the interviewer to adjust the validation logic.
* Two Sum II Input Array Is Sorted

Commented Code

class Solution {

*// Method to find two indices in a sorted array whose elements sum to target*

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

int left = 0, right = numbers.length - 1; *// Initialize two pointers: left at start, right at end*

*// Continue until pointers meet*

while (left < right) {

int currSum = numbers[left] + numbers[right]; *// Calculate sum of elements at pointers*

if (currSum == target) {

*// If sum equals target, return 1-based indices*

return new int[] {left + 1, right + 1};

} else if (currSum > target) {

*// If sum is too large, move right pointer left (since array is sorted)*

right--;

} else {

*// If sum is too small, move left pointer right*

left++;

}

}

*// Return empty array if no solution is found*

return new int[] {};

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The two pointers (left and right) traverse the array of length n toward each other, visiting each element at most once.
  + Each iteration performs O(1) operations: addition, comparison, and pointer updates.
  + In the worst case, the pointers traverse the entire array (e.g., no solution or solution at the ends), resulting in O(n) iterations.
  + Therefore, the total time complexity is **O(n)**, where n is the length of the input array.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses only two integer variables (left and right) and a fixed-size output array (2 elements).
  + No additional data structures scale with input size, so the space complexity is **O(1)** (excluding the output array).

**Concise Summary of the Approach**

The solution finds two indices in a sorted array whose elements sum to a target using a two-pointer technique. It initializes pointers at the array’s start (left) and end (right). Since the array is sorted, it compares the sum of elements at the pointers to the target: if equal, it returns their 1-based indices; if too large, it moves the right pointer left; if too small, it moves the left pointer right. If no solution is found, it returns an empty array. The approach achieves **O(n)** time complexity and **O(1)** space complexity, leveraging the sorted property for efficiency.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time and O(1) space make it optimal, taking advantage of the sorted array to avoid extra data structures like a HashMap.
  + **Clarity**: The two-pointer technique is intuitive and easy to explain, ideal for interviews.
  + **Correctness**: Returns 1-based indices as required (e.g., per LeetCode’s "Two Sum II" problem) and handles the guaranteed single solution.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why not use a HashMap like in the original Two Sum problem?
    - **Response**: Since the array is sorted, the two-pointer approach is more space-efficient (O(1) vs. O(n) for HashMap) and still achieves O(n) time, making it preferable for this problem.
  + **Interviewer might ask**: What if the array isn’t sorted?
    - **Response**: If unsorted, we could sort the array (O(n log n)) and apply the two-pointer approach, but we’d need to track original indices (e.g., using a pair class), increasing space to O(n). Alternatively, use the HashMap approach from the original Two Sum (O(n) time, O(n) space).
  + **Interviewer might ask**: How does this handle duplicates or negative numbers?
    - **Response**: The two-pointer approach works with duplicates and negative numbers since it relies on the sorted order and sum comparison. The problem guarantees exactly one solution, so duplicates don’t affect correctness.
* **Edge Cases Handled**:
  + Empty array or fewer than 2 elements: Returns empty array (assumes not possible per problem constraints).
  + No solution: Returns empty array (though problem typically guarantees one solution).
  + Duplicates: Handled correctly, as the two-pointer approach finds the first valid pair in sorted order.
  + Negative numbers: Handled seamlessly, as addition and comparison work for all integers.
  + **Note**: Add checks for null or invalid input if required (e.g., if (numbers == null || numbers.length < 2) return new int[] {};).
* **Assumptions**:
  + The input array is non-null, has at least 2 elements, and is sorted in non-decreasing order.
  + Exactly one valid solution exists, and indices are 1-based in the output.
  + The sum of any two elements fits within a 32-bit integer.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **HashMap-Based**: Use a HashMap to store number-to-index mappings, as in the original Two Sum. Time: O(n), Space: O(n). Less efficient in space but works for unsorted arrays.
  + **Binary Search**: For each element numbers[i], binary search for target - numbers[i] in the remaining array. Time: O(n log n), Space: O(1). Less efficient than two-pointer but viable.
  + The two-pointer approach is preferred for its simplicity and optimal time/space complexity given the sorted input.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or small arrays (e.g., if (numbers == null || numbers.length < 2) return new int[] {};) if not guaranteed by the problem.
  + **Clarity**: Add comments explaining the two-pointer logic (e.g., “Use sorted property to adjust pointers based on sum comparison”).
  + **Edge Case Discussion**: In an interview, mention handling of duplicates, negative numbers, and the 1-based index requirement to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that the sorted property eliminates the need for extra space, distinguishing it from the unsorted Two Sum problem.
* **Note on 1-Based Indices**: The problem requires 1-based indices in the output (e.g., left + 1, right + 1), which is correctly implemented. Clar Filled to clarify this with the interviewer if the problem’s requirements differ.
* 3Sum

Commented Code

class Solution {

*// Method to find all unique triplets in the array that sum to zero*

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

Arrays.sort(nums); *// Sort the array to enable two-pointer technique and handle duplicates*

List<List<Integer>> result = new ArrayList<>(); *// Initialize result list for triplets*

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

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

*// Skip duplicates for the first element to avoid duplicate triplets*

if (i == 0 || nums[i] != nums[i - 1]) {

twoSum2(nums, i, result); *// Find pairs that sum to -nums[i]*

}

}

return result;

}

*// Helper method to find two numbers that sum to -nums[i] using two pointers*

private void twoSum2(int[] nums, int i, List<List<Integer>> result) {

int left = i + 1; *// Start of the search range (after i)*

int right = nums.length - 1; *// End of the search range*

*// Use two pointers to find pairs*

while (left < right) {

int sum = nums[i] + nums[left] + nums[right]; *// Calculate sum of triplet*

if (sum < 0) {

left++; *// Sum too small, increment left pointer*

} else if (sum > 0) {

right--; *// Sum too large, decrement right pointer*

} else».

*// Found a valid triplet*

result.add(Arrays.asList(nums[i], nums[left], nums[right]));

*// Skip duplicates for left pointer to avoid duplicate triplets*

while (left < right && nums[left] == nums[left + 1]) left++;

*// Skip duplicates for right pointer to avoid duplicate triplets*

while (left < right && nums[right] == nums[right - 1]) right--;

*// Move pointers inward to check next potential pair*

left++;

right--;

}

}

}

}

**Time Complexity**

* **Time Complexity: O(n²)**
  + **Sorting**: The initial Arrays.sort(nums) takes O(n log n), where n is the length of the input array.
  + **Outer Loop**: Iter彼此

System: It looks like your message was cut off. Below, I’ll complete the analysis for the "3Sum" problem, assuming you intended to share the full code (which appears complete). I’ll provide the remaining time complexity details, space complexity, and other analysis components.

**Time Complexity (Continued)**

* **Outer Loop**: The for loop iterates up to n times, but the condition nums[i] <= 0 limits iterations to indices where the first element is non-positive (since positive numbers cannot contribute to a sum of zero with two other numbers in a sorted array). In the worst case, this is still O(n) iterations.
* **Inner Loop (twoSum2)**: For each fixed i, the twoSum2 method uses a two-pointer technique, which takes O(n) time to traverse the remaining elements (left to right). Since the outer loop runs O(n) times, the total time for the two-pointer searches is O(n \* n) = O(n²).
* **Duplicate Handling**: Checking and skipping duplicates (nums[i] != nums[i - 1], nums[left] == nums[left + 1], nums[right] == nums[right - 1]) is O(1) per check, and these checks don’t significantly affect the overall complexity.
* **Overall**: The dominant term is O(n²) from the nested loop structure, with the initial sorting (O(n log n)) being less significant for large n. Thus, the total time complexity is **O(n²)**.

**Space Complexity**

* **Space Complexity: O(n)**
  + The result list stores all unique triplets. In the worst case (e.g., many triplets summing to zero), the number of triplets is O(n²), but each triplet is of constant size (3 integers), so the space for the output is O(n²). However, per standard problem constraints (e.g., LeetCode), the output space is not counted as extra space.
  + The sorting operation (Arrays.sort) typically uses O(n) space in (for the internal merge sort or temporary arrays).
  + Other variables (left, right, i, sum) use O(1) space.
  + Excluding the output, the extra space is dominated by the sorting, resulting in **O(n)** space complexity.
  + **Note**: If the output space is considered (as it might be in some interview contexts), the total space could be O(n²) in pathological cases with many triplets.

**Concise Summary of the Approach**

The solution finds all unique triplets in an array that sum to zero by first sorting the array to enable a two-pointer technique and simplify duplicate handling. It iterates through each element as the first number of the triplet, skipping duplicates and stopping at positive numbers (since they can’t form a zero-sum triplet). For each fixed first number, it uses two pointers to find a pair of numbers that sum to the negation of the first number, adding valid triplets to the result and skipping duplicates to ensure uniqueness. The approach achieves **O(n²)** time complexity and **O(n)** space complexity (excluding output), efficiently handling duplicates and leveraging the sorted array.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n²) time is optimal for this problem, as it requires checking all possible pairs for each fixed element, and sorting reduces the pair-finding complexity to O(n).
  + **Clarity**: The sorting plus two-pointer technique is intuitive and widely used, making it easy to explain in an interview.
  + **Correctness**: Robustly handles duplicates and ensures unique triplets by skipping identical elements.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it without sorting?
    - **Response**: Without sorting, a hash-based approach could store sums of pairs in a HashMap to find triplets, but it’s still O(n²) time and O(n²) space due to storing pairs. Sorting is preferred for simplicity and guaranteed O(n) space (excluding output).
  + **Interviewer might ask**: Why stop at nums[i] <= 0?
    - **Response**: Since the array is sorted, if nums[i] > 0, the sum with two other numbers (which are ≥ nums[i] due to sorting) cannot be zero, so we can skip those iterations.
  + **Interviewer might ask**: How does this handle duplicates?
    - **Response**: The code skips duplicates for the first element (nums[i] != nums[i - 1]) and for the two-pointer elements (nums[left] == nums[left + 1], nums[right] == nums[right - 1]) to ensure only unique triplets are added.
* **Edge Cases Handled**:
  + Empty array or fewer than 3 elements: Returns empty list (implicitly, as the loop won’t produce results).
  + No triplets summing to zero: Returns empty list.
  + Duplicates: Skipped to ensure unique triplets.
  + All zeros (e.g., [0, 0, 0]): Correctly finds [0, 0, 0] if present.
  + Negative and positive numbers: Handled due to sorting and two-pointer logic.
  + **Note**: Add checks for null or small arrays if required (e.g., if (nums == null || nums.length < 3) return new ArrayList<>();).
* **Assumptions**:
  + The input array is non-null and contains integers.
  + Triplets must be unique, and the output format is a list of lists of integers.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **HashMap-Based**: For each i, use a HashMap to find pairs summing to -nums[i]. Time: O(n²), Space: O(n). Less efficient in space and more complex due to duplicate handling.
  + **Brute Force**: Check all triplets using three nested loops. Time: O(n³), Space: O(1) (excluding output). Highly inefficient and not practical.
  + The sorting plus two-pointer approach is preferred for its balance of efficiency and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or insufficient length (e.g., if (nums == null || nums.length < 3) return new ArrayList<>();) to handle edge cases explicitly.
  + **Clarity**: Add comments explaining the role of sorting (e.g., “Sort to enable two-pointer technique and handle duplicates”) and the two-pointer logic.
  + **Optimization**: The current approach is near-optimal, but mention that nums[i] <= 0 optimizes by skipping positive first elements, reducing unnecessary iterations.
  + **Edge Case Discussion**: Proactively mention handling of duplicates, empty arrays, and the role of sorting to demonstrate thoroughness.
* **Note on Duplicate Handling**: The checks for nums[i] != nums[i - 1], nums[left] == nums[left + 1], and nums[right] == nums[right - 1] ensure no duplicate triplets are added, which is critical for correctness and efficiency.
* Container With Most Water

Commented Code

class Solution {

*// Method to find the maximum area of water that can be trapped between two lines*

public int maxArea(int[] heights) {

int left = 0, right = heights.length - 1; *// Initialize two pointers: left at start, right at end*

int maxA = 0; *// Track the maximum area found*

*// Continue until pointers meet*

while (left < right) {

*// Calculate area: minimum height \* width (distance between pointers)*

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

maxA = Math.max(maxA, area); *// Update maximum area if current is larger*

*// Move the pointer corresponding to the smaller height inward*

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

right--; *// If right height is smaller, move right pointer left*

} else {

left++; *// If left height is smaller or equal, move left pointer right*

}

}

*// Return the maximum area found*

return maxA;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The two pointers (left and right) traverse the array of length n toward each other, visiting each element at most once.
  + Each iteration performs O(1) operations: calculating the area (Math.min and multiplication), updating maxA (Math.max), and moving a pointer.
  + In the worst case, the pointers traverse the entire array (e.g., when heights are in a pattern that requires checking most pairs), resulting in O(n) iterations.
  + Therefore, the total time complexity is **O(n)**, where n is the length of the input array.

**Space Complexity**

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

**Concise Summary of the Approach**

The solution finds the maximum area of water that can be trapped between two lines in an array of heights using a two-pointer technique. It initializes pointers at the array’s start (left) and end (right). In each iteration, it calculates the area as the minimum height at the pointers times the distance between them, updating the maximum area if necessary. The pointer at the smaller height is moved inward to potentially increase the area. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently leveraging the two-pointer method to avoid checking all pairs.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as it checks only necessary pairs by moving pointers based on the limiting height, avoiding the O(n²) brute-force approach.
  + **Clarity**: The two-pointer technique is intuitive and easy to explain, making it ideal for interviews.
  + **Correctness**: Correctly maximizes the area by always exploring potentially larger areas while skipping redundant checks.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why move the pointer with the smaller height?
    - **Response**: The area is limited by the smaller height (since water spills over the shorter line). Moving the pointer with the smaller height might find a taller line, potentially increasing the area, while moving the taller pointer would only reduce the width and cannot increase the area (since the height is still capped by the smaller value).
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: A brute-force approach checking all pairs of indices takes O(n²) time, which is inefficient. The two-pointer method is optimal, as it balances width and height considerations in linear time.
  + **Interviewer might ask**: How does this handle edge cases like equal heights?
    - **Response**: When heights are equal, moving either pointer works (the code moves left). The choice doesn’t affect correctness, as both pointers will eventually explore all necessary pairs.
* **Edge Cases Handled**:
  + Array with two elements: Computes the area between them and returns it (correctly handled).
  + Empty array or single element: Not applicable (problem assumes at least two elements).
  + All heights equal: Correctly computes areas, moving left inward (per the else condition).
  + Heights with zeros: Handled correctly, as zero heights produce zero or small areas.
  + **Note**: Add checks for null or insufficient length if required (e.g., if (heights == null || heights.length < 2) return 0;).
* **Assumptions**:
  + The input array is non-null and has at least two elements.
  + Heights are non-negative integers.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Brute Force**: Check all pairs of indices and compute areas. Time: O(n²), Space: O(1). Highly inefficient and not practical for large inputs.
  + **Greedy with Preprocessing**: No viable preprocessing approach exists that improves on the two-pointer method without sacrificing time or space efficiency.
  + The two-pointer approach is preferred for its optimal time and space complexity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or small arrays (e.g., if (heights == null || heights.length < 2) return 0;) if not guaranteed by the problem.
  + **Clarity**: Add comments explaining why the smaller height’s pointer is moved (e.g., “Move smaller height’s pointer to potentially increase area”).
  + **Edge Case Discussion**: In an interview, mention handling of zeros, equal heights, and the rationale for moving the smaller height’s pointer to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that the two-pointer method minimizes redundant checks by leveraging the area’s dependency on the minimum height.
* **Note on Moving Smaller Height**: The decision to move the pointer with the smaller height (if (heights[left] > heights[right])) ensures that we don’t miss potentially larger areas. If we moved the taller pointer, the width would decrease, and the area would be limited by the same or a smaller height, guaranteeing no improvement.
* Trapping Rain Water

Commented Code

public class Solution {

*// Method to calculate the total water trapped between bars of given heights*

public int trap(int[] height) {

*// Handle edge cases: null or empty array*

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

return 0;

}

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

int l = 0, r = height.length - 1;

*// Track maximum height seen from left and right*

int leftMax = height[l], rightMax = height[r];

int res = 0; *// Track total water trapped*

*// Continue until pointers meet*

while (l < r) {

*// If leftMax is smaller, process left pointer*

if (leftMax < rightMax) {

l++; *// Move left pointer right*

leftMax = Math.max(leftMax, height[l]); *// Update max height from left*

res += leftMax - height[l]; *// Add trapped water at current position*

} else {

*// If rightMax is smaller or equal, process right pointer*

r--; *// Move right pointer left*

rightMax = Math.max(rightMax, height[r]); *// Update max height from right*

res += rightMax - height[r]; *// Add trapped water at current position*

}

}

*// Return total water trapped*

return res;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The two pointers (l and r) traverse the array of length n toward each other, visiting each element at most once.
  + Each iteration performs O(1) operations: comparison (leftMax < rightMax), updating leftMax or rightMax (Math.max), and adding to res.
  + In the worst case, the pointers traverse the entire array (e.g., when all heights are equal), resulting in O(n) iterations.
  + Therefore, the total time complexity is **O(n)**, where n is the length of the input array.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses only a few variables (l, r, leftMax, rightMax, res) regardless of input size.
  + No additional data structures are used, so the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution calculates the water trapped between bars in an array of heights using a two-pointer technique. It initializes pointers at the array’s start (l) and end (r), tracking the maximum height seen from the left (leftMax) and right (rightMax). For each position, water trapped is the difference between the minimum of leftMax and rightMax and the current height. The pointer with the smaller max height is moved inward, updating the respective max height and accumulating water. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently leveraging the two-pointer method to avoid extra space.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time and O(1) space make it optimal, avoiding the need for extra arrays or preprocessing.
  + **Clarity**: The two-pointer technique is intuitive and easy to explain, especially with the logic of processing the smaller side first.
  + **Correctness**: Correctly computes water trapped by ensuring each position is bounded by the minimum of the maximum heights on both sides.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why process the side with the smaller max height?
    - **Response**: Water trapped at a position is limited by the smaller of the maximum heights on its left and right. By moving the pointer with the smaller max height, we ensure we’re computing water based on the limiting factor, and the other side’s max height guarantees correctness for the current position.
  + **Interviewer might ask**: Can you solve it with extra space?
    - **Response**: Yes, use two arrays to precompute the maximum height to the left and right of each position. Then, for each index, water is min(leftMax[i], rightMax[i]) - height[i]. Time: O(n), Space: O(n). The two-pointer approach is preferred for O(1) space.
  + **Interviewer might ask**: How does this handle zeros or equal heights?
    - **Response**: Zeros are handled correctly (no water if no boundaries). Equal heights are processed by moving the right pointer (per the else clause), but the logic works regardless of which pointer moves in such cases.
* **Edge Cases Handled**:
  + Null or empty array: Returns 0 (explicitly checked).
  + Single element or two elements: Returns 0 (no water can be trapped, handled implicitly as l < r fails).
  + All heights equal: No water trapped, as leftMax - height[l] or rightMax - height[r] is 0.
  + Zeros in array: Handled correctly, as water depends on bounding heights.
  + **Note**: The code already checks for null or empty arrays, which is robust.
* **Assumptions**:
  + The input array contains non-negative integers.
  + If these assumptions don’t hold (e.g., negative heights), clarify with the interviewer, though negative heights are typically invalid for this problem.
* **Alternative Approaches**:
  + **Two Arrays**: Precompute arrays for maximum heights to the left and right of each index, then compute water at each position. Time: O(n), Space: O(n). Simpler but less space-efficient.
  + **Brute Force**: For each index, find the maximum heights on both sides by scanning the array. Time: O(n²), Space: O(1). Highly inefficient.
  + The two-pointer approach is preferred for its optimal time and space complexity.
* **Suggestions for Improvement**:
  + **Clarity**: Add comments explaining the water calculation (e.g., “Water at each position is min(leftMax, rightMax) - height”). The current code is clear but could benefit from this.
  + **Edge Case Discussion**: In an interview, mention handling of zeros, equal heights, and the rationale for moving the smaller max height’s pointer to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that processing the smaller max height ensures correctness without needing to precompute maxima.
  + **Validation**: The null/empty check is sufficient, but could add an explicit check for height.length < 3 (returns 0, as no water can be trapped) for clarity.
* **Note on Water Calculation**: Water trapped at index i is determined by min(leftMax, rightMax) - height[i]. The two-pointer approach ensures that when processing a position, the smaller of leftMax or rightMax is the limiting factor, and the other side’s max height is sufficient to bound the water.

3. Sliding Window (6 problems)

* Best Time to Buy And Sell Stock

Commented Code

class Solution {

*// Method to find the maximum profit from buying and selling a stock once*

public int maxProfit(int[] prices) {

int left = 0, right = 1; *// Initialize two pointers: left (buy), right (sell)*

int maxP = 0; *// Track the maximum profit*

*// Iterate while right pointer is within bounds*

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]; *// Current profit*

maxP = Math.max(maxP, profit); *// Update max profit if higher*

} else {

*// If selling price is lower or equal, update buy point to current price*

left = right;

}

right++; *// Move sell pointer forward*

}

*// Return the maximum profit (0 if no profit possible)*

return maxP;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The right pointer iterates through the array of length n exactly once.
  + The left pointer only moves forward (never backward) when prices[right] <= prices[left], so each element is visited at most twice (once by right, and potentially once by left when it catches up).
  + Each iteration performs O(1) operations: comparison, subtraction, and updating maxP or pointers.
  + Therefore, the total time complexity is **O(n)**, where n is the length of the input array.

**Space Complexity**

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

**Concise Summary of the Approach**

The solution finds the maximum profit from buying and selling a stock once using a two-pointer technique. It initializes left (buy) at the start and right (sell) at the next index. For each right position, if the price is higher than at left, it calculates the profit and updates the maximum profit. If the price at right is lower or equal, it updates left to right (new potential buy point). The right pointer always moves forward. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently tracking the best buy-sell opportunity.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time and O(1) space make it optimal, processing each price once while maintaining minimal state.
  + **Clarity**: The two-pointer technique (or sliding window) is intuitive, representing buy and sell points, and is easy to explain.
  + **Correctness**: Correctly handles cases where no profit is possible by returning 0.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why move left to right when prices[right] <= prices[left]?
    - **Response**: If the price at right is lower than or equal to the price at left, buying at right is better for future profits, as it offers a lower buy price. This ensures we always consider the minimum price seen so far as the potential buy point.
  + **Interviewer might ask**: Can you solve it with a single variable?
    - **Response**: Yes, track only the minimum price seen so far and the maximum profit, updating both as you iterate. This is equivalent to the two-pointer approach but uses minPrice instead of left. Time: O(n), Space: O(1). (See alternative approach below.)
  + **Interviewer might ask**: How does this handle decreasing prices?
    - **Response**: If prices are always decreasing, left moves to right each time prices[right] <= prices[left], resulting in maxP remaining 0, which is correct since no profit is possible.
* **Edge Cases Handled**:
  + Empty array or single element: Returns 0 (no profit possible, handled implicitly as right won’t iterate).
  + Decreasing prices (e.g., [7,6,4,3,1]): Returns 0, as left keeps moving to right, and no positive profit is found.
  + Increasing prices (e.g., [1,2,3,4,5]): Correctly finds max profit (e.g., 4, buying at 1, selling at 5).
  + Equal prices: Handled by moving left to right, ensuring no negative or zero profit affects maxP.
  + **Note**: Add checks for null or small arrays if required (e.g., if (prices == null || prices.length < 2) return 0;).
* **Assumptions**:
  + The input array is non-null and contains non-negative integers.
  + At least one price is provided (though the code handles single-element arrays implicitly).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Single-Pass with Minimum Price**: Track the minimum price seen so far and the maximum profit. For each price, update minPrice = Math.min(minPrice, prices[i]) and maxProfit = Math.max(maxProfit, prices[i] - minPrice). Time: O(n), Space: O(1). Functionally equivalent but simpler to code.
  + **Brute Force**: Check all possible buy-sell pairs. Time: O(n²), Space: O(1). Highly inefficient and not practical.
  + The two-pointer (or single-pass minimum price) approach is preferred for its optimal efficiency and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or insufficient length (e.g., if (prices == null || prices.length < 2) return 0;) to handle edge cases explicitly.
  + **Clarity**: Add comments explaining the buy-sell logic (e.g., “Move left to right when price[right] <= price[left] to find a better buy price”).
  + **Optimization**: The current approach is optimal, but consider the single-pass minimum price variant for even cleaner code (as shown above).
  + **Edge Case Discussion**: In an interview, mention handling of decreasing prices, single-element arrays, and the logic for moving left to show thoroughness.
* **Note on Two-Pointer Logic**: The left pointer represents the best buy point so far (lowest price), and moving it to right when prices[right] <= prices[left] ensures we always use the lowest possible buy price for future calculations. This is equivalent to tracking the minimum price in the alternative approach.
* Longest Substring Without Repeating Characters

Commented Code

class Solution {

*// Method to find the length of the longest substring without repeating characters*

public int lengthOfLongestSubstring(String s) {

*// Handle edge cases: null or empty string*

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

return 0;

}

*// Handle single-character string*

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

return 1;

}

int left = 0, right = 0; *// Initialize two pointers for sliding window*

int ans = 0; *// Track the length of the longest valid substring*

HashSet<Character> set = new HashSet<>(); *// Store characters in current window*

*// Iterate with right pointer*

while (right < s.length()) {

char c = s.charAt(right); *// Current character*

*// Shrink window from left until duplicate is removed*

while (set.contains(c)) {

set.remove(s.charAt(left)); *// Remove leftmost character*

left++; *// Move left pointer*

}

set.add(c); *// Add current character to set*

ans = Math.max(ans, right - left + 1); *// Update max length*

right++; *// Move right pointer*

}

*// Return the length of the longest substring without repeating characters*

return ans;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The right pointer iterates through the string of length n exactly once.
  + The left pointer only moves forward (never backward) when a duplicate is found, and each character is removed from the HashSet at most once.
  + Each character is processed at most twice: once when added by right and once when removed by left.
  + HashSet operations (contains, add, remove) are O(1) on average.
  + Therefore, the total time complexity is **O(n)**, where n is the length of the input string.
  + **Note**: In rare cases of hash collisions, HashSet operations could degrade, but this is unlikely with a good hash function.

**Space Complexity**

* **Space Complexity: O(min(m, n))**
  + The HashSet stores characters in the current sliding window. In the worst case, it contains all unique characters in the window.
  + The size of the HashSet is bounded by the smaller of:
    - n (the string length, if all characters are unique).
    - m (the size of the character set, e.g., 128 for ASCII, 256 for extended ASCII, or more for Unicode).
  + Other variables (left, right, ans, c) use O(1) space.
  + Thus, the space complexity is **O(min(m, n))**, where m is the size of the character set and n is the string length. For practical purposes (e.g., ASCII), this is often considered O(1).

**Concise Summary of the Approach**

The solution finds the length of the longest substring without repeating characters using a sliding window and a HashSet. It maintains a window with two pointers (left and right), expanding by moving right and adding characters to the HashSet. If a duplicate is found, it shrinks the window by moving left and removing characters until the duplicate is gone. The maximum window size (right - left + 1) is tracked. The approach achieves **O(n)** time complexity and **O(min(m, n))** space complexity, efficiently handling duplicates and variable-length strings.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each character is processed at most twice, and HashSet provides fast lookups.
  + **Clarity**: The sliding window with HashSet is intuitive and easy to explain, making it ideal for interviews.
  + **Robustness**: Handles all edge cases (null, empty, single-character, duplicates) correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you optimize space further?
    - **Response**: For ASCII, use a boolean array of size 128 instead of a HashSet, reducing space to O(1) for fixed character sets. For Unicode, a HashMap or HashSet is necessary, so the current approach is optimal for general cases.
  + **Interviewer might ask**: Can you use a different data structure?
    - **Response**: Use a HashMap to store character-to-index mappings, updating left to the position after the last occurrence of a duplicate. This achieves the same O(n) time but allows direct jumps for left, potentially reducing iterations slightly. (See alternative approach below.)
  + **Interviewer might ask**: How does this handle Unicode or special characters?
    - **Response**: The HashSet works for any character set (e.g., Unicode), with space bounded by the number of unique characters. If restricted to specific characters (e.g., lowercase letters), a smaller array (size 26) could be used.
* **Edge Cases Handled**:
  + Null or empty string: Returns 0 (explicitly checked).
  + Single character: Returns 1 (explicitly checked).
  + All unique characters (e.g., “abcde”): Returns string length (correctly tracked).
  + All duplicates (e.g., “aaaa”): Returns 1 (window shrinks to single character).
  + Mixed characters (e.g., “abca”): Correctly finds longest substring (e.g., “abc” → 3).
  + **Note**: The null and empty checks are robust, covering key edge cases.
* **Assumptions**:
  + The input string may contain any characters (including Unicode).
  + If the input could be invalid (e.g., null), the code handles it explicitly.
  + If specific character sets are assumed (e.g., lowercase letters), clarify with the interviewer.
* **Alternative Approaches**:
  + **HashMap with Index Tracking**: Use a HashMap to store the last index of each character. When a duplicate is found, move left to the position after the last occurrence. Time: O(n), Space: O(min(m, n)). Slightly more efficient in practice due to direct jumps.
*  **Brute Force**: Check all possible substrings and verify no duplicates. Time: O(n³), Space: O(1). Highly inefficient and not practical.
* The sliding window with HashSet or HashMap is preferred for its efficiency and clarity.

 **Suggestions for Improvement**:

* **Clarity**: Add comments explaining the sliding window (e.g., “Expand window with right, shrink with left on duplicates”).
* **Optimization**: Consider the HashMap approach for slightly faster duplicate handling by jumping left directly to the correct position.
* **Edge Case Discussion**: In an interview, mention handling of null, empty, single-character, and duplicate-heavy strings to show thoroughness.
* **Space Optimization**: If the character set is known (e.g., lowercase letters), use a boolean array of size 26 to reduce space to O(1).

 **Note on Sliding Window**: The HashSet ensures no duplicates in the current window. Shrinking the window by moving left when a duplicate is found maintains the invariant of unique characters, and tracking right - left + 1 gives the window size efficiently.

* Longest Repeating Character Replacement

Commented Code

class Solution {

*// Method to find the length of the longest substring with at most k replacements*

public int characterReplacement(String s, int k) {

*// Map to store frequency of characters in the current window*

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

int left = 0, right = 0; *// Initialize two pointers for sliding window*

int ans = 0; *// Track the length of the longest valid substring*

int maxOccurence = 0; *// Track the frequency of the most frequent character in window*

*// Iterate with right pointer to expand window*

while (right < s.length()) {

char c = s.charAt(right); *// Current character*

*// Update frequency of current character*

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

*// Update max frequency of any character in the current window*

maxOccurence = Math.max(maxOccurence, occurence.get(c));

*// If window size minus most frequent character's count exceeds k, shrink window*

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

char leftChar = s.charAt(left); *// Character at left pointer*

occurence.put(leftChar, occurence.get(leftChar) - 1); *// Decrease its frequency*

left++; *// Move left pointer to shrink window*

}

*// Update max length of valid substring*

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

right++; *// Move right pointer to expand window*

}

*// Return the length of the longest valid substring*

return ans;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The right pointer iterates through the string of length n exactly once.
  + The left pointer only moves forward (never backward) when the window becomes invalid, and each character is removed at most once.
  + Each character is processed at most twice: once when added by right and once when removed by left.
  + HashMap operations (put, getOrDefault, get) are O(1) on average.
  + Updating maxOccurence and ans is O(1) per iteration.
  + Therefore, the total time complexity is **O(n)**, where n is the length of the input string.
  + **Note**: In rare cases of hash collisions, HashMap operations could degrade, but this is unlikely with a good hash function.

**Space Complexity**

* **Space Complexity: O(m)**
  + The HashMap stores the frequency of characters in the current window, where m is the size of the character set (e.g., 26 for uppercase letters, 128 for ASCII, or more for Unicode).
  + The window size is at most n, but the HashMap only stores unique characters, so its size is bounded by min(m, n).
  + Other variables (left, right, ans, maxOccurence, c, leftChar) use O(1) space.
  + Thus, the space complexity is **O(m)**, where m is the size of the character set. For problems assuming uppercase letters only (e.g., LeetCode), this is effectively O(1).

**Concise Summary of the Approach**

The solution finds the longest substring where at most k characters can be replaced to make all characters identical, using a sliding window with a HashMap. It tracks character frequencies in the window and the maximum frequency of any character (maxOccurence). The window expands by moving right and adding characters. If the number of characters needing replacement (window size - maxOccurence) exceeds k, the window shrinks by moving left and removing characters. The maximum window size is tracked. The approach achieves **O(n)** time complexity and **O(m)** space complexity, efficiently handling variable window sizes and character frequencies.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each character is processed at most twice, and HashMap provides fast frequency updates.
  + **Clarity**: The sliding window with frequency tracking is intuitive and easy to explain, ideal for interviews.
  + **Robustness**: Handles all edge cases (empty strings, large k, single characters) and works for any character set.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why track maxOccurence instead of recalculating it?
    - **Response**: Recalculating the maximum frequency in the HashMap each time would take O(m) per iteration, making the overall time O(n \* m). Tracking maxOccurence dynamically ensures O(1) updates, keeping the total time O(n).
  + **Interviewer might ask**: Can you optimize space for a fixed character set?
    - **Response**: For a fixed character set (e.g., uppercase letters), use an array of size 26 instead of a HashMap, reducing space to O(1). The current approach is general and works for any character set.
  + **Interviewer might ask**: How does this handle edge cases like k >= n?
    - **Response**: If k is at least the string length, the entire string is valid (as all characters can be replaced), and the code correctly returns n since the window never shrinks.
* **Edge Cases Handled**:
  + Null or empty string: Not explicitly checked, but returns 0 implicitly (as right < s.length() fails).
  + Single character: Returns 1 (window size of 1, no replacements needed).
  + Large k (e.g., k >= n): Returns string length, as all characters can be replaced.
  + All same characters (e.g., “AAAA”): Returns n, as no replacements are needed.
  + No valid substring: Returns correct length based on maximum valid window.
  + **Note**: Add explicit null/empty checks (e.g., if (s == null || s.length() == 0) return 0;) for robustness.
* **Assumptions**:
  + The input string is non-null and may contain any characters.
  + k is a non-negative integer.
  + The problem typically assumes uppercase letters (e.g., LeetCode), but the code works for any character set.
  + If assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches**:
  + **Array-Based for Fixed Character Set**: Use a fixed-size array (e.g., size 26 for uppercase letters) instead of a HashMap. Time: O(n), Space: O(1). More efficient for small character sets but less general.
  + **Brute Force**: Try all possible substrings and check if they can be made valid with k replacements. Time: O(n²), Space: O(m). Highly inefficient.
  + The sliding window with HashMap is preferred for its efficiency and generality.
* **Suggestions for Improvement**:
  + **Input Validation**: Add explicit checks for null or empty strings (e.g., if (s == null || s.length() == 0) return 0;) to handle edge cases robustly.
  + **Clarity**: Add comments explaining the window validity condition (e.g., “Shrink window if replacements needed exceed k”).
  + **Edge Case Discussion**: In an interview, mention handling of large k, single characters, and varying character sets to show thoroughness.
  + **Optimization**: If the character set is known (e.g., uppercase letters), suggest using an array to reduce space to O(1), but note the current approach’s flexibility.
* **Note on Window Validity**: The condition right - left + 1 - maxOccurence > k checks if the number of characters needing replacement (window size minus the count of the most frequent character) exceeds k. Shrinking the window ensures validity while maximizing length.
* Permutation In String

Commented Code

class Solution {

*// Method to check if s2 contains a permutation of s1*

public boolean checkInclusion(String s1, String s2) {

*// If s1 is longer than s2, no permutation is possible*

if (s1.length() > s2.length()) {

return false;

}

*// Initialize frequency arrays for lowercase letters (a-z)*

int[] s1Map = new int[26]; *// Frequency of characters in s1*

int[] s2Map = new int[26]; *// Frequency of characters in s2's current window*

*// Build frequency maps for s1 and first window of s2*

for (int i = 0; i < s1.length(); i++) {

s1Map[s1.charAt(i) - 'a']++; *// Count characters in s1*

s2Map[s2.charAt(i) - 'a']++; *// Count characters in s2's first window*

}

*// Slide window over s2, checking each window of length s1.length()*

for (int i = 0; i < s2.length() - s1.length(); i++) {

if (matches(s1Map, s2Map)) { *// Check if current window matches s1's frequency*

return true;

}

*// Slide window: add next character, remove first character*

s2Map[s2.charAt(i + s1.length()) - 'a']++; *// Add next character*

s2Map[s2.charAt(i) - 'a']--; *// Remove first character*

}

*// Check the last window*

return matches(s1Map, s2Map);

}

*// Helper method to check if two frequency arrays are identical*

private boolean matches(int[] s1Map, int[] s2Map) {

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

if (s1Map[i] != s2Map[i]) {

return false; *// Return false if frequencies differ*

}

}

return true; *// Return true if all frequencies match*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + **Initial Setup**: Building s1Map and s2Map for the first m characters (where m is s1.length()) takes O(m), which is O(n) in the worst case (if m approaches n, the length of s2).
  + **Sliding Window**: The loop iterates n - m times, where n is the length of s2. Each iteration performs:
    - A call to matches, which checks 26 elements (O(1) for a fixed-size alphabet).
    - Two array updates (s2Map increment and decrement), each O(1).
  + The total for the sliding window is O(n - m) \* O(1) ≈ O(n), as each iteration is constant time.
  + **Final Check**: The last matches call is O(1) (fixed 26 elements).
  + Overall, the time complexity is dominated by the linear scan of s2, resulting in **O(n)**, where n is the length of s2.

**Space Complexity**

* **Space Complexity: O(1)**
  + The two arrays s1Map and s2Map are of fixed size (26, for lowercase letters), so they use O(1) space.
  + Other variables (i, loop counters) use O(1) space.
  + The input strings are not counted as extra space.
  + Therefore, the space complexity is **O(1)**, assuming a fixed character set (lowercase letters).

**Concise Summary of the Approach**

The solution checks if s2 contains a permutation of s1 using a sliding window with frequency arrays. It first checks if s1 is longer than s2 (impossible case). It then builds frequency arrays for s1 and the first s1.length() characters of s2. A sliding window of size s1.length() moves across s2, updating the frequency array by adding the next character and removing the first. At each step, it checks if the window’s frequencies match s1’s. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently leveraging fixed-size arrays for a lowercase alphabet.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, processing each character of s2 once, and O(1) space is ideal for a fixed alphabet.
  + **Clarity**: The sliding window with frequency arrays is intuitive and easy to explain, making it interview-friendly.
  + **Robustness**: Handles edge cases (e.g., s1 longer than s2, empty strings, single characters) and assumes lowercase letters, as is common in the problem.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use arrays instead of a HashMap?
    - **Response**: For lowercase letters (26 characters), a fixed-size array is O(1) space and faster than a HashMap (O(m) space, where m is the character set size). For larger character sets (e.g., Unicode), a HashMap would be necessary, with O(m) space.
  + **Interviewer might ask**: How does this handle larger character sets?
    - **Response**: For Unicode or arbitrary characters, use a HashMap to store frequencies, increasing space to O(m), where m is the number of unique characters. The time complexity remains O(n), but matches becomes O(m).
  + **Interviewer might ask**: Can you optimize the matches check?
    - **Response**: Track the number of matching frequencies dynamically (e.g., increment/decrement a counter when frequencies change) to avoid the O(26) loop in matches. This can reduce constant factors but doesn’t change the asymptotic complexity.
* **Edge Cases Handled**:
  + s1 longer than s2: Returns false (explicitly checked).
  + Empty s1 or s2: Handled implicitly (s1.length() > s2.length() covers empty s2, and empty s1 would return true in some problem variants, but clarify).
  + Single character: Correctly checks for a matching single character in s2.
  + No permutation exists: Returns false after checking all windows.
  + **Note**: Add explicit checks for empty s1 (e.g., if (s1.length() == 0) return true;) or null inputs if required.
* **Assumptions**:
  + s1 and s2 contain only lowercase letters (per common problem constraints, e.g., LeetCode).
  + s1 and s2 are non-null.
  + If these assumptions don’t hold, clarify with the interviewer (e.g., handling uppercase or Unicode).
* **Alternative Approaches** (Description Only, No Code):
  + **HashMap-Based Sliding Window**: Use a HashMap for frequencies instead of arrays, suitable for arbitrary character sets. Time: O(n), Space: O(m), where m is the number of unique characters. Less efficient for lowercase letters but more general.
  + **Brute Force**: Generate all permutations of s1 and check if any substring of s2 matches. Time: O(n! \* n), Space: O(1). Highly inefficient.
  + The array-based sliding window is preferred for its efficiency and simplicity given the lowercase letter constraint.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty inputs (e.g., if (s1 == null || s2 == null || s1.length() == 0) return true;) to handle edge cases explicitly.
  + **Clarity**: Add comments explaining the frequency matching (e.g., “Compare frequency arrays to check for permutation”).
  + **Optimization**: Track matching frequencies dynamically to avoid the matches loop, reducing constant factors (e.g., maintain a counter for equal frequencies).
  + **Edge Case Discussion**: In an interview, mention handling of empty strings, single characters, and the lowercase assumption to show thoroughness.
* **Note on Frequency Matching**: The matches function checks if two frequency arrays are identical, ensuring the current window in s2 is a permutation of s1. The sliding window efficiently updates frequencies by adding one character and removing another, keeping the window size fixed at s1.length().
* Minimum Window Substring

Commented Code

class Solution {

*// Method to find the minimum window substring of s that contains all characters of t*

public String minWindow(String s, String t) {

*// If s is shorter than t, no valid window is possible*

if (s.length() < t.length()) return "";

*// Initialize frequency array for characters in t (ASCII, size 128)*

int[] need = new int[128];

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

need[c]++; *// Store frequency of each character in t*

}

int left = 0, right = 0; *// Initialize two pointers for sliding window*

int required = t.length(); *// Number of characters still needed*

int minStart = 0, minLen = Integer.MAX\_VALUE; *// Track minimum window*

*// Expand window by moving right pointer*

while (right < s.length()) {

*// If current character is needed, reduce required count*

if (need[s.charAt(right)] > 0) {

required--;

}

need[s.charAt(right)]--; *// Decrement frequency (mark as used)*

right++; *// Move right pointer*

*// When all required characters are found, try to shrink window*

while (required == 0) {

*// Update minimum window if current window is smaller*

if (right - left < minLen) {

minLen = right - left;

minStart = left;

}

*// Shrink window by moving left pointer*

need[s.charAt(left)]++; *// Increment frequency (release character)*

*// If we released a needed character, increase required count*

if (need[s.charAt(left)] > 0) {

required++;

}

left++; *// Move left pointer*

}

}

*// Return empty string if no valid window found, else return substring*

return minLen == Integer.MAX\_VALUE ? "" : s.substring(minStart, minStart + minLen);

}

}

**Time Complexity**

* **Time Complexity: O(n + m)**
  + **Initialization**: Building the need array for t takes O(m), where m is the length of t.
  + **Sliding Window**: The right pointer iterates through s (length n) once. The left pointer only moves forward when required == 0, and each character in s is processed at most twice (once when added by right, once when removed by left).
  + Array operations (need[c]++, need[c]--) are O(1) since the array size is fixed (128 for ASCII).
  + The total work for the sliding window is O(n), as each character is visited a constant number of times.
  + Overall, the time complexity is O(n + m), where n is the length of s and m is the length of t. In practice, since m ≤ n, this is often simplified to **O(n)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The need array is of fixed size (128 for ASCII), so it uses O(1) space.
  + Other variables (left, right, required, minStart, minLen) use O(1) space.
  + The output substring is not counted as extra space per problem conventions.
  + Therefore, the space complexity is **O(1)**, assuming a fixed character set (e.g., ASCII). For Unicode, a HashMap would make it O(m), where m is the number of unique characters in t.

**Concise Summary of the Approach**

The solution finds the smallest substring of s that contains all characters of t (including duplicates) using a sliding window. It uses a frequency array to track characters needed from t. The window expands by moving right, decrementing the frequency of each character and reducing required when a needed character is found. When all characters are found (required == 0), it shrinks the window by moving left, updating the minimum window if smaller. If no valid window is found, it returns an empty string. The approach achieves **O(n + m)** time complexity and **O(1)** space complexity for a fixed character set, efficiently handling variable window sizes.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n + m) time is optimal, processing each character of s at most twice, and O(1) space is ideal for a fixed alphabet.
  + **Clarity**: The sliding window with frequency tracking is intuitive and standard for substring problems, making it interview-friendly.
  + **Robustness**: Handles edge cases (e.g., s shorter than t, no valid window, duplicates) correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a frequency array instead of a HashMap?
    - **Response**: For a fixed character set (e.g., ASCII or lowercase letters), a size-128 array is O(1) space and faster than a HashMap (O(m) space, where m is unique characters in t). For Unicode, a HashMap would be necessary.
  + **Interviewer might ask**: How does this handle Unicode or larger character sets?
    - **Response**: Use a HashMap to store character frequencies, increasing space to O(m), where m is the number of unique characters in t. Time remains O(n + m).
  + **Interviewer might ask**: Can you optimize further?
    - **Response**: The current approach is near-optimal. Tracking required avoids recomputing the number of needed characters, and the fixed-size array minimizes space. For small constant factors, we could track unique characters in t to skip unnecessary checks, but it doesn’t change the asymptotic complexity.
* **Edge Cases Handled**:
  + s shorter than t: Returns empty string (explicitly checked).
  + Empty t: Should return empty string (problem typically assumes t is non-empty, but clarify).
  + No valid window: Returns empty string (minLen remains Integer.MAX\_VALUE).
  + Duplicates in t: Handled by tracking frequencies in need.
  + Single character in t: Finds smallest window containing that character.
  + **Note**: Add checks for null or empty inputs (e.g., if (s == null || t == null || t.length() == 0) return "";) for robustness.
* **Assumptions**:
  + s and t are non-null and contain ASCII characters (or lowercase letters, per some problem constraints).
  + t is non-empty, and a valid window must include all characters of t with correct frequencies.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only, No Code):
  + **HashMap-Based Sliding Window**: Use a HashMap for character frequencies instead of an array, suitable for arbitrary character sets. Time: O(n + m), Space: O(m), where m is unique characters in t. Less efficient for small alphabets but more general.
  + **Brute Force**: Check all possible substrings of s and verify if they contain all characters of t. Time: O(n² \* m), Space: O(m). Highly inefficient.
  + The array-based sliding window is preferred for its efficiency and simplicity given the ASCII or lowercase constraint.
* **Suggestions for Improvement**:
  + **Input Validation**: Add explicit checks for null or empty inputs (e.g., if (s == null || t == null || t.length() == 0) return "";) to handle edge cases robustly.
  + **Clarity**: Add comments explaining the required counter (e.g., “Tracks remaining characters needed from t”) and window shrinking logic.
  + **Edge Case Discussion**: In an interview, mention handling of duplicates, no valid window, and the fixed-size array assumption to show thoroughness.
  + **Optimization**: If the problem specifies lowercase letters only, use a size-26 array to reduce space slightly, though 128 is still O(1).
* **Note on Window Validity**: The required counter tracks how many characters are still needed to match t’s frequencies. When required == 0, the window contains all characters of t, allowing shrinking to find the smallest valid window. The need array tracks both needed (positive) and excess (negative) characters, ensuring accurate frequency management.
* Sliding Window Maximum

Commented Code

public class Solution {

*// Method to find the maximum element in each sliding window of size k*

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

*// Handle edge cases: null array, empty array, or invalid k*

if (nums == null || nums.length == 0 || k <= 0) {

return new int[0];

}

int n = nums.length; *// Length of input array*

int[] result = new int[n - k + 1]; *// Result array for max values of each window*

Deque<Integer> deque = new LinkedList<>(); *// Deque to store indices of potential max elements*

*// Iterate through the array*

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

*// Remove indices outside the current window (i - k + 1)*

while (!deque.isEmpty() && deque.peek() < i - k + 1) {

deque.poll(); *// Remove from front*

}

*// Remove indices of smaller elements, as they can't be the max*

while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {

deque.pollLast(); *// Remove from back*

}

*// Add current index to deque*

deque.offer(i);

*// If window size is reached, add max (element at front) to result*

if (i >= k - 1) {

result[i - k + 1] = nums[deque.peek()];

}

}

*// Return array of maximums for each window*

return result;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The loop iterates through the array of length n exactly once.
  + Each index is added to the Deque once (offer) and removed at most once (either via poll for out-of-window indices or pollLast for smaller elements). Deque operations (offer, poll, pollLast, peek) are O(1) amortized for a LinkedList.
  + Each element is processed a constant number of times (added once, potentially removed once), and operations within the loop (comparisons, array access) are O(1).
  + Therefore, the total time complexity is **O(n)**, where n is the length of the input array.

**Space Complexity**

* **Space Complexity: O(k)**
  + The Deque stores indices of potential maximum elements in the current window. In the worst case (e.g., decreasing array like [9,8,7,...]), it stores up to k indices.
  + The result array of size n - k + 1 is required for output and is not counted as extra space per problem conventions.
  + Other variables (i, n) use O(1) space.
  + Thus, the space complexity is **O(k)**, where k is the window size. If the output array is counted (in some interview contexts), it would add O(n - k + 1).

**Concise Summary of the Approach**

The solution finds the maximum element in each sliding window of size k in an array using a monotonic deque. It maintains a deque of indices where elements are in decreasing order, ensuring the front always holds the maximum’s index. For each position, it removes out-of-window indices and smaller elements from the deque, adds the current index, and records the maximum (front element) once the window size is reached. The approach achieves **O(n)** time complexity and **O(k)** space complexity, efficiently tracking maximums with a deque.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each element is processed a constant number of times, and the deque ensures O(1) operations.
  + **Clarity**: The monotonic deque is a standard technique for sliding window maximums, making it interview-friendly once explained.
  + **Robustness**: Handles edge cases (null/empty array, invalid k, duplicates) correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a deque instead of a max-heap?
    - **Response**: A max-heap (e.g., PriorityQueue) would take O(log k) for insertions and deletions, leading to O(n log k) time. The deque maintains a monotonic order, allowing O(1) operations, resulting in O(n) time.
  + **Interviewer might ask**: Why keep indices instead of values in the deque?
    - **Response**: Indices allow checking if an element is within the current window (i - k + 1) and accessing the corresponding value via nums[deque.peek()]. Storing values alone wouldn’t provide window position information.
  + **Interviewer might ask**: How does this handle duplicates or negative numbers?
    - **Response**: The deque compares values (nums[deque.peekLast()] < nums[i]), so duplicates and negative numbers are handled correctly, as the comparison is based on element values, not indices.
* **Edge Cases Handled**:
  + Null or empty array: Returns empty array (explicitly checked).
  + Invalid k (k <= 0): Returns empty array (explicitly checked).
  + k == 1: Returns array of all elements (each window has one element).
  + k == n: Returns single maximum (one window covers the entire array).
  + Duplicates or negative numbers: Handled correctly by comparing values.
  + **Note**: The code robustly handles edge cases with explicit checks.
* **Assumptions**:
  + The input array is non-null (checked explicitly).
  + k is a positive integer and k ≤ n (checked explicitly for k <= 0, assumes k ≤ n per problem constraints).
  + If these assumptions don’t hold, the explicit checks cover most cases.
* **Alternative Approaches** (Description Only):
  + **Brute Force**: For each window of size k, find the maximum by scanning all elements. Time: O(n \* k), Space: O(1) (excluding output). Inefficient for large k.
  + **Max-Heap**: Use a max-heap to store elements in the window, removing out-of-window elements. Time: O(n log k), Space: O(k). Less efficient than the deque approach.
  + The deque-based approach is preferred for its optimal O(n) time and clarity.
* **Suggestions for Improvement**:
  + **Clarity**: Add comments explaining the monotonic deque (e.g., “Maintain deque of indices in decreasing order to track max”).
  + **Edge Case Discussion**: In an interview, mention handling of k == n, duplicates, and negative numbers to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that the deque’s monotonic property eliminates unnecessary elements, ensuring efficiency.
  + **Validation**: The null/empty checks are sufficient, but could add an explicit check for k > n (e.g., if (k > nums.length) return new int[0];) for completeness.
* **Note on Monotonic Deque**: The deque maintains indices in decreasing order of their corresponding values (nums[deque.peekLast()] < nums[i]). This ensures the front always holds the maximum element’s index, and removing smaller elements avoids keeping irrelevant candidates for future windows.

4. Stack (7 problems)

* Valid Parentheses

Commented Code

class Solution {

*// Method to check if a string of parentheses is valid*

public boolean isValid(String s) {

*// Map closing brackets to their corresponding opening brackets*

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

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

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

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

*// 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, push it onto the stack*

if (!mappedBrackets.containsKey(c)) {

stack.push(c);

} else {

*// If stack is empty, no matching opening bracket*

if (stack.empty()) {

return false;

}

*// Pop top element and check if it matches the expected opening bracket*

char topElement = stack.pop();

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

return false;

}

}

}

*// Return true if stack is empty (all brackets matched)*

return stack.isEmpty();

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the input string of length n exactly once.
  + Each operation inside the loop is O(1):
    - Checking mappedBrackets.containsKey: O(1) (average case for HashMap).
    - Stack operations (push, pop, empty): O(1).
    - Accessing mappedBrackets.get: O(1).
  + Initializing the HashMap with three entries is O(1).
  + Therefore, the total time complexity is **O(n)**, where n is the length of the input string.

**Space Complexity**

* **Space Complexity: O(n)**
  + The HashMap stores three key-value pairs (for ), }, ]), which is O(1).
  + The Stack stores at most n/2 opening brackets in the worst case (e.g., "((("), requiring O(n) space.
  + Therefore, the total space complexity is **O(n)** due to the stack.

**Concise Summary of the Approach**

The solution checks if a string of parentheses is valid using a stack and a HashMap. The HashMap maps closing brackets to their corresponding opening brackets. For each character, if it’s an opening bracket, it’s pushed onto the stack; if it’s a closing bracket, the stack’s top element is popped and checked against the expected opening bracket. If the stack is empty when a closing bracket is encountered or the brackets don’t match, the string is invalid. The string is valid if the stack is empty at the end. The approach achieves **O(n)** time complexity and **O(n)** space complexity, efficiently validating parentheses.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each character must be processed, and the stack-based approach is standard.
  + **Clarity**: Using a stack and HashMap is intuitive for tracking matching brackets, making it interview-friendly.
  + **Robustness**: Handles all valid bracket types and edge cases correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a HashMap instead of direct comparisons?
    - **Response**: The HashMap provides a clean way to map closing brackets to opening ones, improving readability and extensibility (e.g., for new bracket types). Direct comparisons (e.g., if (c == ')')) are also O(1) but less maintainable.
  + **Interviewer might ask**: Can you solve it without a HashMap?
    - **Response**: Yes, use if-else or switch statements to check matching pairs directly (e.g., if (c == ')' && top != '(')). This is still O(n) time and O(n) space but slightly less elegant.
  + **Interviewer might ask**: How does this handle invalid characters?
    - **Response**: The code assumes valid bracket characters (per problem constraints). To handle invalid characters, add a check (e.g., if (!mappedBrackets.containsKey(c) && !mappedBrackets.containsValue(c)) return false;).
* **Edge Cases Handled**:
  + Empty string: Returns true (stack is empty).
  + Single character: Returns false (stack not empty or mismatch).
  + Unmatched closing bracket: Returns false (stack empty or mismatch).
  + Unmatched opening bracket: Returns false (stack not empty at end).
  + **Note**: Add checks for null or invalid characters (e.g., if (s == null) return false;) for robustness.
* **Assumptions**:
  + The input string contains only valid bracket characters ((, ), {, }, [, ]).
  + The string is non-null.
  + If these assumptions don’t hold (e.g., letters in string), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Direct Comparison**: Use if-else or switch to check bracket pairs without a HashMap. Time: O(n), Space: O(n). Slightly less clean but avoids HashMap overhead.
  + **String Replacement (Non-Standard)**: Repeatedly remove matching pairs (e.g., ()). Time: O(n²) worst case, Space: O(n). Inefficient and not practical.
  + The stack-based approach is preferred for its O(n) time and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or invalid characters (e.g., if (s == null || s.isEmpty()) return true;, or check for non-bracket characters).
  + **Clarity**: Add comments explaining stack usage (e.g., “Stack tracks opening brackets for matching”).
  + **Edge Case Discussion**: In an interview, mention handling of empty strings, unmatched brackets, and invalid inputs to show thoroughness.
  + **Optimization**: Replace HashMap with direct comparisons to reduce constant factors:

if (c == '(' || c == '{' || c == '[') {

stack.push(c);

} else {

if (stack.isEmpty()) return false;

char top = stack.pop();

if ((c == ')' && top != '(') || (c == '}' && top != '{') || (c == ']' && top != '[')) {

return false;

}

}

This avoids HashMap but is less extensible.

* **Note on Stack Logic**: The stack tracks opening brackets. A closing bracket must match the most recent opening bracket (top of stack). The HashMap ensures correct pairing, and the final isEmpty check confirms all brackets are matched. The approach is robust for nested and sequential brackets.
* Min Stack

Commented Code

public class MinStack {

private Stack<Integer> stack; *// Main stack to store all elements*

private Stack<Integer> minStack; *// Auxiliary stack to track minimums*

*// Constructor to initialize both stacks*

public MinStack() {

stack = new Stack<>();

minStack = new Stack<>();

}

*// Push a value onto the main stack, update minStack if needed*

public void push(int val) {

stack.push(val); *// Add to main stack*

*// Push to minStack if it's empty or val is less than or equal to current min*

if (minStack.isEmpty() || val <= minStack.peek()) {

minStack.push(val);

}

}

*// Pop a value from the main stack, update minStack if needed*

public void pop() {

if (stack.isEmpty()) return; *// Handle empty stack*

int top = stack.pop(); *// Remove top element*

*// If popped element is the current minimum, pop from minStack*

if (top == minStack.peek()) {

minStack.pop();

}

}

*// Return the top element of the main stack*

public int top() {

return stack.peek();

}

*// Return the current minimum element from minStack*

public int getMin() {

return minStack.peek();

}

}

**Commented Code**

public class MinStack {

private Stack<Integer> stack; *// Main stack to store all elements*

private Stack<Integer> minStack; *// Auxiliary stack to track minimums*

*// Constructor to initialize both stacks*

public MinStack() {

stack = new Stack<>();

minStack = new Stack<>();

}

*// Push a value onto the main stack, update minStack if needed*

public void push(int val) {

stack.push(val); *// Add to main stack*

*// Push to minStack if it's empty or val is less than or equal to current min*

if (minStack.isEmpty() || val <= minStack.peek()) {

minStack.push(val);

}

}

*// Pop a value from the main stack, update minStack if needed*

public void pop() {

if (stack.isEmpty()) return; *// Handle empty stack*

int top = stack.pop(); *// Remove top element*

*// If popped element is the current minimum, pop from minStack*

if (top == minStack.peek()) {

minStack.pop();

}

}

*// Return the top element of the main stack*

public int top() {

return stack.peek();

}

*// Return the current minimum element from minStack*

public int getMin() {

return minStack.peek();

}

}

**Time Complexity**

* **Constructor (MinStack)**:
  + **Time Complexity: O(1)** – Initializing two empty stacks is a constant-time operation.
* **push(val)**:
  + **Time Complexity: O(1)** – Pushing to stack is O(1). Checking minStack.isEmpty() or minStack.peek() and pushing to minStack (if needed) are also O(1).
* **pop()**:
  + **Time Complexity: O(1)** – Popping from stack is O(1). Checking minStack.peek() and popping from minStack (if needed) are O(1).
* **top()**:
  + **Time Complexity: O(1)** – Peeking at the top of stack is O(1).
* **getMin()**:
  + **Time Complexity: O(1)** – Peeking at the top of minStack is O(1).
* **Summary**: All operations (push, pop, top, getMin) have **O(1)** time complexity, meeting the problem’s requirement for constant-time operations.

**Space Complexity**

* **Space Complexity: O(n)**
  + The stack stores all n elements pushed onto it, requiring O(n) space.
  + The minStack stores elements that are minimums at the time of pushing. In the worst case (e.g., pushing elements in decreasing order like 3, 2, 1), minStack stores all n elements, requiring O(n) space.
  + Therefore, the total space complexity is **O(n)**, where n is the number of elements pushed.

**Concise Summary of the Approach**

The solution implements a stack that supports push, pop, top, and getMin in O(1) time using two stacks: a main stack for all elements and a minStack to track the minimum at each state. For push, the value is added to stack and to minStack if it’s less than or equal to the current minimum. For pop, the top element is removed from stack, and if it equals the current minimum, it’s also removed from minStack. The top operation returns the main stack’s top, and getMin returns the minStack’s top. The approach achieves **O(1)** time for all operations and **O(n)** space, efficiently maintaining the minimum.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(1) time for all operations is optimal, as required by the problem.
  + **Clarity**: Using two stacks is a standard, intuitive solution, making it interview-friendly.
  + **Correctness**: Maintains the minimum accurately, handling duplicates and edge cases.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a second stack (minStack)?
    - **Response**: The minStack tracks the minimum at each state of the main stack. Without it, finding the minimum would require O(n) scanning. Storing minimums ensures O(1) getMin.
  + **Interviewer might ask**: Why push to minStack when val <= minStack.peek()?
    - **Response**: Including equal values handles duplicates correctly. If a minimum value is pushed multiple times, it must be popped multiple times to maintain the correct minimum (e.g., pushing 1, 1 requires two pops to update the minimum).
  + **Interviewer might ask**: Can you optimize space?
    - **Response**: A space-optimized approach could store differences or use a single stack with pairs (value, minimum), but it’s more complex and still O(n) space in the worst case. The two-stack approach is clearer and meets all requirements.
* **Edge Cases Handled**:
  + Empty stack: pop checks for empty stack; top and getMin assume non-empty stack (per problem constraints).
  + Single element: Works correctly (both stacks have one element).
  + Duplicates: Handled by pushing to minStack when val <= minStack.peek().
  + Decreasing sequence (e.g., 3, 2, 1): minStack stores all elements.
  + **Note**: Add explicit checks for top and getMin (e.g., if (stack.isEmpty()) throw new EmptyStackException();) for robustness in real systems.
* **Assumptions**:
  + The stack is non-empty for top and getMin calls (per problem constraints).
  + Input values are integers, and operations are valid.
  + If these assumptions don’t hold (e.g., calling pop on empty stack), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Single Stack with Pairs**: Store pairs of (value, current minimum) in one stack. Time: O(1) for all operations, Space: O(n). Saves no space but avoids two stacks, slightly less intuitive.
  + **Difference Encoding**: Store differences between values and minimums to reduce space in some cases. Time: O(1), Space: O(n) worst case. Complex and error-prone.
  + The two-stack approach is preferred for its simplicity and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for top and getMin:

public int top() {

if (stack.isEmpty()) throw new EmptyStackException();

return stack.peek();

}

public int getMin() {

if (minStack.isEmpty()) throw new EmptyStackException();

return minStack.peek();

}

* + **Use ArrayDeque**: Replace Stack with ArrayDeque for better performance:

private Deque<Integer> stack = new ArrayDeque<>();

private Deque<Integer> minStack = new ArrayDeque<>();

Stack is a legacy class; ArrayDeque is more efficient.

* + **Clarity**: Add comments explaining minStack’s role (e.g., “minStack tracks minimum at each push”).
  + **Edge Case Discussion**: In an interview, mention handling of duplicates, empty stack, and decreasing sequences to show thoroughness.
* **Note on minStack Logic**: The minStack maintains the minimum value at each state of the main stack. Pushing when val <= minStack.peek() ensures duplicates are handled (e.g., pushing 1, 1 keeps both in minStack). Popping from minStack only when the main stack’s top equals the minimum ensures the correct minimum is updated.
* Evaluate Reverse Polish Notation

Commented Code

class Solution {

*// Method to evaluate a Reverse Polish Notation expression*

public int evalRPN(String[] tokens) {

Stack<Integer> stack = new Stack<>(); *// Stack to store operands*

*// Iterate through each token in the input array*

for (String c : tokens) {

if (c.equals("+")) {

*// Pop two operands, add them, and push result*

stack.push(stack.pop() + stack.pop());

} else if (c.equals("-")) {

*// Pop two operands, subtract (second - first), and push result*

int a = stack.pop(); *// Second operand*

int b = stack.pop(); *// First operand*

stack.push(b - a);

} else if (c.equals("\*")) {

*// Pop two operands, multiply, and push result*

stack.push(stack.pop() \* stack.pop());

} else if (c.equals("/")) {

*// Pop two operands, divide (second / first), and push result*

int a = stack.pop(); *// Second operand*

int b = stack.pop(); *// First operand*

stack.push(b / a);

} else {

*// Token is a number, parse and push to stack*

stack.push(Integer.parseInt(c));

}

}

*// Return final result from stack*

return stack.pop();

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the array of n tokens exactly once.
  + Each operation is O(1):
    - String comparison (c.equals) is O(1) for short operator strings.
    - Stack operations (push, pop) are O(1).
    - Integer.parseInt is O(1) for typical integer strings (bounded length).
  + Therefore, the total time complexity is **O(n)**, where n is the length of the tokens array.

**Space Complexity**

* **Space Complexity: O(n)**
  + The stack stores operands, which, in the worst case (e.g., all numbers before operators), holds up to n elements (though typically closer to n/2 for balanced RPN expressions).
  + Other variables (c, a, b) use O(1) space.
  + Therefore, the space complexity is **O(n)** due to the stack.

**Concise Summary of the Approach**

The solution evaluates a Reverse Polish Notation (RPN) expression using a stack. It processes each token: numbers are parsed and pushed onto the stack, while operators (+, -, \*, /) pop two operands, perform the operation (with correct order for subtraction and division), and push the result. The final result is popped from the stack. The approach achieves **O(n)** time complexity and **O(n)** space complexity, efficiently handling RPN expressions with a stack-based method.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each token must be processed, and the stack ensures correct operand ordering.
  + **Clarity**: The stack-based approach is intuitive for RPN, leveraging LIFO to handle operands, making it interview-friendly.
  + **Robustness**: Correctly handles valid RPN expressions with operators and integers.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why pop two operands for subtraction and division in a specific order?
    - **Response**: In RPN, operators apply to the two most recent operands. For subtraction (b - a) and division (b / a), the second-popped operand (a) is the second operand, and the first-popped (b) is the first, ensuring correct evaluation (e.g., "3 1 -" means 3 - 1).
  + **Interviewer might ask**: How does this handle invalid inputs?
    - **Response**: The code assumes valid RPN expressions (per problem constraints). To handle invalid inputs, add checks for null arrays, invalid tokens, or insufficient operands (e.g., stack size < 2 for operators).
  + **Interviewer might ask**: Can you optimize space?
    - **Response**: The stack is necessary for RPN evaluation, and O(n) space is unavoidable in the worst case (e.g., all numbers before operators). No significant space optimization is possible without changing the problem.
* **Edge Cases Handled**:
  + Single token (e.g., ["5"]): Returns the number (popped from stack).
  + Minimal expression (e.g., ["2","1","+"]): Correctly evaluates to 3.
  + Large numbers or negative numbers: Handled by Integer.parseInt.
  + **Note**: Add checks for invalid cases (e.g., null array, invalid tokens, or insufficient operands) for robustness.
* **Assumptions**:
  + The input tokens array is non-null and contains valid RPN expressions (integers or operators +, -, \*, /).
  + At least one token exists, and the expression is well-formed (sufficient operands for operators).
  + Division results in integer division (truncating toward zero, per problem constraints).
  + If these assumptions don’t hold (e.g., invalid tokens like "abc"), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Recursive Evaluation**: Parse the expression recursively by evaluating sub-expressions. Time: O(n), Space: O(n) due to call stack. More complex and less intuitive for RPN.
  + **Array-Based**: Store operands in an array and track position, updating it for operations. Time: O(n), Space: O(n). Less efficient due to array resizing or shifting.
  + The stack-based approach is preferred for its simplicity and alignment with RPN’s LIFO nature.
* **Suggestions for Improvement**:
  + **Input Validation**:
    - Add checks for null or empty input:

if (tokens == null || tokens.length == 0) throw new IllegalArgumentException("Invalid input");

* + - Validate tokens and stack size:

if (c.matches("[+\\-\*/]") && stack.size() < 2) throw new IllegalArgumentException("Insufficient operands");

* + - Check for valid integers using a try-catch for Integer.parseInt:

try {

stack.push(Integer.parseInt(c));

} catch (NumberFormatException e) {

throw new IllegalArgumentException("Invalid number: " + c);

}

* + **Use ArrayDeque**: Replace Stack with ArrayDeque for better performance:

Deque<Integer> stack = new ArrayDeque<>();

Stack is a legacy class; ArrayDeque is more efficient.

* + **Clarity**: Add comments explaining operator order (e.g., “Pop second operand first for subtraction/division”).
  + **Edge Case Discussion**: In an interview, mention handling of single tokens, negative numbers, and potential invalid inputs to show thoroughness.
  + **Optimization**: The current approach is optimal, but using a HashSet for operators could slightly streamline checks:

private static final Set<String> OPERATORS = new HashSet<>(Arrays.asList("+", "-", "\*", "/"));

if (OPERATORS.contains(c)) { ... } else { stack.push(Integer.parseInt(c)); }

* **Note on Operator Order**: For subtraction and division, the order of popping matters (e.g., b - a instead of a - b). This ensures correct RPN evaluation, as the first operand popped is the second in the expression (e.g., "3 1 -" means 3 - 1 = 2).
* Generate Parentheses

Commented Code

class Solution {

*// Method to generate all valid combinations of n pairs of parentheses*

public List<String> generateParenthesis(int n) {

List<String> ans = new ArrayList(); *// List to store valid combinations*

backtrack(ans, new StringBuilder(), 0, 0, n); *// Start backtracking*

return ans;

}

*// Helper method for backtracking*

public void backtrack(List<String> ans, StringBuilder cur, int open, int close, int max) {

*// Base case: if current string length is 2\*n, add to result*

if (cur.length() == max \* 2) {

ans.add(cur.toString());

return;

}

*// Add an opening parenthesis if we haven't used all n*

if (open < max) {

cur.append("("); *// Add '('*

backtrack(ans, cur, open + 1, close, max); *// Recurse*

cur.deleteCharAt(cur.length() - 1); *// Backtrack by removing '('*

}

*// Add a closing parenthesis if valid (more open than close)*

if (close < open) {

cur.append(")"); *// Add ')'*

backtrack(ans, cur, open, close + 1, max); *// Recurse*

cur.deleteCharAt(cur.length() - 1); *// Backtrack by removing ')'*

}

}

}

**Time Complexity**

* **Time Complexity: O(4ⁿ / √n)**
  + The problem generates all valid combinations of n pairs of parentheses, which corresponds to the *n-th Catalan number*, approximately O(4ⁿ / √n).
  + Each valid combination has length 2n, and the backtracking explores all possible ways to place n opening and n closing parentheses, constrained by validity rules (open ≤ n and close < open).
  + At each step, the algorithm makes up to two recursive calls (add '(' or ')'), but pruning (open < max and close < open) limits the number of valid paths to the Catalan number.
  + Each operation (appending, deleting, or adding to ans) is O(1), but copying StringBuilder to String for each valid combination is O(n).
  + Total time: **O(4ⁿ / √n)**, reflecting the number of valid combinations multiplied by O(n) for string conversion.

**Space Complexity**

* **Space Complexity: O(n)**
  + **Recursion Stack**: The backtracking recursion depth is at most 2n (for a string of length 2n), requiring O(n) space.
  + **StringBuilder**: The cur StringBuilder stores up to 2n characters, using O(n) space.
  + **Output List**: The ans list stores all valid combinations, which is part of the output and not counted as extra space per problem conventions.
  + **Auxiliary Space**: Excluding the output, the space used by cur, open, close, and recursion stack is O(n).
  + Therefore, the auxiliary space complexity is **O(n)**.

**Concise Summary of the Approach**

The solution generates all valid combinations of n pairs of parentheses using backtracking. It maintains a StringBuilder to build combinations and tracks the number of open (open) and close (close) parentheses. At each step, it adds an opening parenthesis if fewer than n are used and a closing parenthesis if there are more open than close. When the string reaches length 2n, it’s added to the result. The approach achieves **O(4ⁿ / √n)** time complexity and **O(n)** auxiliary space complexity, efficiently generating all valid combinations.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Correctness**: The backtracking ensures only valid combinations are generated, respecting the constraints (open ≤ n, close < open).
  + **Clarity**: The recursive structure is intuitive, with clear conditions for adding parentheses, making it interview-friendly.
  + **Efficiency**: Matches the theoretical bound for generating Catalan number combinations.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use StringBuilder instead of String?
    - **Response**: StringBuilder is mutable, allowing O(1) append and delete operations. Using String would require creating new strings (O(n) per operation), increasing time complexity.
  + **Interviewer might ask**: Why check close < open?
    - **Response**: This ensures a closing parenthesis is only added when there’s an unmatched opening parenthesis, maintaining validity (e.g., prevents “)(”).
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: An iterative solution using a stack or queue to simulate backtracking is possible but complex and less intuitive. The recursive backtracking approach is clearer and equally efficient.
* **Edge Cases Handled**:
  + n = 0: Returns empty list (no recursive calls, as cur.length() == 0).
  + n = 1: Generates ["()"] (single valid combination).
  + Large n: Correctly generates all combinations (e.g., n = 3 yields ["((()))","(()())","(())()","()(()","()()()"]).
  + **Note**: Add checks for invalid n (e.g., if (n < 0) return new ArrayList<>();) for robustness.
* **Assumptions**:
  + n is non-negative (per problem constraints).
  + The output should contain all valid combinations of n pairs of parentheses.
  + If these assumptions don’t hold (e.g., negative n), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Dynamic Programming**: Build combinations iteratively by combining smaller valid combinations (e.g., for n, combine i and n-i-1 pairs). Time: O(4ⁿ / √n), Space: O(4ⁿ / √n). Less intuitive and space-heavy due to storing intermediate results.
  + **Catalan Number Formula**: Use combinatorial properties to generate combinations systematically. Time: O(4ⁿ / √n), Space: O(n). Complex and not practical for coding interviews.
  + **Iterative Stack/Queue**: Simulate backtracking with a stack or queue, tracking state (string, open, close). Time: O(4ⁿ / √n), Space: O(n). More complex than recursion.
  + The backtracking approach is preferred for its clarity and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**:
    - Add a check for invalid n:

if (n < 0) return new ArrayList<>();

* + **Optimize StringBuilder**:
    - Pass a char array or string directly to avoid StringBuilder conversions, though the current approach is efficient enough.
  + **Clarity**: Add comments explaining backtracking conditions (e.g., “Add ‘(’ if open < n; add ‘)’ if close < open to ensure validity”).
  + **Edge Case Discussion**: In an interview, mention handling of n = 0, n = 1, and large n to show thoroughness.
  + **Use ArrayDeque for Stack Simulation**: If simulating backtracking iteratively, use ArrayDeque instead of Stack for better performance, though the recursive approach is sufficient here.
* **Note on Backtracking Logic**: The backtracking ensures validity by:
  + Only adding ( if open < n (limits to n pairs).
  + Only adding ) if close < open (ensures every ) has a matching ().
  + Backtracking with cur.deleteCharAt reverts changes to explore all paths.
  + The base case (cur.length() == max \* 2) captures complete, valid combinations.
* Daily Temperatures

Commented Code

class Solution {

*// Method to find the number of days until a warmer day for each day*

public int[] dailyTemperatures(int[] temperatures) {

int n = temperatures.length; *// Length of input array*

int[] answer = new int[n]; *// Array to store days until warmer temperature*

Stack<Integer> stack = new Stack<>(); *// Stack to store indices of days*

*// Iterate through each day's temperature*

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

*// While stack is not empty and current temperature is warmer than the day at stack's top*

while (!stack.isEmpty() && temperatures[i] > temperatures[stack.peek()]) {

int index = stack.pop(); *// Pop the index of the cooler day*

answer[index] = i - index; *// Calculate days until warmer day*

}

stack.push(i); *// Push current day's index onto stack*

}

*// Return result array (unprocessed days remain 0)*

return answer;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the array of length n once.
  + Each index is pushed onto the stack exactly once (O(1) per push).
  + Each index is popped at most once, as a warmer day resolves it, and it’s never pushed again (O(1) per pop).
  + The while loop’s pops are amortized across the n iterations, as the total number of pops cannot exceed the number of pushes (n).
  + All operations (comparisons, array access, stack operations) are O(1).
  + Therefore, the total time complexity is **O(n)**, where n is the length of the temperatures array.

**Space Complexity**

* **Space Complexity: O(n)**
  + The answer array stores n integers, but this is part of the output and not counted as extra space per problem conventions.
  + The stack stores indices, and in the worst case (e.g., strictly decreasing temperatures like [5,4,3,2,1]), it holds up to n indices, requiring O(n) space.
  + Other variables (n, i, index) use O(1) space.
  + Therefore, the extra space complexity is **O(n)** due to the stack.

**Concise Summary of the Approach**

The solution finds, for each day, the number of days until a warmer day using a monotonic stack. It iterates through the temperatures array, maintaining a stack of indices for days with decreasing temperatures. When a warmer day is found, it pops indices from the stack, calculating the difference in days (i - index) and storing it in the answer array. Unprocessed days (no warmer day) remain 0. The approach achieves **O(n)** time complexity and **O(n)** space complexity, efficiently resolving each day in a single pass.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each temperature must be processed, and the monotonic stack ensures each index is handled at most twice (push and pop).
  + **Clarity**: The stack-based approach is intuitive for finding the next greater element, making it interview-friendly.
  + **Correctness**: Correctly handles cases where no warmer day exists (returns 0) and maintains order via the stack.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a monotonic stack?
    - **Response**: The stack maintains indices of days with decreasing temperatures, ensuring that when a warmer day is found, all cooler days can be resolved efficiently in order, avoiding redundant comparisons.
  + **Interviewer might ask**: Can you solve it without a stack?
    - **Response**: A brute-force approach checking each day against future days takes O(n²) time. The stack reduces this to O(n) by resolving each day only when a warmer day is found, leveraging monotonicity.
  + **Interviewer might ask**: How does this handle duplicates or equal temperatures?
    - **Response**: The condition temperatures[i] > temperatures[stack.peek()] ensures only strictly warmer days trigger pops, so equal temperatures stay on the stack, which is correct per the problem (find the next *warmer* day).
* **Edge Cases Handled**:
  + Empty array: Not applicable (problem guarantees n ≥ 1).
  + Single day: Returns [0] (no warmer day possible).
  + Decreasing temperatures (e.g., [5,4,3,2,1]): Returns [0,0,0,0,0] (no warmer days, stack never pops).
  + Increasing temperatures (e.g., [1,2,3,4,5]): Returns [1,1,1,1,0] (each day resolved by next).
  + **Note**: Add input validation (e.g., if (temperatures == null || temperatures.length == 0) return new int[0];) for robustness.
* **Assumptions**:
  + The input temperatures is non-null, contains at least one element, and has valid integers (per problem constraints: 30 ≤ temperatures[i] ≤ 100).
  + The problem asks for the next strictly warmer day.
  + If these assumptions don’t hold (e.g., invalid temperatures), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Brute Force**: For each day, scan forward to find the next warmer day. Time: O(n²), Space: O(1) excluding output. Inefficient for large n.
  + **Array-Based with Backward Scan**: Iterate backward, keeping track of the next warmer day using an array or pointers. Time: O(n) with optimizations, Space: O(n). Less intuitive than the stack approach.
  + The monotonic stack approach is preferred for its O(n) time, clarity, and elegance.
* **Suggestions for Improvement**:
  + **Input Validation**:
    - Add checks for null or empty input:

if (temperatures == null || temperatures.length == 0) return new int[0];

* + - Validate temperature ranges if required (e.g., if (temperatures[i] < 30 || temperatures[i] > 100)).
  + **Use ArrayDeque**: Replace Stack with ArrayDeque for better performance:

Deque<Integer> stack = new ArrayDeque<>();

Stack is a legacy class; ArrayDeque is more efficient.

* + **Clarity**: Add comments explaining the monotonic stack (e.g., “Stack stores indices of days with decreasing temperatures”).
  + **Edge Case Discussion**: In an interview, mention handling of decreasing sequences, single elements, and duplicates to show thoroughness.
  + **Optimization**: The current approach is optimal, but you could explore a backward scan to avoid the stack in some cases, though it’s less clear.
* **Note on Monotonic Stack**: The stack maintains indices of days with decreasing temperatures (a monotonically decreasing stack). When a warmer day is found, it resolves all cooler days at the top, ensuring each index is processed efficiently. The answer for unpopped indices remains 0, as no warmer day exists.
* Car Fleet

Commented Code

class Solution {

*// Method to determine the number of car fleets reaching a target*

public int carFleet(int target, int[] position, int[] speed) {

int n = position.length; *// Number of cars*

double[][] cars = new double[n][2]; *// Array to store position and time to target*

*// Populate cars array with position and time to reach target*

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

cars[i][0] = position[i]; *// Store position*

cars[i][1] = (double) (target - position[i]) / speed[i]; *// Calculate time to target*

}

*// Sort cars by position in descending order (farthest to closest)*

Arrays.sort(cars, (a, b) -> Double.compare(b[0], a[0]));

int count = 0; *// Number of fleets*

double prevTime = 0; *// Time of the last fleet's lead car*

*// Calculate number of fleets by comparing times*

for (double[] car : cars) {

if (car[1] > prevTime) { *// If current car arrives later, it forms a new fleet*

count++;

prevTime = car[1]; *// Update time of the new fleet*

}

*// If car[1] <= prevTime, car joins the existing fleet (caught up)*

}

return count; *// Return total number of fleets*

}

}

**Time Complexity**

* **Time Complexity: O(n log n)**
  + **Array Population**: Filling the cars array with positions and times takes O(n), where n is the number of cars. Each iteration performs O(1) operations (array assignment and division).
  + **Sorting**: Sorting the cars array by position in descending order using Arrays.sort with a comparator takes O(n log n), as it uses a comparison-based sort (Timsort in ).
  + **Fleet Calculation**: Iterating through the sorted array to count fleets takes O(n), with O(1) operations per car (comparison and assignment).
  + Total time: O(n) for population + O(n log n) for sorting + O(n) for fleet counting = **O(n log n)**, dominated by the sorting step.

**Space Complexity**

* **Space Complexity: O(n)**
  + The cars array stores n pairs of doubles (position and time), requiring O(n) space.
  + Other variables (n, count, prevTime) use O(1) space.
  + The output (return value) is a single integer, using O(1) space.
  + Therefore, the space complexity is **O(n)** due to the cars array.

**Concise Summary of the Approach**

The solution determines the number of car fleets reaching a target by calculating each car’s arrival time and sorting by position. It creates a cars array with each car’s position and time to reach the target ((target - position) / speed). Cars are sorted by position in descending order (farthest to closest). Iterating through the sorted cars, a new fleet is counted if a car’s arrival time exceeds the previous fleet’s time; otherwise, it joins the existing fleet. The approach achieves **O(n log n)** time complexity due to sorting and **O(n)** space complexity, efficiently computing the number of fleets.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log n) time is optimal due to the need to sort by position, and the single-pass fleet counting is efficient.
  + **Clarity**: The approach is intuitive, using sorting to process cars from farthest to closest and a simple check for fleet formation, making it interview-friendly.
  + **Correctness**: Correctly handles cars catching up to form fleets based on arrival times.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why sort by position in descending order?
    - **Response**: Sorting by position (farthest to closest) ensures we process cars in order of their distance from the target. This allows us to check if a car arrives later than the previous fleet, forming a new fleet, or catches up to join an existing one.
  + **Interviewer might ask**: Why compare arrival times (car[1] > prevTime)?
    - **Response**: A car forms a new fleet only if it arrives later than the previous fleet’s lead car. If its arrival time is less than or equal, it catches up and joins the fleet, as it can’t pass the lead car.
  + **Interviewer might ask**: Can you solve it without sorting?
    - **Response**: Without sorting, we’d need to compare all pairs of cars to determine which form fleets, leading to O(n²) time. Sorting by position reduces this to O(n log n) by processing cars in order.
* **Edge Cases Handled**:
  + Empty array (n = 0): Returns 0 (no fleets, as count remains 0).
  + Single car (n = 1): Returns 1 (one fleet, as car[1] > 0).
  + All cars at same position: Returns 1 if speeds ensure they arrive together (times are equal or close).
  + Cars with same speed: Handled by comparing arrival times.
  + **Note**: Add input validation (e.g., if (position == null || speed == null || position.length != speed.length) return 0;) for robustness.
* **Assumptions**:
  + position and speed arrays are non-null, have equal length n, and contain valid integers.
  + target is positive, and speeds are positive (per problem constraints: 1 ≤ speed[i] ≤ 10⁶).
  + Cars cannot pass each other, forming fleets when they meet.
  + If these assumptions don’t hold (e.g., zero speed), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Brute Force**: For each car, compute its arrival time and check if it forms a new fleet by comparing with all later cars. Time: O(n²), Space: O(1) excluding output. Inefficient for large n.
  + **Stack-Based**: Use a stack to track arrival times, similar to the provided approach but maintaining times explicitly. Time: O(n log n) if sorting is used, Space: O(n). Similar to the current approach but less direct.
  + The sorting-based approach is preferred for its clarity and efficiency.
* **Suggestions for Improvement**:
  + **Input Validation**:
    - Add checks for invalid inputs:

if (position == null || speed == null || position.length != speed.length) return 0;

if (n == 0) return 0;

* + - Validate positive speeds to avoid division by zero (though guaranteed by constraints).
  + **Optimize Sorting**: Use integer positions directly if possible, avoiding double conversion:

int[][] cars = new int[n][2];

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

cars[i][0] = position[i];

cars[i][1] = (target - position[i]) / speed[i]; *// Store time as needed*

}

However, doubles are necessary for precise time calculations.

* + **Clarity**: Add comments explaining fleet formation (e.g., “Car forms new fleet if it arrives later than the previous fleet”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single cars, and same-position cases to show thoroughness.
  + **Optimization**: Consider a stack-based approach to avoid explicit sorting in rare cases, but sorting is typically necessary for clarity and correctness.
* **Note on Fleet Formation**: Cars form a fleet if they arrive at the target at the same time or catch up to a slower car ahead. Sorting by position (descending) ensures we process cars from farthest to closest, and comparing arrival times (car[1] > prevTime) identifies new fleets when a car cannot catch up to the previous fleet.
* Largest Rectangle In Histogram

Commented Code

class Solution {

*// Method to find the largest rectangle area in a histogram*

public int largestRectangleArea(int[] heights) {

int maxArea = 0; *// Track maximum rectangle area*

Stack<Integer> stack = new Stack<>(); *// Stack to store indices of bars*

int n = heights.length; *// Length of heights array*

*// Iterate through all bars, including a dummy bar of height 0 at the end*

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

int currentHeight = (i == n) ? 0 : heights[i]; *// Use 0 for end to clear stack*

*// Pop bars taller than current height to calculate areas*

while (!stack.isEmpty() && currentHeight < heights[stack.peek()]) {

int height = heights[stack.pop()]; *// Height of popped bar*

*// Width is from current index to the previous stack index (or start if empty)*

int width = stack.isEmpty() ? i : i - stack.peek() - 1;

maxArea = Math.max(maxArea, height \* width); *// Update max area*

}

stack.push(i); *// Push current index onto stack*

}

return maxArea; *// Return the largest area found*

}

}

**Commented Code**

class Solution {

*// Method to find the largest rectangle area in a histogram*

public int largestRectangleArea(int[] heights) {

int maxArea = 0; *// Track maximum rectangle area*

Stack<Integer> stack = new Stack<>(); *// Stack to store indices of bars*

int n = heights.length; *// Length of heights array*

*// Iterate through all bars, including a dummy bar of height 0 at the end*

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

int currentHeight = (i == n) ? 0 : heights[i]; *// Use 0 for end to clear stack*

*// Pop bars taller than current height to calculate areas*

while (!stack.isEmpty() && currentHeight < heights[stack.peek()]) {

int height = heights[stack.pop()]; *// Height of popped bar*

*// Width is from current index to the previous stack index (or start if empty)*

int width = stack.isEmpty() ? i : i - stack.peek() - 1;

maxArea = Math.max(maxArea, height \* width); *// Update max area*

}

stack.push(i); *// Push current index onto stack*

}

return maxArea; *// Return the largest area found*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the array of length n once, plus one extra iteration for the dummy bar (i.e., i <= n).
  + Each index is pushed onto the stack exactly once (O(1) per push).
  + Each index is popped at most once, as a shorter bar triggers the pop, and the index is not pushed again (O(1) per pop).
  + The while loop’s pops are amortized across the n iterations, as the total number of pops cannot exceed the number of pushes (n).
  + All operations (array access, stack operations, comparisons) are O(1).
  + Therefore, the total time complexity is **O(n)**, where n is the length of the heights array.

**Space Complexity**

* **Space Complexity: O(n)**
  + The stack stores indices of bars, and in the worst case (e.g., strictly increasing heights like [1,2,3,4,5]), it holds up to n indices, requiring O(n) space.
  + Other variables (maxArea, n, i, currentHeight, height, width) use O(1) space.
  + Therefore, the space complexity is **O(n)** due to the stack.

**Concise Summary of the Approach**

The solution finds the largest rectangle in a histogram using a monotonic stack. It iterates through the heights array, maintaining a stack of indices for bars with increasing heights. When a shorter bar is encountered (or a dummy bar of height 0 at the end), taller bars are popped, and their rectangle areas are calculated using their height and the width between the current index and the previous stack index. The maximum area is updated accordingly. The approach achieves **O(n)** time complexity and **O(n)** space complexity, efficiently computing the largest rectangle by processing each bar in a single pass.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each bar must be processed, and the monotonic stack ensures each bar is handled at most twice (push and pop).
  + **Clarity**: The stack-based approach is a standard solution for histogram problems, making it interview-friendly once explained.
  + **Correctness**: Correctly handles all bars by using a dummy bar (height 0) to process remaining stack elements.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a dummy bar of height 0 at the end?
    - **Response**: The dummy bar ensures all bars in the stack are processed by triggering pops for any remaining bars, as it’s shorter than all actual heights. This simplifies the logic and avoids a separate loop to clear the stack.
  + **Interviewer might ask**: How is the width calculated?
    - **Response**: For a popped bar, the width is the difference between the current index (i) and the index of the next bar on the stack (stack.peek()), minus 1, as the popped bar’s rectangle extends to just before the current bar. If the stack is empty, the width is i (from start to current index).
  + **Interviewer might ask**: Can you solve it without a stack?
    - **Response**: A brute-force approach checking all possible rectangles for each bar takes O(n²) time. A divide-and-conquer approach (using segment trees or similar) can achieve O(n log n), but the stack-based approach is simpler and achieves O(n).
* **Edge Cases Handled**:
  + Empty array (n = 0): Returns 0 (no iterations, maxArea remains 0).
  + Single bar (e.g., [5]): Returns correct area (processed with dummy bar, area = 5 \* 1).
  + All equal heights (e.g., [2,2,2]): Returns largest rectangle (e.g., 2 \* 3 = 6).
  + Increasing/decreasing heights: Correctly handled by stack pops and width calculations.
  + **Note**: Add input validation (e.g., if (heights == null || heights.length == 0) return 0;) for robustness.
* **Assumptions**:
  + The input heights array is non-null and contains non-negative integers (per problem constraints).
  + The array represents bar heights in a histogram.
  + If these assumptions don’t hold (e.g., negative heights), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Brute Force**: For each bar, compute the maximum rectangle by checking left and right boundaries where heights are ≥ current height. Time: O(n²), Space: O(1). Inefficient for large n.
  + **Divide and Conquer**: Find the minimum height in a range, calculate its area, and recurse on left and right subranges. Time: O(n log n) with segment tree optimization, Space: O(n). More complex and slower than the stack approach.
  + The monotonic stack approach is preferred for its O(n) time and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**:
    - Add checks for null or empty input:

if (heights == null || heights.length == 0) return 0;

* + - Validate non-negative heights if required (though guaranteed by constraints).
  + **Use ArrayDeque**: Replace Stack with ArrayDeque for better performance:

Deque<Integer> stack = new ArrayDeque<>();

Stack is a legacy class; ArrayDeque is more efficient.

* + **Clarity**: Add comments explaining the monotonic stack and width calculation (e.g., “Stack maintains indices of increasing heights; width spans to next shorter bar”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single bars, and equal heights to show thoroughness.
  + **Optimization**: The current approach is optimal, but you could precompute the array length or use a sentinel value in the array to avoid the i == n check, though this adds complexity.
* **Note on Monotonic Stack**: The stack maintains indices of bars with increasing heights (monotonically increasing stack). When a shorter bar is encountered, taller bars are popped, and their areas are calculated using the width between the current index and the next stack index (or start if empty). The dummy bar (height 0) ensures all bars are processed.

5. Binary Search (7 problems)

* Binary Search

Commented Code

class Solution {

*// Method to find the index of target in a sorted array using binary search*

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

int left = 0, right = nums.length - 1; *// Initialize two pointers: left at start, right at end*

*// Continue while search space is valid*

while (left <= right) {

int mid = left + (right - left) / 2; *// Calculate middle index to avoid overflow*

*// If target is found at mid, return its index*

if (nums[mid] == target) {

return mid;

} else if (nums[mid] < target) {

left = mid + 1; *// Target is in right half, exclude mid*

} else {

right = mid - 1; *// Target is in left half, exclude mid*

}

}

*// Return -1 if target is not found*

return -1;

}

}

**Time Complexity**

* **Time Complexity: O(log n)**
  + The algorithm uses binary search, which divides the search space in half with each iteration.
  + Starting with n elements, the number of iterations is logarithmic, specifically log₂(n).
  + Each iteration performs O(1) operations: calculating mid, comparing nums[mid] with target, and updating pointers.
  + Therefore, the total time complexity is **O(log n)**, where n is the length of the input array.

**Space Complexity**

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

**Concise Summary of the Approach**

The solution finds the index of a target value in a sorted array using binary search. It initializes two pointers, left at the start and right at the end, and iteratively computes the middle index. If the middle element equals the target, it returns its index. If the middle element is less than the target, it searches the right half; otherwise, it searches the left half. If the target is not found, it returns -1. The approach achieves **O(log n)** time complexity and **O(1)** space complexity, leveraging the sorted array for efficient searching.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(log n) time is optimal for searching in a sorted array, far better than O(n) linear search.
  + **Clarity**: Binary search is a standard algorithm, easy to explain and implement, making it ideal for interviews.
  + **Correctness**: Handles all cases correctly, assuming the input is sorted, and returns -1 when the target is absent.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use left + (right - left) / 2 instead of (left + right) / 2?
    - **Response**: The formula left + (right - left) / 2 prevents integer overflow when left and right are large. It’s equivalent but safer for large arrays.
  + **Interviewer might ask**: What if the array is rotated (e.g., Rotated Sorted Array)?
    - **Response**: For a rotated sorted array, modify the logic to determine which half is sorted and whether the target lies in it, still achieving O(log n) time. The current code assumes a fully sorted array.
  + **Interviewer might ask**: How does this handle duplicates?
    - **Response**: The code returns the index of the first occurrence of the target it finds. If the problem requires finding the first or last occurrence, additional logic (e.g., continuing search left or right) would be needed.
* **Edge Cases Handled**:
  + Empty array: Returns -1 (implicitly, as left > right initially).
  + Single element: Correctly checks if it’s the target.
  + Target not present: Returns -1 after search space is exhausted.
  + Target at start/end: Handled correctly by including left <= right.
  + **Note**: Add checks for null or empty arrays (e.g., if (nums == null || nums.length == 0) return -1;) for robustness.
* **Assumptions**:
  + The input array is sorted in ascending order and non-null.
  + The target is an integer that may or may not exist in the array.
  + If these assumptions don’t hold (e.g., unsorted array, null input), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Linear Search**: Scan the array sequentially to find the target. Time: O(n), Space: O(1). Inefficient for large sorted arrays.
  + **Recursive Binary Search**: Implement binary search recursively. Time: O(log n), Space: O(log n) due to call stack. Less practical than iterative for large inputs.
  + The iterative binary search is preferred for its simplicity and O(1) space.
* **Suggestions for Improvement**:
  + **Input Validation**: Add explicit checks for null or empty arrays (e.g., if (nums == null || nums.length == 0) return -1;) to handle edge cases robustly.
  + **Clarity**: Add comments explaining the binary search logic (e.g., “Divide search space in half, exclude mid based on comparison”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single elements, and the importance of sorted input to show thoroughness.
  + **Optimization**: The current approach is optimal, but note the use of left + (right - left) / 2 to avoid overflow, which is a best practice.
* Search a 2D Matrix

Commented Code

class Solution {

*// Method to search for a target value in a sorted 2D matrix*

public boolean searchMatrix(int[][] matrix, int target) {

int m = matrix.length; *// Number of rows*

int n = matrix[0].length; *// Number of columns*

int left = 0, right = m \* n - 1; *// Initialize pointers for binary search*

*// Perform binary search treating the matrix as a flattened sorted array*

while (left <= right) {

int mid = left + (right - left) / 2; *// Calculate middle index*

*// Convert 1D index to 2D coordinates (row = mid / n, col = mid % n)*

int mid\_val = matrix[mid / n][mid % n];

*// If target is found, return true*

if (mid\_val == target)

return true;

*// If middle value is less than target, search right half*

else if (mid\_val < target)

left = mid + 1;

*// If middle value is greater than target, search left half*

else

right = mid - 1;

}

*// Return false if target is not found*

return false;

}

}

**Time Complexity**

* **Time Complexity: O(log(m \* n))**
  + The algorithm uses binary search on a virtual flattened array of size m \* n (where m is the number of rows and n is the number of columns).
  + Each iteration of binary search divides the search space in half, requiring log₂(m \* n) iterations.
  + Operations within the loop (calculating mid, accessing matrix[mid / n][mid % n], comparisons) are O(1).
  + Therefore, the total time complexity is **O(log(m \* n))**, where m is the number of rows and n is the number of columns.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses only a few variables (m, n, left, right, mid, mid\_val) regardless of input size.
  + No additional data structures are used, so the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution searches for a target value in a sorted 2D matrix (sorted row-wise and column-wise) using binary search. It treats the matrix as a flattened sorted array of size m \* n, mapping 1D indices to 2D coordinates (row = mid / n, col = mid % n). The binary search compares the middle element to the target, adjusting the search space (left or right) accordingly. If the target is found, it returns true; otherwise, it returns false. The approach achieves **O(log(m \* n))** time complexity and **O(1)** space complexity, leveraging the sorted property for efficiency.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(log(m \* n)) time is optimal for searching a sorted matrix, matching the performance of binary search on a sorted array.
  + **Clarity**: Treating the matrix as a flattened array simplifies the logic, and the index-to-coordinate mapping is easy to explain.
  + **Correctness**: Works correctly for all cases, assuming the matrix is sorted as specified (each row sorted, first element of each row greater than the last element of the previous row).
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why treat the matrix as a flattened array?
    - **Response**: The matrix’s sorted property (each row sorted, and rows ordered such that matrix[i][0] > matrix[i-1][n-1]) ensures the flattened array is sorted, allowing binary search to work efficiently in O(log(m \* n)) time.
  + **Interviewer might ask**: Can you search row by row?
    - **Response**: Searching each row with binary search takes O(m \* log n) time, which is less efficient when m is large. The flattened approach is optimal since it considers the entire matrix in one binary search.
  + **Interviewer might ask**: How does this handle duplicates or edge cases?
    - **Response**: The code returns true for the first occurrence of the target. Edge cases like empty matrices or invalid dimensions should be checked explicitly for robustness.
* **Edge Cases Handled**:
  + Target not present: Returns false after search space is exhausted.
  + Single element matrix: Correctly checks the single element.
  + Target at boundaries (first/last element): Handled by binary search including left <= right.
  + **Note**: Add checks for null or empty matrices (e.g., if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return false;) for robustness.
* **Assumptions**:
  + The matrix is non-null, non-empty, and has at least one row and column.
  + The matrix is sorted (each row is sorted, and matrix[i][0] > matrix[i-1][n-1]).
  + All rows have the same number of columns (rectangular matrix).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Row-by-Row Binary Search**: Perform binary search on each row. Time: O(m \* log n), Space: O(1). Less efficient for large m.
  + **Linear Search from Top-Right**: Start from the top-right corner and move left (if target is smaller) or down (if target is larger), leveraging the sorted property. Time: O(m + n), Space: O(1). Less efficient for large matrices but simpler in some cases.
  + The flattened binary search is preferred for its optimal O(log(m \* n)) time and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add explicit checks for null or empty matrices (e.g., if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return false;) to handle edge cases robustly.
  + **Clarity**: Add comments explaining the index mapping (e.g., “Map 1D index to 2D: row = mid / n, col = mid % n”).
  + **Edge Case Discussion**: In an interview, mention handling of empty matrices, single-element matrices, and the sorted property to show thoroughness.
  + **Optimization**: The current approach is optimal, but note the use of left + (right - left) / 2 to avoid integer overflow, a best practice.
* **Note on Index Mapping**: The mapping row = mid / n, col = mid % n converts a 1D index into 2D coordinates, allowing the matrix to be treated as a sorted array. This is efficient because the sorted property ensures matrix[i][j] ≤ matrix[i][j+1] and matrix[i][n-1] < matrix[i+1][0], preserving order in the flattened view.
* Koko Eating Bananas

Commented Code

class Solution {

*// Method to find the minimum eating speed to finish all piles within h hours*

public static int minEatingSpeed(int[] piles, int h) {

int left = 1, right = 1; *// Initialize binary search bounds*

*// Set right to the maximum pile size*

for (int pile : piles) {

right = Math.max(right, pile);

}

*// Binary search to find the minimum speed*

while (left < right) {

int mid = left + (right - left) / 2; *// Calculate middle speed*

*// If can finish at this speed, try a lower speed*

if (canFinish(piles, mid, h)) {

right = mid; *// Include mid in search space*

} else {

left = mid + 1; *// Exclude mid, need higher speed*

}

}

*// Return the minimum speed*

return left;

}

*// Helper method to check if piles can be eaten at given speed within h hours*

private static boolean canFinish(int[] piles, int speed, int h) {

int hours = 0;

*// Calculate total hours needed at given speed*

for (int pile : piles) {

hours += Math.ceil((double) pile / speed); *// Ceiling of pile/speed*

}

return hours <= h; *// Return true if hours are within limit*

}

}

**Time Complexity**

* **Time Complexity: O(n \* log m)**
  + **Initialization**: Finding the maximum pile size to set right takes O(n), where n is the length of piles.
  + **Binary Search**: The search space for speed ranges from 1 to m, where m is the maximum pile size. Binary search performs O(log m) iterations.
  + **canFinish**: For each speed tested, the canFinish method iterates through n piles, performing O(1) operations per pile (division and ceiling). Thus, each call to canFinish takes O(n).
  + Total time: O(n) for initialization + O(log m) \* O(n) for binary search iterations = **O(n \* log m)**, where n is the number of piles and m is the maximum pile size.

**Space Complexity**

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

**Concise Summary of the Approach**

The solution finds the minimum eating speed for Koko to finish all banana piles within h hours using binary search. It sets the search range from 1 (minimum possible speed) to the maximum pile size. For each speed, it checks if all piles can be eaten within h hours by summing the ceiling of pile/speed for each pile. If feasible, it tries a lower speed; otherwise, it increases the speed. The approach achieves **O(n \* log m)** time complexity and **O(1)** space complexity, efficiently finding the minimum speed.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n \* log m) is optimal, as binary search narrows down the speed efficiently, and each check is linear in the number of piles.
  + **Clarity**: The binary search approach is intuitive for finding a minimum value under a constraint, making it interview-friendly.
  + **Correctness**: Correctly handles the ceiling calculation and ensures the minimum speed is found.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use binary search instead of linear search?
    - **Response**: Linearly testing speeds from 1 to max(piles) would take O(m \* n) time, where m is the maximum pile size (potentially very large, e.g., 10⁹). Binary search reduces the speed testing to O(log m), making it much faster.
  + **Interviewer might ask**: Why set right to the maximum pile size?
    - **Response**: The maximum pile size is the highest possible speed needed (if Koko must eat the largest pile in one hour). Setting right to max(piles) ensures the search space covers all feasible speeds.
  + **Interviewer might ask**: How does this handle edge cases like large h?
    - **Response**: If h is large (e.g., h ≥ sum(piles)), the minimum speed is 1, as Koko can eat one banana per hour. The code correctly returns 1 in such cases.
* **Edge Cases Handled**:
  + Empty or null array: Not explicitly checked, but problem typically assumes valid input.
  + Single pile: Works correctly, as canFinish computes hours for one pile.
  + Large h (e.g., h ≥ sum(piles)): Returns 1, as speed of 1 is sufficient.
  + Large pile sizes: Handled by binary search up to max(piles).
  + **Note**: Add checks for null or empty arrays (e.g., if (piles == null || piles.length == 0) return 0;) for robustness.
* **Assumptions**:
  + The input array piles is non-null, non-empty, and contains positive integers.
  + h is a positive integer, and h ≥ piles.length (as Koko needs at least one hour per pile).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Linear Search**: Test speeds from 1 to max(piles) sequentially, checking if each speed allows finishing within h hours. Time: O(n \* m), Space: O(1). Inefficient for large m.
  + **Brute Force with Precomputation**: Precompute hours for a range of speeds, but still requires testing many speeds, similar to linear search. Inefficient.
  + The binary search approach is preferred for its efficiency and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty arrays or invalid h (e.g., if (piles == null || piles.length == 0 || h < piles.length) return 0;) to handle edge cases robustly.
  + **Clarity**: Add comments explaining the binary search logic (e.g., “Search for minimum speed where hours ≤ h”).
  + **Edge Case Discussion**: In an interview, mention handling of single piles, large h, and large pile sizes to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that Math.ceil((double) pile / speed) can be replaced with (pile + speed - 1) / speed to avoid floating-point arithmetic, though the performance impact is minimal.
* **Note on Binary Search**: The binary search finds the smallest speed where canFinish returns true. Since canFinish is monotonic (higher speeds always reduce hours), binary search is ideal. Setting right = mid (instead of mid - 1) ensures the minimum speed is included in the search space.
* Find Minimum In Rotated Sorted Array

Commented Code

class Solution {

*// Method to find the minimum element in a rotated sorted array*

public int findMin(int[] nums) {

int left = 0; *// Initialize left pointer at start*

int right = nums.length - 1; *// Initialize right pointer at end*

int ans = nums[0]; *// Initialize answer with first element*

*// Handle single-element array*

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, take left as 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 middle 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 the minimum element*

return ans;

}

}

**Time Complexity**

* **Time Complexity: O(log n)**
  + The algorithm uses binary search, dividing the search space in half each iteration.
  + The array of length n requires at most log₂(n) iterations to find the minimum.
  + Each iteration performs O(1) operations: comparisons, calculating mid, and updating ans or pointers.
  + The single-element check is O(1).
  + Therefore, the total time complexity is **O(log n)**, where n is the length of the input array.

**Space Complexity**

* **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, so the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution finds the minimum element in a rotated sorted array using binary search. It initializes left and right pointers and tracks the minimum in ans. If the array is sorted (nums[left] < nums[right]), the minimum is nums[left]. Otherwise, it computes the middle index, updates ans with the minimum of the current ans and nums[mid], and adjusts the search space: if nums[mid] > nums[right], the minimum is in the right half; otherwise, it’s in the left half (including mid). The approach achieves **O(log n)** time complexity and **O(1)** space complexity, leveraging the rotated sorted property.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(log n) time is optimal for a rotated sorted array, matching standard binary search.
  + **Clarity**: The binary search logic is straightforward and easy to explain, ideal for interviews.
  + **Correctness**: Correctly finds the minimum by leveraging the sorted and rotated properties.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why check nums[left] < nums[right]?
    - **Response**: If nums[left] < nums[right], the subarray from left to right is sorted, so the minimum is nums[left]. This optimizes the case where no rotation or partial sorting occurs.
  + **Interviewer might ask**: Why compare nums[mid] with nums[right]?
    - **Response**: In a rotated sorted array, the minimum lies at the rotation point. If nums[mid] > nums[right], the rotation point (and minimum) is in the right half (mid + 1 to right). Otherwise, it’s in the left half (left to mid).
  + **Interviewer might ask**: How does this handle duplicates?
    - **Response**: Duplicates (e.g., [2,2,2,0,1]) are not handled correctly by this code, as nums[mid] > nums[right] or nums[mid] <= nums[right] may not disambiguate the rotation point. A modified version is needed for duplicates (see suggestions).
* **Edge Cases Handled**:
  + Single element: Returns nums[0] (explicitly checked).
  + No rotation (sorted array): Returns nums[0] via nums[left] < nums[right] check.
  + Full rotation: Correctly finds minimum at rotation point.
  + **Note**: Does not handle duplicates correctly (e.g., [2,2,2,0,1]). Add checks for null/empty arrays for robustness.
* **Assumptions**:
  + The array is non-null, non-empty, and sorted in ascending order with a single rotation.
  + No duplicates (per the standard problem; if duplicates are possible, the code needs adjustment).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Linear Search**: Scan the array to find the minimum. Time: O(n), Space: O(1). Inefficient for large arrays.
  + **Binary Search with Duplicate Handling**: For arrays with duplicates (e.g., [2,2,2,0,1]), add a check when nums[mid] == nums[right], decrementing right to handle ambiguity. Time: O(log n) average, O(n) worst case with duplicates, Space: O(1).
  + The binary search approach is preferred for its efficiency, but needs modification for duplicates.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty arrays (e.g., if (nums == null || nums.length == 0) return -1;).
  + **Handle Duplicates**: Modify the code to handle duplicates by adding a case for nums[mid] == nums[right], e.g., right-- to reduce the search space, as the minimum could be at right or earlier.
  + **Clarity**: Add comments explaining the rotation point logic (e.g., “Minimum lies at rotation point; compare mid with right to decide half”).
  + **Edge Case Discussion**: In an interview, mention handling of single elements, no rotation, and the limitation with duplicates to show thoroughness.
  + **Optimization**: The current approach is optimal for no duplicates, but note that ans = nums[0] is redundant in some cases, as the minimum is always found at the rotation point or nums[left] in sorted cases.
* **Note on Rotation Point**: The minimum in a rotated sorted array is at the rotation point, where nums[i] > nums[i+1]. The binary search identifies this point by comparing nums[mid] with nums[right], as the right half contains the rotation if nums[mid] > nums[right]. The ans variable ensures the minimum is tracked, though checking nums[left] in the sorted case optimizes early termination.
* Search In Rotated Sorted Array

Commented Code

class Solution {

*// Method to search for a target in a rotated sorted array*

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

int left = 0; *// Initialize left pointer at start*

int right = nums.length - 1; *// Initialize right pointer at end*

*// Binary search to find the target*

while (left <= right) {

int mid = (left + right) / 2; *// Calculate middle index*

*// If target is found at mid, return its 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 range, search right half*

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

left = mid + 1;

} else {

*// Target is in sorted left half, search left half*

right = mid - 1;

}

} else {

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

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

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

right = mid - 1;

} else {

*// Target is in sorted right half, search right half*

left = mid + 1;

}

}

}

*// Return -1 if target is not found*

return -1;

}

}

**Time Complexity**

* **Time Complexity: O(log n)**
  + The algorithm uses binary search, dividing the search space in half each iteration.
  + The array of length n requires at most log₂(n) iterations to find the target or determine it’s not present.
  + Each iteration performs O(1) operations: calculating mid, comparing nums[mid], nums[left], and nums[right] with target, and updating pointers.
  + Therefore, the total time complexity is **O(log n)**, where n is the length of the input array.

**Space Complexity**

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

**Concise Summary of the Approach**

The solution searches for a target in a rotated sorted array using binary search. It initializes left and right pointers and computes the middle index. If the target is at mid, it returns the index. It determines which half (left to mid or mid to right) is sorted by comparing nums[left] with nums[mid]. If the left half is sorted, it checks if the target lies within its range; otherwise, it searches the right half. If the right half is sorted, it applies similar logic. If the target is not found, it returns -1. The approach achieves **O(log n)** time complexity and **O(1)** space complexity, leveraging the sorted property despite rotation.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(log n) time is optimal for searching in a rotated sorted array, matching standard binary search.
  + **Clarity**: The logic of identifying the sorted half and narrowing the search is intuitive once explained, making it interview-friendly.
  + **Correctness**: Handles the rotation point correctly, assuming no duplicates (per standard problem constraints).
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why check nums[left] <= nums[mid]?
    - **Response**: This determines if the left half (left to mid) is sorted. If nums[left] <= nums[mid], the left half has no rotation point, so we can check if the target lies in this sorted range. Otherwise, the right half is sorted.
  + **Interviewer might ask**: How would you handle duplicates?
    - **Response**: With duplicates (e.g., [2,2,2,0,1]), nums[left] <= nums[mid] may not clearly identify the sorted half. You’d need to handle cases where nums[left] == nums[mid] == nums[right] by decrementing right to reduce ambiguity, potentially degrading to O(n) in the worst case.
  + **Interviewer might ask**: Why use (left + right) / 2 instead of left + (right - left) / 2?
    - **Response**: Both are equivalent, but left + (right - left) / 2 prevents integer overflow for large arrays. The current code is safe for typical inputs but could be improved with the overflow-safe formula.
* **Edge Cases Handled**:
  + Empty array: Returns -1 (implicitly, as left > right initially).
  + Single element: Correctly checks if it’s the target.
  + No rotation (sorted array): Works as standard binary search.
  + Target not present: Returns -1 after search space is exhausted.
  + Target at rotation point or boundaries: Handled correctly by checking mid and sorted halves.
  + **Note**: Add checks for null or empty arrays (e.g., if (nums == null || nums.length == 0) return -1;) for robustness.
* **Assumptions**:
  + The array is non-null, non-empty, and sorted in ascending order with a single rotation (or no rotation).
  + No duplicates (per standard problem constraints; if duplicates are allowed, the code needs adjustment).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Linear Search**: Scan the array to find the target. Time: O(n), Space: O(1). Inefficient for large arrays.
  + **Two-Step Binary Search**: First find the rotation point (minimum element) in O(log n), then perform binary search on the appropriate sorted half. Time: O(log n), Space: O(1). More complex but equivalent in efficiency.
  + The single-pass binary search is preferred for its simplicity and directness.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty arrays (e.g., if (nums == null || nums.length == 0) return -1;) to handle edge cases robustly.
  + **Overflow Safety**: Use mid = left + (right - left) / 2 to prevent potential integer overflow, though unlikely for typical inputs.
  + **Clarity**: Add comments explaining the sorted half logic (e.g., “Check if left half is sorted to determine search range”).
  + **Edge Case Discussion**: In an interview, mention handling of no rotation, single elements, and the limitation with duplicates to show thoroughness.
  + **Optimization**: The current approach is optimal for no duplicates, but note the need for modification if duplicates are allowed.
* **Note on Sorted Half Logic**: The key is identifying the sorted half using nums[left] <= nums[mid]. If true, the left half is sorted, and we check if the target lies in [nums[left], nums[mid]]. If false, the right half is sorted, and we check if the target lies in [nums[mid], nums[right]]. This ensures the search narrows to the half containing the target, accounting for the rotation.
* Time Based Key Value Store

Commented Code

class TimeMap {

*// Map to store key-value pairs with timestamps*

private Map<String, TreeMap<Integer, String>> map;

*// Initialize the data structure with an empty HashMap*

public TimeMap() {

map = new HashMap<>();

}

*// Store key, value, and timestamp in the map*

public void set(String key, String value, int timestamp) {

*// Get or create a TreeMap for the key, then add timestamp-value pair*

map.computeIfAbsent(key, k -> new TreeMap<>()).put(timestamp, value);

}

*// Retrieve the most recent value for a key before or at the given timestamp*

public String get(String key, int timestamp) {

*// Get the TreeMap for the key*

TreeMap<Integer, String> treeMap = map.get(key);

*// If key doesn't exist, return empty string*

if (treeMap == null) {

return "";

}

*// Find the entry with the largest timestamp <= given timestamp*

Map.Entry<Integer, String> entry = treeMap.floorEntry(timestamp);

*// If no such entry exists, return empty string; otherwise, return the value*

return entry == null ? "" : entry.getValue();

}

}

**Commented Code**

class TimeMap {

*// Map to store key-value pairs with timestamps*

private Map<String, TreeMap<Integer, String>> map;

*// Initialize the data structure with an empty HashMap*

public TimeMap() {

map = new HashMap<>();

}

*// Store key, value, and timestamp in the map*

public void set(String key, String value, int timestamp) {

*// Get or create a TreeMap for the key, then add timestamp-value pair*

map.computeIfAbsent(key, k -> new TreeMap<>()).put(timestamp, value);

}

*// Retrieve the most recent value for a key before or at the given timestamp*

public String get(String key, int timestamp) {

*// Get the TreeMap for the key*

TreeMap<Integer, String> treeMap = map.get(key);

*// If key doesn't exist, return empty string*

if (treeMap == null) {

return "";

}

*// Find the entry with the largest timestamp <= given timestamp*

Map.Entry<Integer, String> entry = treeMap.floorEntry(timestamp);

*// If no such entry exists, return empty string; otherwise, return the value*

return entry == null ? "" : entry.getValue();

}

}

**Time Complexity**

* **Constructor (TimeMap)**:
  + **Time Complexity: O(1)** – Initializing an empty HashMap is a constant-time operation.
* **set Operation**:
  + **Time Complexity: O(log t)** – map.computeIfAbsent is O(1) on average for HashMap operations. Inserting into the TreeMap for the given key takes O(log t), where t is the number of timestamps for that key (due to the balanced binary search tree operations in TreeMap).
* **get Operation**:
  + **Time Complexity: O(log t)** – Retrieving the TreeMap via map.get is O(1) on average. The floorEntry operation in TreeMap takes O(log t) to find the largest timestamp less than or equal to the input timestamp, where t is the number of timestamps for the key.
* **Summary**: Both set and get operations are O(log t), where t is the number of timestamps for a specific key. In practice, t is the number of set calls for that key, often much smaller than the total number of operations.

**Space Complexity**

* **Space Complexity: O(n \* t)**
  + The HashMap stores a TreeMap for each unique key, where n is the number of unique keys (from set calls).
  + Each TreeMap stores timestamp-value pairs, with t being the number of timestamps for a given key.
  + In the worst case, if there are n unique keys, each with t timestamps, the total space is O(n \* t).
  + The values (strings) and timestamps (integers) contribute to the space, but their sizes are typically bounded by problem constraints (e.g., string length or timestamp range).
  + Thus, the space complexity is **O(n \* t)**, where n is the number of unique keys and t is the maximum number of timestamps per key.

**Concise Summary of the Approach**

The solution implements a time-based key-value store using a HashMap of TreeMaps. The HashMap maps each key to a TreeMap, which stores timestamp-value pairs in sorted order by timestamp. The set operation adds a timestamp-value pair to the TreeMap for the given key, creating a new TreeMap if the key doesn’t exist. The get operation retrieves the value with the largest timestamp less than or equal to the input timestamp using TreeMap’s floorEntry, returning an empty string if no such value exists. The approach achieves **O(log t)** time complexity for both set and get operations and **O(n \* t)** space complexity, efficiently handling timestamp-based queries.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(log t) for set and get is optimal for ordered timestamp queries, as TreeMap provides efficient logarithmic-time operations.
  + **Clarity**: The HashMap + TreeMap structure is intuitive for storing and querying time-based data, making it interview-friendly.
  + **Robustness**: Handles all edge cases, including missing keys, no valid timestamp, and multiple values per key.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a TreeMap instead of a list of timestamp-value pairs?
    - **Response**: A list would require O(t) time for get to scan for the largest timestamp ≤ input, where t is the number of timestamps for a key. TreeMap’s floorEntry provides O(log t) performance, making it more efficient for frequent queries.
  + **Interviewer might ask**: Can you optimize space for small datasets?
    - **Response**: For small numbers of timestamps per key, a list might be simpler, but it would increase get time to O(t). TreeMap balances time and space for general cases. If memory is critical and timestamps are sparse, a HashMap with direct timestamp-to-value mappings could be considered, but it doesn’t support floorEntry-like queries efficiently.
  + **Interviewer might ask**: How does this handle very large timestamps or strings?
    - **Response**: The code handles large timestamps (within integer limits) and strings as is, since TreeMap uses integers for keys and strings for values. If timestamps exceed integer limits or strings are very long, clarify constraints with the interviewer (e.g., use long for timestamps).
* **Edge Cases Handled**:
  + Missing key: Returns empty string (checked via map.get(key) == null).
  + No timestamp ≤ input: Returns empty string (checked via entry == null).
  + Single timestamp for a key: Correctly returns the value if timestamp ≤ input.
  + Multiple timestamps: TreeMap’s floorEntry finds the correct value.
  + **Note**: Add checks for null inputs or invalid timestamps (e.g., if (key == null || value == null) in set, or negative timestamps) if required.
* **Assumptions**:
  + Keys and values are non-null strings, and timestamps are positive integers.
  + The input is valid, and the problem assumes ASCII or reasonable string lengths.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **List-Based Storage**: Store timestamp-value pairs in a list for each key in the HashMap. Use binary search or linear scan for get. Time: O(1) for set, O(t) or O(log t) for get with linear or binary search, Space: O(n \* t). Less efficient for frequent get operations.
  + **Single HashMap**: Use a HashMap with composite keys (e.g., key\_timestamp). Time: O(1) for set, O(t) for get to scan timestamps, Space: O(n \* t). Inefficient for get.
  + The HashMap + TreeMap approach is preferred for its balance of O(log t) time for both operations and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null keys/values or invalid timestamps in set (e.g., if (key == null || value == null) return;) and get (e.g., if (key == null) return "";) for robustness.
  + **Clarity**: Add comments explaining the TreeMap’s role (e.g., “TreeMap stores timestamps in sorted order for efficient floor queries”).
  + **Edge Case Discussion**: In an interview, mention handling of missing keys, no valid timestamp, and multiple timestamps to show thoroughness.
  + **Optimization**: The current approach is optimal for general cases, but note that computeIfAbsent is concise and avoids explicit get/put checks.
* **Note on TreeMap’s Role**: The TreeMap ensures timestamps are stored in sorted order, allowing floorEntry to efficiently find the largest timestamp ≤ input in O(log t) time. The HashMap provides O(1) access to the TreeMap for each key, making the structure efficient for both operations.
* Median of Two Sorted Arrays

Commented Code

class Solution {

*// Method to find the median of two sorted arrays*

public double findMedianSortedArrays(int[] nums1, int[] nums2) {

int m = nums1.length, n = nums2.length; *// Lengths of the two arrays*

int total = m + n; *// Total number of elements*

int median1 = 0, median2 = 0; *// Track two potential median elements*

int i = 0, j = 0; *// Pointers for nums1 and nums2*

*// Merge arrays until reaching the middle position(s)*

for (int k = 0; k <= total / 2; k++) {

median1 = median2; *// Shift previous median value*

*// Choose smaller element from nums1 or nums2*

if (i < m && (j >= n || nums1[i] <= nums2[j])) {

median2 = nums1[i++]; *// Take from nums1, increment i*

} else {

median2 = nums2[j++]; *// Take from nums2, increment j*

}

}

*// If total length is even, average the two middle elements*

if (total % 2 == 0) return (median1 + median2) / 2.0;

*// If total length is odd, return the middle element*

else return median2;

}

}

**Time Complexity**

* **Time Complexity: O((m + n) / 2)**
  + The algorithm simulates merging the two sorted arrays until it reaches the middle position(s), i.e., (m + n) / 2 + 1 elements.
  + Each iteration of the loop performs O(1) operations: comparisons, assignments, and pointer increments.
  + The loop runs (m + n) / 2 + 1 times, which is proportional to O(m + n).
  + Therefore, the total time complexity is **O(m + n)**, where m and n are the lengths of nums1 and nums2.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses a constant number of variables (m, n, total, median1, median2, i, j, k) regardless of input size.
  + No additional data structures are used, so the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution finds the median of two sorted arrays by simulating a merge process up to the middle position(s). It uses two pointers (i for nums1, j for nums2) to select the smaller element from either array in each step, tracking the last two elements (median1, median2) to handle both odd and even total lengths. For an even total length, it averages the two middle elements; for an odd length, it returns the middle element. The approach achieves **O(m + n)** time complexity and **O(1)** space complexity, efficiently merging without storing the full merged array.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Simplicity**: The merge-based approach is intuitive, mimicking the process of merging two sorted arrays, making it easier to understand and explain.
  + **Space Efficiency**: O(1) space is optimal, as it avoids storing the merged array.
  + **Correctness**: Handles both odd and even total lengths and edge cases like empty arrays.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you achieve better time complexity?
    - **Response**: Yes, a binary search approach on the shorter array can achieve O(log(min(m, n))) time by partitioning the arrays to find the median directly, avoiding the full merge. This is more complex but faster for large arrays.
  + **Interviewer might ask**: Why track median1 and median2?
    - **Response**: For even-length arrays, the median is the average of the two middle elements. Tracking median1 (previous) and median2 (current) ensures we have both when needed. For odd-length arrays, median2 is the median.
  + **Interviewer might ask**: How does this handle duplicates or negative numbers?
    - **Response**: The code works for duplicates and negative numbers, as it only compares elements (nums1[i] <= nums2[j]) and selects the smaller one, which is valid for any integers.
* **Edge Cases Handled**:
  + One array empty: Works correctly (e.g., if nums2 is empty, j >= n is true, so it processes nums1 only).
  + Single element arrays: Correctly finds the median (e.g., one element each gives average for even case).
  + Different length arrays: Handled by merging until (m + n) / 2 + 1.
  + Odd/even total length: Handled by conditional return.
  + **Note**: Add checks for null inputs (e.g., if (nums1 == null || nums2 == null) throw new IllegalArgumentException();) for robustness.
* **Assumptions**:
  + Both arrays are non-null, sorted in ascending order, and may be empty.
  + The total length (m + n) is at least 1 (per problem constraints).
  + Elements are integers (including negatives and duplicates).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Binary Search**: Partition the shorter array to find a split where the left halves of both arrays contain the smaller elements, achieving O(log(min(m, n))) time, O(1) space. More complex but faster.
  + **Full Merge**: Merge both arrays into a new array and find the median. Time: O(m + n), Space: O(m + n). Less space-efficient.
  + The merge-based approach is simpler but less time-efficient than binary search.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null arrays (e.g., if (nums1 == null || nums2 == null) throw new IllegalArgumentException();) to handle invalid inputs.
  + **Clarity**: Add comments explaining the merge logic (e.g., “Simulate merge to find middle elements for median”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single elements, and odd/even cases to show thoroughness.
  + **Optimization**: Consider the binary search approach for O(log(min(m, n))) time if performance is critical, though the current approach is simpler for explanation.
* **Note on Merge Logic**: The algorithm simulates merging by selecting the smaller element from nums1[i] or nums2[j] until reaching the middle position(s). For even length, it uses the last two elements (median1, median2); for odd length, it uses the last element (median2). This avoids storing the merged array, keeping space at O(1).

6. Linked List (11 problems)

* Reverse Linked List

Commented Code

*/\*\**

*\* 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 {

*// Method to reverse a singly linked list*

public ListNode reverseList(ListNode head) {

ListNode prev = null; *// Initialize previous pointer as null*

ListNode curr = head; *// Initialize current pointer to head*

*// Iterate through the list*

while (curr != null) {

ListNode temp = curr.next; *// Store next node*

curr.next = prev; *// Reverse the link by pointing current to previous*

prev = curr; *// Move previous pointer forward*

curr = temp; *// Move current pointer forward*

}

*// Return new head (last non-null node)*

return prev;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the linked list exactly once, where n is the number of nodes.
  + Each iteration performs O(1) operations: storing the next node, updating pointers, and moving prev and curr.
  + Therefore, the total time complexity is **O(n)**, where n is the length of the linked list.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses only a few variables (prev, curr, temp) regardless of the list size.
  + No additional data structures are used, so the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution reverses a singly linked list iteratively using three pointers: prev, curr, and temp. It initializes prev as null and curr as the head. For each node, it saves the next node in temp, reverses the link by setting curr.next to prev, and advances prev and curr. The final prev becomes the new head. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently reversing the list in-place.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited to reverse links, and O(1) space is ideal for in-place reversal.
  + **Clarity**: The iterative approach is straightforward and easier to implement than recursive solutions, making it interview-friendly.
  + **Correctness**: Handles all cases, including empty lists and single-node lists, correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it recursively?
    - **Response**: Yes, a recursive solution processes each node by reversing the rest of the list and linking the current node to the end. It also takes O(n) time but uses O(n) space due to the call stack. The iterative approach is preferred for space efficiency.
  + **Interviewer might ask**: Why use a temp variable?
    - **Response**: The temp variable saves curr.next before reversing the link, as curr.next is overwritten. Without it, we’d lose access to the next node in the list.
  + **Interviewer might ask**: How does this handle edge cases?
    - **Response**: The code handles empty lists (head == null, returns null), single-node lists (no links to reverse, returns head), and multi-node lists correctly via the loop.
* **Edge Cases Handled**:
  + Empty list (head == null): Returns null (implicitly, as curr is null).
  + Single node: Returns the same node (no links to reverse).
  + Multiple nodes: Correctly reverses all links.
  + **Note**: The code is robust, but adding an explicit null check (e.g., if (head == null) return null;) could clarify intent, though unnecessary here.
* **Assumptions**:
  + The input is a valid singly linked list (possibly empty).
  + Nodes contain integer values, but the logic is independent of the values.
  + If these assumptions don’t hold (e.g., cyclic list), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Recursive Solution**: Recursively reverse the rest of the list, then link the current node to the end. Time: O(n), Space: O(n) due to call stack. Less space-efficient but elegant.
  + **Stack-Based**: Push all nodes onto a stack, then pop to rebuild in reverse order. Time: O(n), Space: O(n). Inefficient due to extra space.
  + The iterative approach is preferred for its O(1) space and simplicity.
* **Suggestions for Improvement**:
  + **Clarity**: Add comments explaining pointer roles (e.g., “prev tracks new head, curr processes current node”).
  + **Edge Case Discussion**: In an interview, mention handling of empty lists, single nodes, and long lists to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that prev = null initialization handles the empty list case naturally.
  + **Validation**: While not needed here, mention that checking for cycles (if allowed) would require additional logic.
* **Note on Pointer Logic**: The reversal process updates each node’s next pointer to point to the previous node (curr.next = prev), effectively flipping the list. The temp variable ensures the next node is not lost, and prev becomes the new head when curr reaches null.
* Merge Two Sorted Lists

Commented Code

*/\*\**

*\* 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 {

*// Method to merge two sorted linked lists into one sorted list*

public ListNode mergeTwoLists(ListNode list1, ListNode list2) {

ListNode dummy = new ListNode(0); *// Dummy node to simplify merging*

ListNode merge = dummy; *// Pointer to build the merged list*

*// Merge lists while both have nodes*

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

*// Choose smaller value node and append to merged list*

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

merge.next = list1; *// Append list1 node*

list1 = list1.next; *// Move list1 pointer*

} else {

merge.next = list2; *// Append list2 node*

list2 = list2.next; *// Move list2 pointer*

}

merge = merge.next; *// Move merge pointer*

}

*// Append remaining nodes from list1, if any*

if (list1 != null) {

merge.next = list1;

}

*// Append remaining nodes from list2, if any*

else {

merge.next = list2;

}

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

return dummy.next;

}

}

**Time Complexity**

* **Time Complexity: O(n + m)**
  + The algorithm processes each node in both lists exactly once, where n and m are the lengths of list1 and list2, respectively.
  + The while loop continues until one list is exhausted, taking O(min(n, m)) iterations, followed by appending the remaining nodes from the non-empty list (O(max(n, m) - min(n, m))).
  + Each iteration performs O(1) operations: comparisons, pointer updates, and assignments.
  + Therefore, the total time complexity is **O(n + m)**, where n and m are the lengths of the input lists.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses only a constant amount of extra space: the dummy node and the merge pointer.
  + The merged list is created by reassigning next pointers, not creating new nodes.
  + Therefore, the space complexity is **O(1)**, excluding the space needed for the output.

**Concise Summary of the Approach**

The solution merges two sorted linked lists into a single sorted list using a dummy node and a merge pointer. It compares the values of the current nodes from both lists, appending the smaller one to the merged list and advancing the corresponding list’s pointer. Once one list is exhausted, it appends the remaining nodes from the other list. The dummy node simplifies edge cases, and the merged list’s head is returned. The approach achieves **O(n + m)** time complexity and **O(1)** space complexity, efficiently merging the lists in-place.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n + m) time is optimal, as each node must be visited to merge, and O(1) space is ideal for in-place merging.
  + **Clarity**: The iterative approach with a dummy node is intuitive, easy to implement, and interview-friendly.
  + **Robustness**: Handles edge cases like empty lists and lists of different lengths correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a dummy node?
    - **Response**: The dummy node simplifies the merging process by providing a starting point for the merged list, avoiding special cases for the head node and ensuring clean pointer updates.
  + **Interviewer might ask**: Can you solve it recursively?
    - **Response**: Yes, a recursive solution compares the head nodes, links the smaller one to the merged result of the remaining lists, and returns. It takes O(n + m) time but O(n + m) space due to the call stack. The iterative approach is preferred for space efficiency.
  + **Interviewer might ask**: How does this handle duplicates or negative values?
    - **Response**: The code handles duplicates and negative values correctly, as it only compares node values (list1.val <= list2.val), ensuring proper sorting regardless of value types.
* **Edge Cases Handled**:
  + Both lists empty: Returns null (dummy.next is null).
  + One list empty: Returns the other list (handled by appending remaining nodes).
  + Lists of different lengths: Correctly merges until one list is exhausted, then appends the rest.
  + Single node lists: Merges correctly by comparing the single nodes.
  + **Note**: Add checks for null inputs explicitly (e.g., if (list1 == null) return list2; if (list2 == null) return list1;) for clarity, though the code handles these implicitly.
* **Assumptions**:
  + Both input lists are sorted in ascending order and may be empty.
  + Nodes contain integer values (including negatives and duplicates).
  + The lists are singly linked and acyclic.
  + If these assumptions don’t hold (e.g., unsorted lists), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Recursive Merge**: Recursively select the smaller head node and merge the remaining lists. Time: O(n + m), Space: O(n + m) due to call stack. Less space-efficient but elegant.
  + **Copy to Array**: Convert lists to arrays, merge the arrays, and build a new list. Time: O(n + m), Space: O(n + m). Inefficient due to extra space.
  + The iterative approach with a dummy node is preferred for its O(1) space and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add explicit checks for null inputs (e.g., if (list1 == null) return list2; if (list2 == null) return list1;) to clarify handling of empty lists, though the code works implicitly.
  + **Clarity**: Add comments explaining the dummy node’s role (e.g., “Dummy node simplifies head assignment”).
  + **Edge Case Discussion**: In an interview, mention handling of empty lists, different lengths, and duplicates to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that the dummy node avoids edge case complexity, making the code robust and concise.
* **Note on Dummy Node**: The dummy node provides a placeholder head for the merged list, allowing the merge pointer to build the list without special handling for the head. After merging, dummy.next points to the true head of the merged list.
* Reorder List

Commented Code

*/\*\**

*\* 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 {

*// Method to reorder the linked list as L0 → Ln → L1 → Ln-1 → L2 → ...*

public void reorderList(ListNode head) {

*// Handle null list*

if (head == null) {

return;

}

*// Step 1: Find the middle of the list using slow and fast pointers*

ListNode slow = head, fast = head;

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

slow = slow.next; *// Slow moves one step*

fast = fast.next.next; *// Fast moves two steps*

}

*// Step 2: Reverse the second half of the list*

ListNode prev = null, curr = slow, temp;

while (curr != null) {

temp = curr.next; *// Store next node*

curr.next = prev; *// Reverse the link*

prev = curr; *// Move prev forward*

curr = temp; *// Move curr forward*

}

*// Step 3: Merge the first half and reversed second half*

ListNode first = head, second = prev;

while (second.next != null) {

temp = first.next; *// Store next node of first half*

first.next = second; *// Link to node from second half*

first = temp; *// Move first pointer*

temp = second.next; *// Store next node of second half*

second.next = first; *// Link to node from first half*

second = temp; *// Move second pointer*

}

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + **Step 1: Finding the Middle**: The slow and fast pointer technique traverses the list, with the fast pointer moving twice as fast. This takes O(n/2) ≈ O(n) time, where n is the number of nodes.
  + **Step 2: Reversing the Second Half**: Reversing the second half (approximately n/2 nodes) takes O(n/2) ≈ O(n) time, as each node is processed once.
  + **Step 3: Merging the Halves**: Merging alternates nodes from the first and second halves, processing approximately n/2 nodes, taking O(n/2) ≈ O(n) time.
  + Each step performs O(1) operations per node, so the total time complexity is **O(n)**, where n is the number of nodes in the list.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses a constant number of variables (slow, fast, prev, curr, temp, first, second) regardless of list size.
  + No additional data structures are used, and all operations (reversing and merging) are done in-place.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution reorders a singly linked list such that L0 → Ln → L1 → Ln-1 → L2 → ... using three steps. First, it finds the middle of the list using slow and fast pointers. Second, it reverses the second half of the list in-place. Third, it merges the first half with the reversed second half by alternating nodes. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently reordering the list in-place.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited at least once, and O(1) space is ideal for in-place manipulation.
  + **Clarity**: The three-step process (find middle, reverse, merge) is structured and explainable, making it interview-friendly.
  + **Robustness**: Handles edge cases like null or single-node lists correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use slow and fast pointers to find the middle?
    - **Response**: The slow pointer moves one step and the fast pointer moves two steps, so when fast reaches the end, slow is at the middle. This is O(n) and uses O(1) space, ideal for splitting the list.
  + **Interviewer might ask**: Can you solve it recursively?
    - **Response**: A recursive solution could reverse and merge recursively, but it would use O(n) space due to the call stack and be more complex. The iterative approach is preferred for space efficiency.
  + **Interviewer might ask**: Why does the merge stop at second.next != null?
    - **Response**: The second half may have one fewer node than the first (e.g., in odd-length lists), so stopping when second.next is null avoids overlinking and ensures the last node is handled correctly.
* **Edge Cases Handled**:
  + Null list: Returns immediately (head == null).
  + Single node: No change needed (middle is head, second half is empty).
  + Two nodes: Reorders as L0 → L1.
  + Odd/even length lists: Handled correctly by finding middle and merging until second.next is null.
  + **Note**: The code is robust, but explicit checks for single or two-node lists could clarify intent, though unnecessary here.
* **Assumptions**:
  + The input is a valid singly linked list (possibly empty) with no cycles.
  + Node values are integers, but the logic is value-agnostic.
  + If these assumptions don’t hold (e.g., cyclic list), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **List to Array**: Convert the list to an array, reorder elements, and rebuild the list. Time: O(n), Space: O(n). Less efficient due to extra space.
  + **Recursive Reordering**: Recursively split, reverse, and merge. Time: O(n), Space: O(n) due to call stack. More complex and less space-efficient.
  + The iterative approach is preferred for its O(1) space and clarity.
* **Suggestions for Improvement**:
  + **Clarity**: Add comments for each step (e.g., “Step 1: Find middle using slow and fast pointers”).
  + **Edge Case Discussion**: In an interview, mention handling of null, single-node, and odd/even length lists to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that stopping at second.next != null ensures correct handling of odd-length lists.
  + **Validation**: The null check is sufficient, but could add explicit single-node check (e.g., if (head.next == null) return;) for clarity.
* **Note on Merging Logic**: The merge alternates nodes from the first half (first) and reversed second half (second), linking first → second → first.next → second.next. Stopping at second.next != null ensures the last node of the second half is linked correctly, avoiding issues with odd-length lists where the second half is shorter.
* Remove Nth Node From End of List

Commented Code

*/\*\**

*\* 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 {

*// Method to remove the nth node from the end of a linked list*

public ListNode removeNthFromEnd(ListNode head, int n) {

ListNode dummy = new ListNode(1); *// Dummy node to handle edge cases*

dummy.next = head; *// Link dummy to head*

ListNode front = dummy; *// Front pointer starts at dummy*

ListNode back = dummy; *// Back pointer starts at dummy*

*// Move front pointer n+1 steps ahead to create gap*

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

front = front.next;

}

*// Move both pointers until front reaches the end*

while (front != null) {

front = front.next;

back = back.next;

}

*// Remove the nth node by updating back.next*

back.next = back.next.next;

*// Return head of modified list*

return dummy.next;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The first loop moves the front pointer n + 1 steps, taking O(n) time in the worst case (if n is close to the list length).
  + The second loop moves both front and back pointers until front reaches the end, taking O(n - n) = O(n) steps, where n is the length of the list.
  + The removal operation (back.next = back.next.next) is O(1).
  + The total time complexity is **O(n)**, where n is the number of nodes in the linked list, as the list is traversed at most once.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses a constant amount of extra space: the dummy node and two pointers (front, back).
  + No additional data structures are used, so the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution removes the nth node from the end of a singly linked list using two pointers and a dummy node. A dummy node is used to handle edge cases (e.g., removing the head). The front pointer is moved n + 1 steps ahead, then both front and back pointers move together until front reaches the end, positioning back just before the node to remove. The node is removed by updating back.next. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently handling all cases in a single pass.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as the list must be traversed to locate the nth node from the end, and O(1) space is ideal.
  + **Clarity**: The two-pointer technique with a dummy node is a standard, interview-friendly solution for linked list problems.
  + **Robustness**: Handles edge cases like removing the head or last node using the dummy node.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a dummy node?
    - **Response**: The dummy node simplifies edge cases, such as removing the head node, by ensuring there’s always a node before the one to remove, avoiding special handling for the head.
  + **Interviewer might ask**: Why move front by n + 1 steps?
    - **Response**: Moving front by n + 1 steps creates a gap such that when front reaches the end, back is just before the node to remove, allowing easy link update (back.next = back.next.next).
  + **Interviewer might ask**: Can you solve it in one pass without knowing the length?
    - **Response**: This solution already achieves one pass using the two-pointer technique. An alternative requiring length calculation would need two passes, which is less efficient.
* **Edge Cases Handled**:
  + Empty list: Not explicitly checked, but n is typically valid (per problem constraints).
  + Single node (n = 1): Removes the node, returns null (dummy.next is null).
  + Remove head node: Handled by dummy node, as back points to dummy.
  + Remove last node: Handled by the gap, positioning back correctly.
  + **Note**: Add checks for null input or invalid n (e.g., if (head == null || n <= 0) return head;) for robustness.
* **Assumptions**:
  + The input list is non-null, acyclic, and n is valid (1 ≤ n ≤ length of list).
  + The list has at least n nodes.
  + If these assumptions don’t hold (e.g., invalid n), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Two-Pass Solution**: First pass to find the list length L, then traverse to the (L - n)th node to remove it. Time: O(n), Space: O(1). Less efficient due to two passes.
  + **Recursive Solution**: Recursively traverse to find and remove the node. Time: O(n), Space: O(n) due to call stack. Less space-efficient and more complex.
  + The two-pointer approach is preferred for its single-pass efficiency and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null list or invalid n (e.g., if (head == null || n <= 0) return head;). Could also verify n is not larger than the list length, though typically guaranteed by problem constraints.
  + **Clarity**: Add comments explaining the two-pointer gap (e.g., “Move front n+1 steps to position back before the node to remove”).
  + **Edge Case Discussion**: In an interview, mention handling of single-node lists, removing head/last node, and invalid n to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that the dummy node ensures robustness for edge cases like removing the head.
* **Note on Two-Pointer Technique**: The n + 1 step gap ensures that when front reaches the end (null), back is at the node just before the one to remove. This allows a simple link update (back.next = back.next.next) to skip the target node, maintaining the list’s integrity.
* Copy List With Random Pointer

Commented Code

*/\**

*// Definition for a Node.*

*class Node {*

*int val;*

*Node next;*

*Node random;*

*public Node(int val) {*

*this.val = val;*

*this.next = null;*

*this.random = null;*

*}*

*}*

*\*/*

public class Solution {

*// Method to create a deep copy of a linked list with random pointers*

public Node copyRandomList(Node head) {

*// Handle null list*

if (head == null) return null;

*// Map to store original nodes to their copies*

HashMap<Node, Node> oldToNew = new HashMap<>();

*// Step 1: Create new nodes and map them to original nodes*

Node curr = head;

while (curr != null) {

oldToNew.put(curr, new Node(curr.val)); *// Create copy with same value*

curr = curr.next;

}

*// Step 2: Set next and random pointers for copied nodes*

curr = head;

while (curr != null) {

oldToNew.get(curr).next = oldToNew.get(curr.next); *// Set next pointer*

oldToNew.get(curr).random = oldToNew.get(curr.random); *// Set random pointer*

curr = curr.next;

}

*// Return head of copied list*

return oldToNew.get(head);

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + **Step 1: Create Nodes**: The first loop iterates through the list of n nodes, creating a new node for each and adding it to the HashMap. Each operation (creating a node, putting in HashMap) is O(1) on average, so this step takes O(n).
  + **Step 2: Set Pointers**: The second loop iterates through the list again, setting next and random pointers for each copied node. Each HashMap lookup and pointer assignment is O(1) on average, so this step also takes O(n).
  + Total time: O(n) + O(n) = **O(n)**, where n is the number of nodes in the list.

**Space Complexity**

* **Space Complexity: O(n)**
  + The HashMap stores a mapping from each original node to its copy, requiring O(n) space for n nodes.
  + Additional variables (curr, oldToNew) use O(1) space.
  + The copied nodes themselves are part of the output and not counted as extra space per problem conventions.
  + Therefore, the space complexity is **O(n)** due to the HashMap.

**Concise Summary of the Approach**

The solution creates a deep copy of a linked list with random pointers using a HashMap. It first creates a copy of each node and maps original nodes to their copies. Then, it sets the next and random pointers of the copied nodes using the mappings. The approach handles null lists and returns the head of the copied list. It achieves **O(n)** time complexity and **O(n)** space complexity, efficiently managing both next and random pointers with a hash-based mapping.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Clarity**: The two-pass approach (create nodes, set pointers) is intuitive and easy to explain, making it interview-friendly.
  + **Correctness**: The HashMap ensures accurate mapping of original to copied nodes, handling complex random pointer configurations.
  + **Robustness**: Handles edge cases like null lists and arbitrary random pointers (including null or pointing to any node).
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Can you solve it with O(1) space?
    - **Response**: Yes, an O(1) space solution exists by interleaving original and copied nodes (e.g., 1→1'→2→2'), then setting random pointers, and finally separating the lists. It takes O(n) time but is more complex. The HashMap approach is simpler and clearer.
  + **Interviewer might ask**: Why use a HashMap?
    - **Response**: The HashMap maps original nodes to their copies, allowing O(1) lookups to set next and random pointers correctly, especially for arbitrary random pointers, simplifying the logic.
  + **Interviewer might ask**: How does this handle cycles in random pointers?
    - **Response**: The problem typically assumes no cycles (standard constraint), but the code works regardless, as it copies nodes and pointers without traversing random links, only using direct mappings.
* **Edge Cases Handled**:
  + Null list: Returns null (explicitly checked).
  + Single node: Copies the node, handles random pointing to itself or null.
  + random pointing to any node or null: Correctly mapped via HashMap.
  + Long lists with complex random pointers: Handled by creating all copies first, then setting pointers.
  + **Note**: Add checks for invalid inputs (e.g., ensuring n nodes exist) if required by the problem.
* **Assumptions**:
  + The input list is valid, possibly empty, with next and random pointers pointing to nodes in the list or null.
  + No cycles in the next or random pointers (per standard problem constraints).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Interleaving Nodes**: Interleave copied nodes with original nodes (e.g., 1→1'→2→2'), set random pointers using the interleaved structure, then separate the lists. Time: O(n), Space: O(1). More complex but space-efficient.
  + **Recursive Copy**: Recursively copy nodes and handle next and random pointers, using a HashMap to avoid re-copying. Time: O(n), Space: O(n) due to call stack and HashMap. Less intuitive.
  + The HashMap approach is preferred for its clarity and simplicity, despite O(n) space.
* **Suggestions for Improvement**:
  + **Clarity**: Add comments for each step (e.g., “Step 1: Create copies and map to originals”).
  + **Edge Case Discussion**: In an interview, mention handling of null lists, single nodes, and arbitrary random pointers to show thoroughness.
  + **Optimization**: The current approach is optimal for clarity and time, but mention the O(1) space interleaving method if asked for space optimization.
  + **Validation**: The null check is sufficient, but could clarify handling of single nodes or null random pointers explicitly in comments.
* **Note on HashMap Usage**: The HashMap ensures each original node maps to its copy, allowing direct access to set next and random pointers. The two-pass structure (create nodes, then set pointers) avoids issues with random pointers referencing nodes not yet copied, making the solution robust and straightforward.
* Add Two Numbers

Commented Code

*/\*\**

*\* 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 {

*// Method to add two numbers represented as linked lists*

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

ListNode dummyHead = new ListNode(0); *// Dummy node to simplify list construction*

ListNode tail = dummyHead; *// Pointer to build the result list*

int carry = 0; *// Carry for addition*

*// Process while there are digits in l1, l2, or a carry*

while (l1 != null || l2 != null || carry != 0) {

*// Get digits, use 0 if list is exhausted*

int digit1 = (l1 != null) ? l1.val : 0;

int digit2 = (l2 != null) ? l2.val : 0;

*// Calculate sum, digit, and new carry*

int sum = digit1 + digit2 + carry;

int digit = sum % 10; *// Current digit*

carry = sum / 10; *// Update carry*

*// Create new node with the digit and append to result*

ListNode newNode = new ListNode(digit);

tail.next = newNode;

tail = tail.next;

*// Move to next nodes, if available*

l1 = (l1 != null) ? l1.next : null;

l2 = (l2 != null) ? l2.next : null;

}

*// Return head of result list (skip dummy node)*

ListNode result = dummyHead.next;

dummyHead.next = null; *// Avoid memory leaks*

return result;

}

}

**Commented Code**

*/\*\**

*\* 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 {

*// Method to add two numbers represented as linked lists*

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

ListNode dummyHead = new ListNode(0); *// Dummy node to simplify list construction*

ListNode tail = dummyHead; *// Pointer to build the result list*

int carry = 0; *// Carry for addition*

*// Process while there are digits in l1, l2, or a carry*

while (l1 != null || l2 != null || carry != 0) {

*// Get digits, use 0 if list is exhausted*

int digit1 = (l1 != null) ? l1.val : 0;

int digit2 = (l2 != null) ? l2.val : 0;

*// Calculate sum, digit, and new carry*

int sum = digit1 + digit2 + carry;

int digit = sum % 10; *// Current digit*

carry = sum / 10; *// Update carry*

*// Create new node with the digit and append to result*

ListNode newNode = new ListNode(digit);

tail.next = newNode;

tail = tail.next;

*// Move to next nodes, if available*

l1 = (l1 != null) ? l1.next : null;

l2 = (l2 != null) ? l2.next : null;

}

*// Return head of result list (skip dummy node)*

ListNode result = dummyHead.next;

dummyHead.next = null; *// Avoid memory leaks*

return result;

}

}

**Time Complexity**

* **Time Complexity: O(max(n, m))**
  + The algorithm processes each node in the longer of the two lists (l1 or l2), where n and m are their lengths.
  + The while loop continues until both lists are exhausted and the carry is 0, taking at most O(max(n, m)) iterations.
  + Each iteration performs O(1) operations: accessing node values, arithmetic, creating a node, and updating pointers.
  + Therefore, the total time complexity is **O(max(n, m))**, where n and m are the lengths of l1 and l2.

**Space Complexity**

* **Space Complexity: O(max(n, m))**
  + The output list contains at most max(n, m) + 1 nodes (due to a possible carry at the end), but this is considered part of the output and not extra space.
  + The solution uses a constant amount of extra space: dummyHead, tail, carry, and temporary variables (digit1, digit2, sum, digit, newNode).
  + Therefore, the extra space complexity is **O(1)**, excluding the output list. If the output list is counted (in some interview contexts), it would be O(max(n, m)).

**Concise Summary of the Approach**

The solution adds two numbers represented as linked lists (digits in reverse order) using a dummy node and a tail pointer. It iterates through both lists, summing corresponding digits and the carry, creating a new node for each resulting digit, and updating the carry. If one list is exhausted, it uses 0 for missing digits. The process continues until both lists and the carry are exhausted. The approach achieves **O(max(n, m))** time complexity and **O(1)** extra space complexity, efficiently constructing the result list.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(max(n, m)) time is optimal, as each digit must be processed, and O(1) extra space is ideal.
  + **Clarity**: The iterative approach with a dummy node is straightforward and interview-friendly.
  + **Robustness**: Handles unequal list lengths, carries, and edge cases like empty lists or single digits.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a dummy node?
    - **Response**: The dummy node simplifies list construction by providing a starting point, avoiding special handling for the head node and ensuring clean pointer updates.
  + **Interviewer might ask**: Can you solve it recursively?
    - **Response**: Yes, a recursive solution processes each pair of nodes, handling the carry and building the list. It takes O(max(n, m)) time but O(max(n, m)) space due to the call stack. The iterative approach is preferred for space efficiency.
  + **Interviewer might ask**: How does this handle leading zeros or different-length lists?
    - **Response**: The code handles different-length lists by using 0 for exhausted lists. Leading zeros in the input don’t affect the result, as digits are processed in reverse order, and the carry ensures correct addition.
* **Edge Cases Handled**:
  + Both lists empty: Returns a list with a single 0 (due to carry = 0 initially).
  + One list empty: Processes the non-empty list with 0s for the other (e.g., l1 = [], l2 = [1]).
  + Different length lists: Handled by using 0 for missing digits.
  + Carry after last digit: Handled by continuing the loop while carry != 0.
  + **Note**: Add explicit null checks (e.g., if (l1 == null && l2 == null) return new ListNode(0);) for clarity, though the code handles these implicitly.
* **Assumptions**:
  + Both input lists are non-null, represent non-negative integers in reverse order, and are valid linked lists.
  + Node values are digits (0–9).
  + If these assumptions don’t hold (e.g., negative numbers, invalid digits), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Recursive Solution**: Recursively process each pair of nodes, handling carry and linking nodes. Time: O(max(n, m)), Space: O(max(n, m)) due to call stack. Less space-efficient.
  + **Convert to Numbers**: Convert lists to integers, add them, then convert back to a list. Time: O(max(n, m)), Space: O(max(n, m)). Fails for large numbers due to integer overflow.
  + The iterative approach is preferred for its simplicity, space efficiency, and robustness.
* **Suggestions for Improvement**:
  + **Input Validation**: Add explicit checks for null inputs (e.g., if (l1 == null && l2 == null) return new ListNode(0);) for clarity, though the code handles these cases.
  + **Clarity**: Add comments explaining the carry and dummy node (e.g., “Carry tracks overflow from sum ≥ 10”).
  + **Edge Case Discussion**: In an interview, mention handling of empty lists, unequal lengths, and final carry to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that dummyHead.next = null avoids potential memory leaks in some environments.
* **Note on Carry Handling**: The carry (sum / 10) propagates any overflow (sum ≥ 10) to the next digit, ensuring correct addition. The loop condition (l1 != null || l2 != null || carry != 0) ensures all digits and any final carry are processed, making the solution robust for all input sizes.
* Linked List Cycle

Commented Code

*/\*\**

*\* 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 {

*// Method to detect if a linked list has a cycle*

public boolean hasCycle(ListNode head) {

*// Handle null list*

if (head == null) {

return false;

}

ListNode fast = head; *// Fast pointer moves two steps*

ListNode slow = head; *// Slow pointer moves one step*

*// Traverse until fast reaches end or cycle is found*

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

fast = fast.next.next; *// Move fast by two steps*

slow = slow.next; *// Move slow by one step*

if (fast == slow) return true; *// Cycle detected if pointers meet*

}

*// No cycle if fast reaches the end*

return false;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm uses Floyd’s Cycle-Finding Algorithm (also known as the "tortoise and hare" algorithm).
  + In a list with n nodes and no cycle, the fast pointer reaches the end in O(n) steps, as it moves twice as fast as the slow pointer.
  + If there is a cycle, the fast pointer catches the slow pointer in O(n) steps in the worst case (depending on the cycle length and entry point).
  + Each iteration performs O(1) operations: pointer updates and comparisons.
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the linked list.

**Space Complexity**

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

**Concise Summary of the Approach**

The solution detects a cycle in a singly linked list using Floyd’s Cycle-Finding Algorithm. It uses two pointers: fast (moving two steps) and slow (moving one step). If there’s a cycle, the pointers meet; otherwise, fast reaches the end. The approach handles null lists and returns true for a cycle or false otherwise. It achieves **O(n)** time complexity and **O(1)** space complexity, efficiently detecting cycles without extra space.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as the list must be traversed to detect a cycle, and O(1) space is ideal.
  + **Clarity**: Floyd’s algorithm is a standard, intuitive solution for cycle detection, making it interview-friendly.
  + **Robustness**: Correctly handles edge cases like null or single-node lists.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: How does Floyd’s algorithm work?
    - **Response**: The fast pointer moves twice as fast as the slow pointer. In a cycle, they eventually meet because the relative speed ensures the fast pointer catches up (like runners on a circular track). If no cycle exists, the fast pointer reaches the end.
  + **Interviewer might ask**: Can you find the cycle’s starting point?
    - **Response**: Yes, after detecting a cycle (when fast == slow), reset one pointer to the head and move both at one step per iteration. Their next meeting point is the cycle’s start. This also takes O(n) time and O(1) space.
  + **Interviewer might ask**: Why check fast != null && fast.next != null?
    - **Response**: The fast pointer moves two steps, so we must ensure both fast and fast.next exist to avoid null pointer exceptions. If either is null, the list ends, and no cycle exists.
* **Edge Cases Handled**:
  + Null list: Returns false (explicitly checked).
  + Single node (no cycle): Returns false (fast reaches null immediately).
  + List with no cycle: Returns false (fast reaches end).
  + List with cycle: Returns true when pointers meet.
  + **Note**: The code is robust, but the null check could be paired with a check for single nodes for clarity, though unnecessary here.
* **Assumptions**:
  + The input is a singly linked list that may or may not have a cycle.
  + The list may be empty (null head).
  + Node values are irrelevant to cycle detection.
  + If these assumptions don’t hold (e.g., invalid list structure), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **HashSet-Based**: Store visited nodes in a HashSet. If a node is revisited, a cycle exists. Time: O(n), Space: O(n). Less space-efficient but simpler to understand.
  + **Modify List**: Mark visited nodes (e.g., set val to a sentinel or modify next). Time: O(n), Space: O(1). Destructive and not allowed in many cases.
  + Floyd’s algorithm is preferred for its O(1) space and elegance.
* **Suggestions for Improvement**:
  + **Clarity**: Add comments explaining Floyd’s algorithm (e.g., “Fast and slow pointers meet in a cycle due to relative speed”).
  + **Edge Case Discussion**: In an interview, mention handling of null lists, single nodes, and cycles of varying sizes to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that the fast.next != null check ensures safety for two-step moves.
  + **Validation**: The null check is sufficient, but could add explicit single-node handling for clarity (e.g., if (head.next == null) return false;).
* **Note on Floyd’s Algorithm**: The fast pointer’s two-step movement ensures it catches the slow pointer in a cycle, as their relative speed reduces the distance between them. The fast != null && fast.next != null check prevents null pointer issues, and meeting at any point confirms a cycle.
* Find The Duplicate Number

Commented Code

class Solution {

*// Method to find the duplicate number in an array*

public int findDuplicate(int[] nums) {

*// Initialize slow and fast pointers*

int slow = nums[0]; *// Start at first element*

int fast = nums[nums[0]]; *// Start at value of nums[0]*

*// Phase 1: Find intersection point of slow and fast pointers*

while (slow != fast) {

slow = nums[slow]; *// Move slow one step*

fast = nums[nums[fast]]; *// Move fast two steps*

}

*// Phase 2: Find entrance to the cycle*

slow = 0; *// Reset slow to start*

while (slow != fast) {

slow = nums[slow]; *// Move slow one step*

fast = nums[fast]; *// Move fast one step*

}

*// Return the duplicate number (entrance to cycle)*

return slow;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm uses Floyd’s Cycle-Finding Algorithm (tortoise and hare) to detect a cycle in the array, treating it as a linked list where each value points to an index.
  + **Phase 1 (Find Intersection)**: The slow pointer moves one step, and the fast pointer moves two steps per iteration. In a cycle, they meet in O(n) time in the worst case, where n is the array length, as the fast pointer catches up due to the relative speed.
  + **Phase 2 (Find Cycle Entrance)**: After resetting slow to 0, both pointers move one step per iteration and meet at the cycle’s entrance in O(n) time.
  + Each step involves O(1) operations (array access).
  + Total time: O(n) for both phases, so the overall time complexity is **O(n)**, where n is the length of the array.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses only two variables (slow, fast) regardless of array size.
  + No additional data structures are used, and the array is not modified.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution finds the duplicate number in an array (with values from 1 to n and length n+1) using Floyd’s Cycle-Finding Algorithm. It treats the array as a linked list where each value points to an index. In Phase 1, slow and fast pointers move at different speeds to find an intersection point inside the cycle (caused by the duplicate). In Phase 2, the slow pointer is reset to 0, and both move one step at a time to find the cycle’s entrance, which is the duplicate number. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently detecting the duplicate without modifying the array.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal for finding a duplicate, and O(1) space meets the problem’s constraint of not using extra space or modifying the array.
  + **Clarity**: Floyd’s algorithm is a clever application to arrays, and once explained, it’s elegant and interview-friendly.
  + **Robustness**: Works for the guaranteed single duplicate and handles all valid inputs per problem constraints.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: How does Floyd’s algorithm apply to an array?
    - **Response**: The array is treated as a linked list where each value nums[i] is a pointer to index nums[i]. Since there’s a duplicate, multiple indices point to the same index, creating a cycle. The cycle’s entrance is the duplicate number.
  + **Interviewer might ask**: Why reset slow to 0 in Phase 2?
    - **Response**: After finding the intersection in Phase 1, resetting slow to 0 and moving both pointers one step at a time ensures they meet at the cycle’s entrance, which is mathematically proven to be the duplicate (Floyd’s algorithm).
  + **Interviewer might ask**: Can you solve it with O(1) space without Floyd’s algorithm?
    - **Response**: Floyd’s algorithm is the standard O(1) space solution without modifying the array. Alternatives like sorting (modifies array) or using a hash set (O(n) space) violate constraints.
* **Edge Cases Handled**:
  + Array with single duplicate: Correctly finds the duplicate (guaranteed by problem: n+1 elements, values 1 to n).
  + Small arrays (e.g., [1,1]): Works correctly, as the cycle is immediate.
  + Large arrays with duplicate far from start: Floyd’s algorithm finds the cycle efficiently.
  + **Note**: Add checks for invalid inputs (e.g., if (nums == null || nums.length < 2) return -1;) for robustness, though typically guaranteed by problem constraints.
* **Assumptions**:
  + The array has length n+1, contains integers from 1 to n, and has exactly one duplicate.
  + The array is non-null and valid per problem constraints.
  + If these assumptions don’t hold (e.g., multiple duplicates), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **HashSet**: Store seen numbers in a HashSet; the first repeat is the duplicate. Time: O(n), Space: O(n). Violates O(1) space constraint.
  + **Sorting**: Sort the array and find the first pair of equal numbers. Time: O(n log n), Space: O(1) but modifies the array, violating constraints.
  + **Binary Search**: Use binary search to find a number where the count of numbers ≤ it exceeds the expected count. Time: O(n log n), Space: O(1). Slower than Floyd’s.
  + Floyd’s algorithm is preferred for its O(n) time, O(1) space, and adherence to constraints.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or invalid arrays (e.g., if (nums == null || nums.length < 2) return -1;) for robustness, though typically unnecessary.
  + **Clarity**: Add comments explaining Floyd’s algorithm phases (e.g., “Phase 1: Find cycle intersection; Phase 2: Find cycle entrance”).
  + **Edge Case Discussion**: In an interview, mention handling of small arrays and the guarantee of a single duplicate to show thoroughness.
  + **Optimization**: The current approach is optimal, but note that slow = nums[0] and fast = nums[nums[0]] ensure entering the cycle quickly.
* **Note on Floyd’s Algorithm**: The array is treated as a linked list where nums[i] points to index nums[i]. The duplicate creates a cycle (e.g., nums[i] = nums[j] = k means indices i and j point to index k). Phase 1 finds an intersection in the cycle, and Phase 2 finds the cycle’s entrance (the duplicate) by resetting slow and moving both pointers at the same speed.
* LRU Cache

Commented Code

class LRUCache {

int capacity; *// Maximum number of key-value pairs the cache can hold*

Map<Integer, Integer> lhmap; *// LinkedHashMap to store key-value pairs with LRU order*

*// Constructor to initialize the cache with given capacity*

public LRUCache(int capacity) {

this.capacity = capacity;

lhmap = new LinkedHashMap<>(capacity); *// Initialize LinkedHashMap*

}

*// Get value by key, update to most recently used, return -1 if key not found*

public int get(int key) {

if (lhmap.containsKey(key)) {

int val = lhmap.get(key); *// Get value*

lhmap.remove(key); *// Remove to update order*

lhmap.put(key, val); *// Re-insert as most recently used*

return val;

}

return -1; *// Key not found*

}

*// Put key-value pair, update if key exists, evict least recently used if at capacity*

public void put(int key, int value) {

if (lhmap.containsKey(key)) {

lhmap.remove(key); *// Remove existing key to update order*

} else if (lhmap.size() == capacity) {

*// Remove least recently used key (first key in LinkedHashMap)*

Integer firstKey = lhmap.keySet().iterator().next();

lhmap.remove(firstKey);

}

lhmap.put(key, value); *// Insert as most recently used*

}

}

**Time Complexity**

* **Constructor (LRUCache)**:
  + **Time Complexity: O(1)** – Initializing a LinkedHashMap with a given capacity is a constant-time operation.
* **get Operation**:
  + **Time Complexity: O(1)** – Checking containsKey, get, remove, and put in a LinkedHashMap are O(1) on average, as it uses a hash table for key access and a doubly-linked list for order maintenance.
* **put Operation**:
  + **Time Complexity: O(1)** – Similar to get, operations like containsKey, remove, keySet().iterator().next(), and put are O(1) on average in a LinkedHashMap.
* **Summary**: Both get and put operations have **O(1)** time complexity on average, leveraging the efficient hash table and linked list structure of LinkedHashMap.

**Space Complexity**

* **Space Complexity: O(capacity)**
  + The LinkedHashMap stores up to capacity key-value pairs, each requiring O(1) space.
  + The capacity variable and other overhead (e.g., references) use O(1) space.
  + Therefore, the total space complexity is **O(capacity)**, where capacity is the maximum size of the cache.

**Concise Summary of the Approach**

The solution implements an LRU (Least Recently Used) cache using a LinkedHashMap, which maintains insertion order (or access order, if configured). The get operation retrieves a value by key, updates it to the most recently used position, and returns -1 if the key is absent. The put operation inserts or updates a key-value pair, removing the least recently used entry if the cache is at capacity. The approach achieves **O(1)** time complexity for both operations and **O(capacity)** space complexity, leveraging LinkedHashMap’s efficient hash table and order maintenance.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(1) time for both get and put is optimal, as required by the problem.
  + **Clarity**: Using LinkedHashMap simplifies the implementation, as it handles LRU ordering internally, making it interview-friendly.
  + **Correctness**: Correctly maintains LRU order and handles capacity constraints.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use LinkedHashMap instead of a custom implementation?
    - **Response**: LinkedHashMap provides O(1) access, insertion, and removal with built-in order maintenance (via a doubly-linked list). A custom implementation with a hash map and doubly-linked list achieves the same complexity but requires more code and is error-prone.
  + **Interviewer might ask**: Can you implement it without LinkedHashMap?
    - **Response**: Yes, use a HashMap for key-node mappings and a custom doubly-linked list to track LRU order. get moves the node to the front, and put adds to the front, removing from the tail if at capacity. This also achieves O(1) time but is more complex to code.
  + **Interviewer might ask**: Why remove and re-insert in get and put?
    - **Response**: Removing and re-inserting updates the key to the most recently used position in the LinkedHashMap’s internal linked list, maintaining LRU order.
* **Edge Cases Handled**:
  + Empty cache: get returns -1 (no keys in lhmap).
  + Single entry: get and put work correctly, updating order as needed.
  + Full capacity: put removes the least recently used entry before adding a new one.
  + Non-existent key in get: Returns -1.
  + **Note**: Add checks for invalid capacity (e.g., if (capacity <= 0) throw new IllegalArgumentException();) in the constructor for robustness.
* **Assumptions**:
  + The capacity is positive (per problem constraints).
  + Keys and values are valid integers.
  + The cache is accessed via valid get and put operations.
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Custom HashMap + Doubly-Linked List**: Use a HashMap for O(1) key-node lookups and a doubly-linked list to maintain LRU order. Time: O(1) for get and put, Space: O(capacity). More complex but avoids LinkedHashMap dependency.
  + **Array-Based**: Use an array to store key-value pairs and track order. Time: O(capacity) for get and put due to shifting for order updates, Space: O(capacity). Inefficient for large caches.
  + The LinkedHashMap approach is preferred for its simplicity and built-in order maintenance.
* **Suggestions for Improvement**:
  + **Input Validation**: Add a check in the constructor (e.g., if (capacity <= 0) throw new IllegalArgumentException();) to ensure valid capacity.
  + **Clarity**: Add comments explaining LRU order maintenance (e.g., “Remove and re-insert to update to most recently used”).
  + **Edge Case Discussion**: In an interview, mention handling of empty cache, full capacity, and non-existent keys to show thoroughness.
  + **Optimization**: Configure LinkedHashMap for access order (e.g., new LinkedHashMap<>(capacity, 0.75f, true)) to update order on get automatically, though the current explicit remove-and-put works fine.
* **Note on LinkedHashMap**: The LinkedHashMap maintains insertion order by default, and remove-then-put ensures the accessed or updated key becomes the most recently used. Using keySet().iterator().next() efficiently accesses the least recently used key (first in the order) for eviction.
* Merge k Sorted Lists

Commented Code

*/\*\**

*\* 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 {

*// Method to merge k sorted linked lists into one sorted list*

public ListNode mergeKLists(ListNode[] lists) {

*// Min-heap to store all values from the lists*

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

*// Add all values from all lists to the min-heap*

for (ListNode list : lists) {

while (list != null) {

minHeap.add(list.val); *// Add node value to heap*

list = list.next; *// Move to next node*

}

}

*// Create a dummy node to build the merged list*

ListNode dummy = new ListNode(1);

ListNode merge = dummy;

*// Build the merged list by extracting values from the heap*

while (!minHeap.isEmpty()) {

merge.next = new ListNode(minHeap.remove()); *// Create node with smallest value*

merge = merge.next; *// Move merge pointer*

}

*// Return the head of the merged list*

return dummy.next;

}

}

**Time Complexity**

* **Time Complexity: O(N log N)**
  + Let N be the total number of nodes across all k lists.
  + **Adding to Heap**: Iterating through all lists and adding each node’s value to the PriorityQueue takes O(N log N). Each add operation is O(log N) due to heapification, and there are N nodes.
  + **Building Result List**: Removing N values from the heap takes O(N log N), as each remove operation (extract-min) is O(log N). Creating and linking each node is O(1).
  + Total time: O(N log N) for adding + O(N log N) for removing = **O(N log N)**, where N is the total number of nodes.

**Space Complexity**

* **Space Complexity: O(N)**
  + The PriorityQueue stores all N node values, requiring O(N) space.
  + The dummy node and merge pointer use O(1) space.
  + The output list (with N nodes) is not counted as extra space per problem conventions.
  + Therefore, the extra space complexity is **O(N)** due to the heap.

**Concise Summary of the Approach**

The solution merges k sorted linked lists into one sorted list using a min-heap. It first adds all node values from the lists to a PriorityQueue, then constructs the merged list by repeatedly extracting the smallest value from the heap and creating a new node. A dummy node simplifies list construction. The approach achieves **O(N log N)** time complexity and **O(N)** space complexity, where N is the total number of nodes, effectively sorting all values to produce the merged list.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Simplicity**: Using a min-heap to collect and sort all values is straightforward and leverages ’s PriorityQueue for efficiency.
  + **Correctness**: Guarantees a sorted merged list by extracting values in ascending order.
  + **Flexibility**: Works for any number of lists (k) and handles varying list lengths.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a min-heap instead of merging lists directly?
    - **Response**: A min-heap ensures the smallest value is always extracted in O(log N), but it requires O(N) space. Merging lists pairwise or using a heap of list heads can reduce space to O(k) or O(1), depending on the approach, and may be more efficient for large N.
  + **Interviewer might ask**: Can you optimize the space complexity?
    - **Response**: Yes, a more efficient approach uses a min-heap of size k to store only the current heads of the lists, achieving O(k) space and O(N log k) time, which is better for large N and small k.
  + **Interviewer might ask**: How does this handle duplicates or negative values?
    - **Response**: The code handles duplicates and negative values correctly, as the PriorityQueue sorts integers in ascending order, regardless of their values.
* **Edge Cases Handled**:
  + Empty array of lists (lists is empty): Returns null (heap is empty, so dummy.next is null).
  + Lists are empty: Returns null (no values added to heap).
  + Single list: Returns that list’s nodes in order (heap processes one list).
  + Lists of different lengths: Handled by iterating each list to its end.
  + **Note**: Add checks for null lists or invalid inputs (e.g., if (lists == null || lists.length == 0) return null;) for robustness.
* **Assumptions**:
  + The input lists is non-null, contains k sorted linked lists (possibly empty), and each node has an integer value.
  + The lists are sorted in ascending order.
  + If these assumptions don’t hold (e.g., unsorted lists), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Min-Heap of List Heads**: Use a min-heap to store the current head of each list (size k). Extract the smallest node, add the next node from that list, and repeat. Time: O(N log k), Space: O(k). More space-efficient for large N.
  + **Pairwise Merge**: Merge lists two at a time (like merge sort). Time: O(N log k) (merging k lists, each step doubling the list size), Space: O(1) excluding output. Slower but space-efficient.
  + **Divide and Conquer**: Recursively divide the list array and merge pairs. Time: O(N log k), Space: O(log k) due to recursion stack. Balances time and space.
  + The min-heap of list heads is often preferred for its O(N log k) time and O(k) space, especially for large N.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty input (e.g., if (lists == null || lists.length == 0) return null;) to handle edge cases explicitly.
  + **Clarity**: Add comments explaining the heap’s role (e.g., “Min-heap sorts all values for merged list construction”).
  + **Edge Case Discussion**: In an interview, mention handling of empty lists, single lists, and different lengths to show thoroughness.
  + **Optimization**: Use a min-heap of size k to store only the current heads of lists, reducing space to O(k) and time to O(N log k). This is more efficient for large N and small k. Example:

PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) -> a.val - b.val);

for (ListNode list : lists) {

if (list != null) minHeap.add(list);

}

ListNode dummy = new ListNode(0);

ListNode merge = dummy;

while (!minHeap.isEmpty()) {

ListNode node = minHeap.remove();

merge.next = node;

merge = merge.next;

if (node.next != null) minHeap.add(node.next);

}

return dummy.next;

This achieves O(N log k) time and O(k) space.

* **Note on Current Approach**: The approach collects all values in a min-heap, effectively sorting them, then builds the list. While correct, it uses O(N) space, which is less efficient than the O(k) space approach using a heap of list heads. The PriorityQueue ensures values are extracted in sorted order, and the dummy node simplifies list construction.
* Reverse Nodes In k Group

Commented Code

*\*\**

*\* 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 {

*// Method to reverse every k nodes in a linked list*

public ListNode reverseKGroup(ListNode head, int k) {

*// Handle edge cases: null list or k = 1 (no reversal needed)*

if (head == null || k == 1) return head;

*// Initialize dummy node to simplify head handling*

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy; *// Points to the node before the current k-group*

ListNode curr = head; *// Points to the start of the current k-group*

*// Count total number of nodes*

int count = 0;

while (curr != null) {

count++;

curr = curr.next;

}

*// Reverse k nodes at a time while enough nodes remain*

while (count >= k) {

curr = prev.next; *// Start of current k-group*

ListNode next = curr.next; *// Next node to process*

*// Reverse k nodes*

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

curr.next = next.next; *// Skip next node*

next.next = prev.next; *// Link next to front of group*

prev.next = next; *// Move next to front*

next = curr.next; *// Update next for next iteration*

}

prev = curr; *// Move prev to end of reversed group*

count -= k; *// Reduce count by k*

}

*// Return head of modified list*

return dummy.next;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + **Counting Nodes**: The first loop counts the total number of nodes, taking O(n) time, where n is the number of nodes in the list.
  + **Reversing Groups**: Each group of k nodes is reversed in O(k) time (the inner loop runs k-1 iterations, each with O(1) pointer updates). There are at most ⌊n/k⌋ groups, so reversing all groups takes O(k \* ⌊n/k⌋) ≈ O(n) time.
  + Total time: O(n) for counting + O(n) for reversing = **O(n)**, where n is the number of nodes.

**Space Complexity**

* **Space Complexity: O(1)**
  + The solution uses a constant number of variables: dummy, prev, curr, next, and count.
  + No additional data structures are used, and all operations are performed in-place.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution reverses every group of k nodes in a singly linked list using an iterative approach. It first counts the total nodes to determine how many k-groups can be reversed. A dummy node simplifies handling the head. For each k-group, it reverses the nodes by adjusting pointers, linking the reversed group back into the list. Remaining nodes (less than k) are left unchanged. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently reversing groups in-place.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited, and O(1) space is ideal for in-place reversal.
  + **Clarity**: The iterative approach with a dummy node is structured and interview-friendly, breaking the problem into counting and reversing steps.
  + **Robustness**: Handles edge cases like null lists, k=1, and partial groups correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why count the nodes first?
    - **Response**: Counting nodes ensures we only reverse complete groups of k nodes, avoiding partial reversals when fewer than k nodes remain. This prevents incorrect modifications.
  + **Interviewer might ask**: Can you solve it recursively?
    - **Response**: Yes, a recursive solution can reverse the first k nodes and recursively process the rest. It takes O(n) time but O(n/k) space due to the call stack. The iterative approach is preferred for space efficiency.
  + **Interviewer might ask**: Why use a dummy node?
    - **Response**: The dummy node simplifies handling the head of the list, especially when the first group is reversed, as it provides a consistent prev pointer for linking groups.
* **Edge Cases Handled**:
  + Null list: Returns null (checked explicitly).
  + k = 1: Returns the list unchanged (checked explicitly).
  + Single node or k > n: Returns the list unchanged (due to count < k).
  + Partial group at end: Left unreversed (handled by count >= k).
  + **Note**: Add checks for invalid k (e.g., if (k <= 0) return head;) for robustness, though typically guaranteed by problem constraints.
* **Assumptions**:
  + The input list is valid, possibly empty, and acyclic.
  + k is a positive integer, and node values are integers (value-agnostic logic).
  + If these assumptions don’t hold (e.g., negative k), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Recursive Solution**: Reverse the first k nodes recursively, then recurse on the remaining list. Time: O(n), Space: O(n/k) due to call stack. Less space-efficient.
  + **Stack-Based**: Push k nodes onto a stack, pop to reverse, and repeat. Time: O(n), Space: O(k). Less efficient due to extra space.
  + The iterative in-place approach is preferred for its O(1) space and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid k (e.g., if (k <= 0) return head;) to handle edge cases explicitly.
  + **Clarity**: Add comments explaining the reversal process (e.g., “Reverse k nodes by adjusting pointers, then link to next group”).
  + **Edge Case Discussion**: In an interview, mention handling of null lists, k=1, and partial groups to show thoroughness.
  + **Optimization**: Instead of counting nodes upfront, check for k nodes during each reversal (traverse k nodes to ensure enough exist). This avoids the initial pass but requires repeated checks, still O(n) time. Example:

ListNode prev = dummy, curr = head;

while (true) {

*// Check if k nodes exist*

ListNode temp = curr;

for (int i = 0; i < k && temp != null; i++) temp = temp.next;

if (temp == null) break; *// Fewer than k nodes remain*

*// Reverse k nodes (same as current code)*

*// Update prev and curr for next group*

}

* **Note on Reversal Logic**: For each k-group, the inner loop reverses nodes by inserting the next node at the front of the group (prev.next = next). The prev pointer moves to the end of the reversed group (curr) to link to the next segment. The dummy node ensures the head is handled seamlessly.

7. Trees (15 problems)

* Invert Binary Tree

Commented Code

*/\*\**

*\* 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 {

*// Method to invert a binary tree by swapping left and right subtrees*

public TreeNode invertTree(TreeNode root) {

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

if (root == null) {

return null;

}

*// Recursively invert right and left subtrees*

TreeNode right = invertTree(root.right);

TreeNode left = invertTree(root.left);

*// Swap the left and right children*

root.left = right;

root.right = left;

*// Return the root of the inverted tree*

return root;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm visits each node in the binary tree exactly once to perform the swap.
  + For a tree with n nodes, the recursive function processes each node with O(1) operations (checking null, swapping pointers).
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the tree.

**Space Complexity**

* **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 (e.g., a skewed tree), the height is h = n, leading to O(n) space.
  + In a balanced binary tree, the height is h = log n, leading to O(log n) space.
  + Variables left and right use O(1) space per recursive call.
  + Therefore, the space complexity is **O(h)**, where h is the height of the tree, ranging from O(log n) for balanced trees to O(n) for skewed trees.

**Concise Summary of the Approach**

The solution inverts a binary tree by recursively swapping the left and right subtrees of each node. It uses a depth-first approach: for each node, it recursively inverts the right and left subtrees, then swaps the node’s children. Null nodes are handled by returning null. The approach achieves **O(n)** time complexity to visit all nodes and **O(h)** space complexity due to the recursion stack, efficiently transforming the tree in-place.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited to swap its children.
  + **Clarity**: The recursive solution is concise and intuitive, making it ideal for interviews.
  + **Correctness**: Handles all tree structures (empty, single node, balanced, skewed) correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use recursion instead of iteration?
    - **Response**: Recursion mirrors the tree’s structure, making the code simpler and easier to understand. An iterative approach using a stack or queue is possible but more complex, with the same O(n) time and O(h) space complexity.
  + **Interviewer might ask**: How does this handle unbalanced trees?
    - **Response**: The algorithm works for any binary tree, as it processes each node exactly once. For unbalanced trees, the recursion stack may grow to O(n), but the logic remains correct.
  + **Interviewer might ask**: Can you invert the tree iteratively?
    - **Response**: Yes, use a stack or queue to process nodes iteratively, pushing the root, swapping its children, and adding children to the stack/queue. This achieves O(n) time and O(h) space but is less concise.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns null.
  + Single node: Returns the node (no children to swap).
  + Skewed tree (e.g., linked list): Correctly swaps null and non-null children.
  + Balanced tree: Swaps all subtrees correctly.
  + **Note**: The code is robust, but explicit validation (e.g., handling invalid node values) could be added if required.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers (irrelevant to the logic, as only pointers are swapped).
  + If these assumptions don’t hold (e.g., cyclic tree), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative with Stack**: Use a stack to store nodes, pop each node, swap its children, and push the children. Time: O(n), Space: O(h). More complex but avoids recursion.
  + **Iterative with Queue (Level-Order)**: Process nodes level by level using a queue, swapping children as you go. Time: O(n), Space: O(w) where w is the maximum width of the tree (O(n) in worst case). Less common for this problem.
  + The recursive approach is preferred for its simplicity and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: The code handles null inputs well, but you could add explicit checks for clarity:

if (root == null) return null;

(Already present, but emphasize in interview.)

* + **Clarity**: Add comments explaining the swap (e.g., “Swap left and right subtrees to invert the tree”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and skewed trees to show thoroughness.
  + **Iterative Option**: Be prepared to code an iterative version if asked:

public TreeNode invertTree(TreeNode root) {

if (root == null) return null;

Deque<TreeNode> stack = new ArrayDeque<>();

stack.push(root);

while (!stack.isEmpty()) {

TreeNode node = stack.pop();

TreeNode temp = node.left;

node.left = node.right;

node.right = temp;

if (node.right != null) stack.push(node.right);

if (node.left != null) stack.push(node.left);

}

return root;

}

This uses ArrayDeque for efficiency and achieves the same result.

* + **Optimization**: The recursive approach is optimal, but using ArrayDeque for iteration avoids recursion stack overhead in systems with limited stack space.
* **Note on Inversion Logic**: The algorithm swaps the left and right children of each node after recursively inverting the subtrees. This ensures the entire tree is mirrored (e.g., leftmost node becomes rightmost). The base case (root == null) stops recursion at leaf nodes’ children, and the post-order processing (recurse, then swap) ensures correct inversion.
* Maximum Depth of Binary Tree

Commented Code

*/\*\**

*\* 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 {

*// Method to find the maximum depth of a binary tree*

public int maxDepth(TreeNode root) {

*// Base case: if node is null, depth is 0*

if (root == null) {

return 0;

}

*// Recursively compute depths of left and right subtrees, return max + 1*

return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm visits each node in the binary tree exactly once to compute the depth.
  + For a tree with n nodes, each recursive call performs O(1) operations (null check, Math.max, and addition).
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the tree.

**Space Complexity**

* **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 (e.g., a skewed tree), the height is h = n, leading to O(n) space.
  + In a balanced binary tree, the height is h = log n, leading to O(log n) space.
  + No additional data structures are used, and temporary variables use O(1) space per call.
  + Therefore, the space complexity is **O(h)**, where h is the height of the tree, ranging from O(log n) for balanced trees to O(n) for skewed trees.

**Concise Summary of the Approach**

The solution computes the maximum depth of a binary tree using a recursive depth-first search (DFS). For each node, it recursively calculates the depths of the left and right subtrees and returns the maximum of the two plus 1 (for the current node). Null nodes return a depth of 0. The approach achieves **O(n)** time complexity to visit all nodes and **O(h)** space complexity due to the recursion stack, efficiently determining the tree’s maximum depth.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited to determine the maximum depth.
  + **Clarity**: The recursive solution is concise and mirrors the tree’s structure, making it ideal for interviews.
  + **Correctness**: Handles all tree configurations (empty, single node, balanced, skewed) correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use recursion instead of iteration?
    - **Response**: Recursion is natural for tree problems, as it leverages the tree’s hierarchical structure. An iterative approach using a stack or queue is possible but more complex, with the same O(n) time and O(h) space complexity.
  + **Interviewer might ask**: How does this handle unbalanced trees?
    - **Response**: The algorithm works for any binary tree, as it recursively explores both subtrees. For unbalanced trees, the recursion stack may grow to O(n), but the logic remains correct.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative solution using a stack (DFS) or queue (BFS) can compute the depth. BFS with a queue is common, tracking levels to find the maximum depth. It also achieves O(n) time and O(w) space, where w is the maximum width.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns 0.
  + Single node: Returns 1 (no children, so max(0,0) + 1).
  + Skewed tree (e.g., linked list): Returns correct depth by following the non-null path.
  + Balanced tree: Returns correct depth by taking the maximum of subtrees.
  + **Note**: The code is robust, but explicit validation (e.g., handling invalid node values) could be added if required.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers (irrelevant to the logic, as only structure matters).
  + If these assumptions don’t hold (e.g., cyclic tree), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative DFS with Stack**: Use a stack to store nodes and their depths, processing each node and updating the maximum depth. Time: O(n), Space: O(h). More complex but avoids recursion.
  + **Iterative BFS with Queue**: Use a queue to process nodes level by level, counting levels to determine depth. Time: O(n), Space: O(w) where w is the maximum width (O(n) in worst case). Common for depth-related problems.
  + The recursive approach is preferred for its simplicity and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: The code handles null inputs well, but you could emphasize this in an interview:

if (root == null) return 0;

(Already present, but highlight in discussion.)

* + **Clarity**: Add comments explaining the recursive step (e.g., “Depth is 1 plus max of left and right subtree depths”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and skewed trees to show thoroughness.
  + **Iterative Option**: Be prepared to code an iterative BFS version if asked:

public int maxDepth(TreeNode root) {

if (root == null) return 0;

Queue<TreeNode> queue = new ArrayDeque<>();

queue.offer(root);

int depth = 0;

while (!queue.isEmpty()) {

int size = queue.size();

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

TreeNode node = queue.poll();

if (node.left != null) queue.offer(node.left);

if (node.right != null) queue.offer(node.right);

}

depth++;

}

return depth;

}

This uses BFS to count levels, achieving O(n) time and O(w) space.

* + **Optimization**: The recursive approach is optimal, but using ArrayDeque for iterative solutions avoids legacy Stack class issues.
* **Note on Recursive Logic**: The algorithm uses post-order DFS, computing subtree depths before combining them with Math.max. The +1 accounts for the current node’s level. The base case (root == null) ensures empty subtrees contribute 0 to the depth, making the solution robust for all tree shapes.
* Diameter of Binary Tree

Commented Code

*/\*\**

*\* 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 {

int maxDiameter = 0; *// Class variable to track maximum diameter*

*// Method to find the diameter of a binary tree*

public int diameterOfBinaryTree(TreeNode root) {

getHeight(root); *// Compute heights and update maxDiameter*

return maxDiameter; *// Return the maximum diameter found*

}

*// Helper method to compute height and update max diameter*

private int getHeight(TreeNode node) {

*// Base case: if node is null, height is 0*

if (node == null) return 0;

*// Recursively compute heights of left and right subtrees*

int leftHeight = getHeight(node.left);

int rightHeight = getHeight(node.right);

*// Update max diameter as the sum of left and right heights*

maxDiameter = Math.max(maxDiameter, leftHeight + rightHeight);

*// Return height of current node (1 + max of subtree heights)*

return 1 + Math.max(leftHeight, rightHeight);

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The getHeight method visits each node in the binary tree exactly once to compute its height and update the diameter.
  + For a tree with n nodes, each recursive call performs O(1) operations (null check, Math.max, addition, and updating maxDiameter).
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the tree.

**Space Complexity**

* **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 (e.g., a skewed tree), the height is h = n, leading to O(n) space.
  + In a balanced binary tree, the height is h = log n, leading to O(log n) space.
  + The maxDiameter variable uses O(1) space, and no additional data structures are used.
  + Therefore, the space complexity is **O(h)**, where h is the height of the tree, ranging from O(log n) for balanced trees to O(n) for skewed trees.

**Concise Summary of the Approach**

The solution computes the diameter of a binary tree (the longest path between any two nodes) using a recursive depth-first search (DFS). The getHeight helper function calculates the height of each node’s subtrees and updates a global maxDiameter variable with the maximum path length (sum of left and right subtree heights). The diameter is the maximum number of edges between any two nodes, returned as maxDiameter. The approach achieves **O(n)** time complexity to visit all nodes and **O(h)** space complexity due to the recursion stack, efficiently computing the diameter in a single pass.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited to consider all possible paths.
  + **Clarity**: The recursive solution is concise and leverages the tree’s structure, making it interview-friendly.
  + **Correctness**: Correctly computes the diameter by considering paths through each node, handling all tree configurations.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why track maxDiameter globally?
    - **Response**: The diameter may not pass through the root, so we track the maximum path (left height + right height) at each node. A global variable avoids passing the diameter up the recursion tree, simplifying the code.
  + **Interviewer might ask**: Why is the diameter leftHeight + rightHeight?
    - **Response**: The diameter at a node is the longest path through it, which is the sum of the heights of its left and right subtrees (number of edges). We take the maximum across all nodes.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: An iterative solution using a stack or queue is possible but complex, requiring tracking heights and diameters for each node. It achieves O(n) time and O(h) space but is less intuitive than recursion.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns 0 (no path, maxDiameter remains 0).
  + Single node: Returns 0 (no edges, as leftHeight = rightHeight = 0).
  + Skewed tree (e.g., linked list): Returns correct diameter (e.g., n-1 for n nodes).
  + Balanced tree: Considers all paths, including those not passing through the root.
  + **Note**: The code is robust, but explicit validation (e.g., handling invalid node values) could be added if required.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers (irrelevant, as only structure matters).
  + The diameter is measured as the number of edges in the longest path.
  + If these assumptions don’t hold (e.g., cyclic tree), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative DFS with Stack**: Use a stack to simulate recursion, tracking heights and updating the maximum diameter. Time: O(n), Space: O(h). More complex but avoids recursion stack.
  + **BFS with Pairwise Distances**: Compute the longest path between all pairs of nodes using BFS for each node. Time: O(n²), Space: O(n). Inefficient and impractical.
  + The recursive approach is preferred for its O(n) time and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: The code handles null inputs well, but you could emphasize this in an interview:

if (root == null) return 0;

(Implicitly handled, but mention explicitly.)

* + **Clarity**: Add comments explaining the diameter calculation (e.g., “Diameter is max of left + right heights across all nodes”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and skewed trees to show thoroughness.
  + **Iterative Option**: Be prepared to code an iterative version if asked:

public int diameterOfBinaryTree(TreeNode root) {

if (root == null) return 0;

Deque<TreeNode> stack = new ArrayDeque<>();

Map<TreeNode, Integer> heights = new HashMap<>();

int maxDiameter = 0;

stack.push(root);

while (!stack.isEmpty()) {

TreeNode node = stack.peek();

if (node.left != null && !heights.containsKey(node.left)) {

stack.push(node.left);

} else if (node.right != null && !heights.containsKey(node.right)) {

stack.push(node.right);

} else {

stack.pop();

int leftHeight = heights.getOrDefault(node.left, 0);

int rightHeight = heights.getOrDefault(node.right, 0);

maxDiameter = Math.max(maxDiameter, leftHeight + rightHeight);

heights.put(node, 1 + Math.max(leftHeight, rightHeight));

}

}

return maxDiameter;

}

This uses a stack and a HashMap to track heights, achieving O(n) time and O(h) space.

* + **Optimization**: The recursive approach is optimal, but using ArrayDeque for iterative solutions avoids legacy Stack class issues.
* **Note on Diameter Calculation**: The diameter is the longest path (in edges) between any two nodes, which may or may not pass through the root. For each node, the path through it is leftHeight + rightHeight, and maxDiameter tracks the maximum across all nodes. The getHeight function returns the height (longest path from node to leaf), used both for diameter calculation and parent nodes.
* Balanced Binary Tree

Commented Code

*/\*\**

*\* 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 {

*// Method to check if a binary tree is balanced*

public boolean isBalanced(TreeNode root) {

*// Base case: empty tree is balanced*

if (root == null) {

return true;

}

*// Compute heights of left and right subtrees*

int leftHeight = getHeight(root.left);

int rightHeight = getHeight(root.right);

*// Check if heights differ by more than 1*

if (Math.abs(leftHeight - rightHeight) > 1) {

return false;

}

*// Recursively check if left and right subtrees are balanced*

return isBalanced(root.left) && isBalanced(root.right);

}

*// Helper method to compute the height of a subtree*

private int getHeight(TreeNode node) {

*// Base case: null node has height 0*

if (node == null) {

return 0;

}

*// Height is 1 plus max of left and right subtree heights*

return 1 + Math.max(getHeight(node.left), getHeight(node.right));

}

}

**Time Complexity**

* **Time Complexity: O(n log n) in balanced trees, O(n²) in worst case**
  + The isBalanced method is called for each node in the tree (O(n) calls in total, where n is the number of nodes).
  + For each node, the getHeight method traverses the entire subtree rooted at that node to compute its height, which takes O(n) time in the worst case (e.g., skewed tree).
  + In a balanced tree, the height is O(log n), and getHeight is called for each of the n nodes, leading to O(n log n) total time.
  + In a skewed tree (e.g., a linked list), getHeight takes O(n) for the root, O(n-1) for the next node, etc., summing to O(n²).
  + Therefore, the time complexity ranges from **O(n log n)** for balanced trees to **O(n²)** for skewed trees.

**Space Complexity**

* **Space Complexity: O(h)**
  + The space complexity is determined by the recursion stack of both isBalanced and getHeight, which depends on the height h of the tree.
  + In a balanced tree, h = log n, leading to O(log n) space.
  + In a skewed tree, h = n, leading to O(n) space.
  + No additional data structures are used beyond the recursion stack.
  + Therefore, the space complexity is **O(h)**, where h is the height of the tree, ranging from O(log n) for balanced trees to O(n) for skewed trees.

**Concise Summary of the Approach**

The solution checks if a binary tree is balanced (i.e., the height difference between left and right subtrees of any node is at most 1) using a recursive approach. For each node, it computes the heights of its left and right subtrees using a getHeight helper function and checks if their absolute difference exceeds 1. It then recursively verifies if both subtrees are balanced. The approach achieves **O(n log n)** time complexity for balanced trees (O(n²) worst case) and **O(h)** space complexity, correctly determining if the tree is balanced.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Clarity**: The recursive solution is straightforward, separating height computation and balance checking, making it interview-friendly.
  + **Correctness**: Accurately checks the balance condition for all nodes.
  + **Simplicity**: Easy to understand and implement, though not the most efficient.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why is this approach inefficient?
    - **Response**: The getHeight function is called repeatedly for each node’s subtrees, recomputing heights for overlapping subtrees. This leads to O(n²) time in the worst case, as each node may trigger a full subtree traversal.
  + **Interviewer might ask**: Can you optimize it to O(n)?
    - **Response**: Yes, by combining height computation and balance checking in a single DFS pass, we can avoid recomputing heights. Return -1 for unbalanced subtrees or the height if balanced, achieving O(n) time.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: An iterative solution using a stack or queue is possible but complex, requiring tracking heights and balance states. It achieves O(n) time and O(h) space but is less intuitive than the optimized recursive approach.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns true (balanced by definition).
  + Single node: Returns true (heights 0 and 0, balanced).
  + Skewed tree (e.g., linked list): Returns false if height difference > 1.
  + Balanced tree: Returns true if all nodes satisfy the balance condition.
  + **Note**: The code is robust, but an optimized version could reduce redundant computations.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers (irrelevant, as only structure matters).
  + A balanced tree has a height difference of at most 1 for each node’s subtrees.
  + If these assumptions don’t hold (e.g., cyclic tree), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Optimized Recursive (Single Pass)**: Combine height and balance checking in one DFS. Return -1 for unbalanced subtrees or the height if balanced. Time: O(n), Space: O(h). More efficient and preferred.
  + **Iterative with Stack**: Use a stack to perform post-order DFS, tracking heights and checking balance. Time: O(n), Space: O(h). More complex but avoids recursion.
  + **BFS with Level Tracking**: Compute heights level by level, checking balance. Time: O(n log n) or worse, Space: O(w) where w is the maximum width. Less efficient.
  + The optimized recursive approach is preferred for O(n) time and clarity.
* **Suggestions for Improvement**:
  + **Optimize to O(n)**: Modify the code to check balance and compute height in one pass:

public boolean isBalanced(TreeNode root) {

return checkHeight(root) != -1;

}

private int checkHeight(TreeNode node) {

if (node == null) return 0;

int leftHeight = checkHeight(node.left);

if (leftHeight == -1) return -1; *// Left subtree unbalanced*

int rightHeight = checkHeight(node.right);

if (rightHeight == -1) return -1; *// Right subtree unbalanced*

if (Math.abs(leftHeight - rightHeight) > 1) return -1; *// Current node unbalanced*

return 1 + Math.max(leftHeight, rightHeight); *// Return height*

}

This achieves O(n) time by avoiding redundant height calculations.

* + **Clarity**: Add comments explaining the balance condition (e.g., “Tree is balanced if height difference ≤ 1 and subtrees are balanced”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and skewed trees to show thoroughness.
  + **Input Validation**: The code handles null inputs well, but emphasize this in discussion:

if (root == null) return true;

* + **Iterative Option**: Be prepared to discuss an iterative solution, though the optimized recursive version is usually sufficient.
* **Note on Balance Logic**: A tree is balanced if, for every node, the height difference between its left and right subtrees is at most 1, and both subtrees are balanced. The current approach computes heights separately, leading to redundant traversals. The optimized version (suggested above) combines these checks, returning -1 for unbalanced subtrees to short-circuit recursion.
* Same Tree

Commented Code

*/\*\**

*\* 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 {

*// Method to check if two binary trees are identical*

public boolean isSameTree(TreeNode p, TreeNode q) {

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

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

return true;

}

*// Base case: one node is null, trees are different*

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

return false;

}

*// Check if current nodes have different values*

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

return false;

}

*// Recursively check left and right subtrees*

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

}

}

**Commented Code**

*/\*\**

*\* 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 {

*// Method to check if two binary trees are identical*

public boolean isSameTree(TreeNode p, TreeNode q) {

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

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

return true;

}

*// Base case: one node is null, trees are different*

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

return false;

}

*// Check if current nodes have different values*

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

return false;

}

*// Recursively check left and right subtrees*

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

}

}

**Time Complexity**

* **Time Complexity: O(min(n, m))**
  + The algorithm recursively compares nodes of both trees, stopping early if a mismatch is found (e.g., one node is null or values differ).
  + In the worst case, it visits all nodes of the smaller tree (with min(n, m) nodes, where n and m are the number of nodes in trees p and q).
  + Each recursive call performs O(1) operations (null checks, value comparison, logical AND).
  + Therefore, the time complexity is **O(min(n, m))**, where n and m are the sizes of the trees. If the trees have similar sizes (e.g., both have n nodes), this is O(n).

**Space Complexity**

* **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 (min(h\_p, h\_q), where h\_p and h\_q are the heights of trees p and q).
  + In the worst case (e.g., both trees are skewed), the height is equal to the number of nodes, leading to O(min(n, m)) space.
  + In balanced trees, the height is O(log n) or O(log m), leading to O(min(log n, log m)) space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(min(h\_p, h\_q))**, ranging from O(min(log n, log m)) for balanced trees to O(min(n, m)) for skewed trees.

**Concise Summary of the Approach**

The solution checks if two binary trees are identical by recursively comparing their structures and node values. It returns true if both nodes are null, false if one is null or values differ, and recursively checks the left and right subtrees otherwise. The approach achieves **O(min(n, m))** time complexity to compare nodes of the smaller tree and **O(min(h\_p, h\_q))** space complexity due to the recursion stack, efficiently determining if the trees are the same.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(min(n, m)) time is optimal, as all nodes of the smaller tree must be compared to confirm identity.
  + **Clarity**: The recursive solution is concise and mirrors the tree’s structure, making it ideal for interviews.
  + **Correctness**: Handles all cases (identical trees, different structures, different values) correctly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why check both null cases separately?
    - **Response**: Checking p == null && q == null handles the case where both subtrees are empty (identical), while p == null || q == null catches cases where one subtree is empty and the other isn’t (not identical). This ensures structural differences are detected early.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative solution using a stack or queue can compare nodes level by level or in-order, achieving O(min(n, m)) time and O(min(h\_p, h\_q)) or O(w) space (where w is the maximum width). Recursion is simpler and equally efficient.
  + **Interviewer might ask**: What if the trees have different shapes but same values?
    - **Response**: The algorithm checks both structure (null vs. non-null) and values, so different shapes (e.g., one has a left child, the other doesn’t) return false.
* **Edge Cases Handled**:
  + Both trees empty (p = null, q = null): Returns true.
  + One tree empty (p = null or q = null): Returns false.
  + Single node trees: Returns true if values match, false otherwise.
  + Different structures or values: Returns false (e.g., one tree has extra nodes or different values).
  + **Note**: The code is robust, but explicit validation (e.g., handling invalid node values) could be added if required.
* **Assumptions**:
  + The input trees are valid binary trees, possibly empty.
  + Node values are integers, and identity requires matching values and structure.
  + If these assumptions don’t hold (e.g., cyclic trees), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative DFS with Stack**: Use a stack to store pairs of nodes from both trees, comparing them in-order or pre-order. Time: O(min(n, m)), Space: O(min(h\_p, h\_q)). More complex but avoids recursion.
  + **Iterative BFS with Queue**: Use a queue to compare nodes level by level, checking values and structure. Time: O(min(n, m)), Space: O(w) where w is the maximum width of the smaller tree. Suitable for level-order comparison.
  + The recursive approach is preferred for its simplicity and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: The code handles null cases well, but you could combine null checks for conciseness:

if (p == null || q == null) return p == q;

This handles both null cases in one line.

* + **Clarity**: Add comments explaining the recursive checks (e.g., “Compare node values and recursively check subtrees”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and structural differences to show thoroughness.
  + **Iterative Option**: Be prepared to code an iterative BFS version if asked:

public boolean isSameTree(TreeNode p, TreeNode q) {

if (p == null || q == null) return p == q;

Deque<TreeNode[]> queue = new ArrayDeque<>();

queue.offer(new TreeNode[]{p, q});

while (!queue.isEmpty()) {

TreeNode[] pair = queue.poll();

TreeNode nodeP = pair[0], nodeQ = pair[1];

if (nodeP == null && nodeQ == null) continue;

if (nodeP == null || nodeQ == null || nodeP.val != nodeQ.val) return false;

queue.offer(new TreeNode[]{nodeP.left, nodeQ.left});

queue.offer(new TreeNode[]{nodeP.right, nodeQ.right});

}

return true;

}

This uses BFS with ArrayDeque for efficiency, achieving O(min(n, m)) time and O(w) space.

* + **Optimization**: The recursive approach is optimal, but using ArrayDeque for iterative solutions avoids legacy Stack class issues.
* **Note on Comparison Logic**: The algorithm checks structural and value equality simultaneously. It stops early on mismatches (null vs. non-null or different values), ensuring efficiency. The recursive AND (&&) ensures both left and right subtrees are identical for the trees to be considered the same.
* Subtree of Another Tree

Commented Code

*/\*\**

*\* 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 {

*// Method to check if subRoot is a subtree of root*

public boolean isSubtree(TreeNode root, TreeNode subRoot) {

*// Base case: empty subRoot is a subtree*

if (subRoot == null) {

return true;

}

*// Base case: empty root cannot contain non-empty subRoot*

if (root == null) {

return false;

}

*// Check if current trees are identical or if subRoot is in left/right subtree*

if (sameTree(root, subRoot)) {

return true;

}

return isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);

}

*// Helper method to check if two trees are identical*

public boolean sameTree(TreeNode root, TreeNode subRoot) {

*// Both null: identical*

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

return true;

}

*// Both non-null and values match: check subtrees*

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

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

}

*// One null or values differ: not identical*

return false;

}

}

**Time Complexity**

* **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, calling sameTree at each node in the worst case (e.g., no match until the last node).
  + The sameTree function compares two trees, taking O(min(m, n)) time, as it stops early if trees differ in structure or values.
  + In the worst case, sameTree is called for all n nodes of root, and each call takes O(min(m, n)) time (e.g., when subRoot is almost identical to each subtree but differs at the last node).
  + Therefore, the total time complexity is **O(n \* min(m, n))**. If m is much smaller than n, this approximates O(n \* m).

**Space Complexity**

* **Space Complexity: O(max(h\_r, h\_s))**
  + The space complexity is determined by the recursion stack for isSubtree and sameTree.
  + isSubtree recurses to the height of root, which is h\_r (height of root). In the worst case (skewed tree), h\_r = n; in a balanced tree, h\_r = log n.
  + sameTree recurses to the height of the smaller tree, min(h\_r, h\_s), where h\_s is the height of subRoot. In the worst case, h\_s = m (skewed) or h\_s = log m (balanced).
  + The deepest recursion occurs when isSubtree reaches a leaf and calls sameTree, using O(max(h\_r, h\_s)) space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(max(h\_r, h\_s))**, ranging from O(max(log n, log m)) for balanced trees to O(max(n, m)) for skewed trees.

**Concise Summary of the Approach**

The solution checks if subRoot is a subtree of root using a recursive approach. The isSubtree function checks if the trees rooted at root and subRoot are identical (via sameTree) or if subRoot is a subtree of root’s left or right subtree. The sameTree helper compares two trees for identical structure and values. The approach achieves **O(n \* min(m, n))** time complexity and **O(max(h\_r, h\_s))** space complexity, efficiently determining if subRoot is a subtree of root.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Clarity**: The recursive solution is intuitive, separating subtree checking (isSubtree) and tree comparison (sameTree), making it interview-friendly.
  + **Correctness**: Handles all cases, including null trees, identical trees, and partial matches.
  + **Modularity**: The sameTree helper is reusable and clearly defined.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why check subRoot == null as true?
    - **Response**: An empty tree is considered a subtree of any tree by convention (similar to an empty set being a subset). This handles edge cases cleanly.
  + **Interviewer might ask**: Can you optimize the time complexity?
    - **Response**: The current approach is straightforward but inefficient due to repeated sameTree calls. Optimizations like string serialization (e.g., pre-order traversal with unique separators) or KMP for subtree matching can reduce complexity to O(n + m) in some cases, but they’re more complex and require careful handling of duplicates.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: An iterative solution using stacks for DFS or queues for BFS is possible but complex, requiring iterative versions of both isSubtree and sameTree. It achieves similar time and space complexity but is less intuitive.
* **Edge Cases Handled**:
  + Empty subRoot (subRoot = null): Returns true (empty tree is a subtree).
  + Empty root (root = null, subRoot != null): Returns false (non-empty subtree cannot exist).
  + Single node trees: Checks value and structure correctly.
  + Different structures or values: Returns false via sameTree.
  + **Note**: The code is robust, but explicit validation (e.g., handling invalid node values) could be added if required.
* **Assumptions**:
  + Both input trees are valid binary trees, possibly empty.
  + Node values are integers, and subtree identity requires matching values and structure.
  + If these assumptions don’t hold (e.g., cyclic trees), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **String Serialization**: Convert both trees to pre-order or in-order traversal strings with unique separators for nulls. Check if subRoot’s string is a substring of root’s string using KMP. Time: O(n + m) for serialization and matching, Space: O(n + m). Complex and sensitive to duplicate values.
  + **Iterative DFS/BFS**: Use stacks or queues to iterate through root’s nodes, checking each subtree against subRoot iteratively. Time: O(n \* min(m, n)), Space: O(max(h\_r, h\_s)). More complex than recursion.
  + **Merkle Tree Hashing**: Compute hashes for subtrees and compare. Time: O(n + m), Space: O(n + m). Advanced and not typically interview-friendly.
  + The recursive approach is preferred for its simplicity and clarity, despite higher time complexity.
* **Suggestions for Improvement**:
  + **Combine Null Checks**: Simplify the null checks in isSubtree:

if (subRoot == null) return true;

if (root == null) return false;

(Already clear, but mention in interview for clarity.)

* + **Optimize Time Complexity**: Use serialization for O(n + m) time, but note its complexity:

public boolean isSubtree(TreeNode root, TreeNode subRoot) {

if (subRoot == null) return true;

if (root == null) return false;

String rootStr = serialize(root);

String subStr = serialize(subRoot);

return rootStr.contains(subStr);

}

private String serialize(TreeNode node) {

if (node == null) return "#";

return node.val + "," + serialize(node.left) + "," + serialize(node.right);

}

This requires careful handling of duplicates (e.g., unique separators).

* + **Clarity**: Add comments explaining the subtree logic (e.g., “Check if subRoot matches current tree or any subtree”).
  + **Edge Case Discussion**: In an interview, mention handling of null cases, single nodes, and large trees to show thoroughness.
  + **Iterative Option**: Be prepared to discuss an iterative DFS/BFS approach, though the recursive version is typically sufficient.
* **Note on Subtree Logic**: The isSubtree function checks if subRoot matches the tree rooted at root or any of its subtrees. The sameTree helper ensures exact matches in structure and values. The OR (||) in isSubtree allows checking all possible subtrees, and early termination in sameTree improves efficiency.
* Lowest Common Ancestor of a Binary Search Tree

Commented Code

*/\*\**

*\* 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 {

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

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

TreeNode cur = root; *// Start at the root*

*// Traverse the BST based on p and q values*

while (cur != null) {

*// If both p and q are greater than current node, go right*

if (p.val > cur.val && q.val > cur.val) {

cur = cur.right;

}

*// If both p and q are less than current node, go left*

else if (p.val < cur.val && q.val < cur.val) {

cur = cur.left;

}

*// If p and q are on different sides or one equals cur, cur is the LCA*

else {

return cur;

}

}

return null; *// No LCA found (should not occur given problem constraints)*

}

}

**Time Complexity**

* **Time Complexity: O(h)**
  + The algorithm traverses a path from the root to the lowest common ancestor (LCA) in a binary search tree (BST).
  + The number of iterations depends on the height h of the tree, as it moves left or right based on the values of p and q.
  + In a balanced BST, h = log n, where n is the number of nodes, leading to O(log n) time.
  + In a skewed BST (e.g., a linked list), h = n, leading to O(n) time.
  + Each iteration performs O(1) operations (comparisons and pointer updates).
  + Therefore, the time complexity is **O(h)**, ranging from O(log n) for balanced trees to O(n) for skewed trees.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses an iterative approach with only a single pointer (cur) and no additional data structures.
  + No recursion stack or auxiliary space is needed.
  + Therefore, the space complexity is **O(1)**, regardless of tree size or shape.

**Concise Summary of the Approach**

The solution finds the lowest common ancestor (LCA) of two nodes p and q in a binary search tree using an iterative approach. It traverses the BST starting from the root, moving right if both p and q have values greater than the current node, left if both are smaller, or returns the current node if p and q are on different sides or one matches the current node. The approach achieves **O(h)** time complexity, where h is the tree height, and **O(1)** space complexity, efficiently leveraging the BST property to find the LCA.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(h) time is optimal for a BST, leveraging the BST property to avoid searching the entire tree. O(1) space is ideal for iterative solutions.
  + **Clarity**: The iterative approach is concise and easy to follow, making it interview-friendly.
  + **Correctness**: Correctly identifies the LCA by exploiting the BST property (all nodes in left subtree < current node < all nodes in right subtree).
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why does this work for a BST but not a general binary tree?
    - **Response**: The BST property ensures that if p and q are both greater or both less than the current node, they lie in the same subtree, allowing directional traversal. In a general binary tree, we’d need to search both subtrees, as no ordering exists.
  + **Interviewer might ask**: Can you solve it recursively?
    - **Response**: Yes, a recursive solution follows the same logic but uses O(h) space for the recursion stack. The iterative approach is more space-efficient.
  + **Interviewer might ask**: What if p or q is not in the tree?
    - **Response**: The problem guarantees p and q exist in the tree. If not guaranteed, we’d need to verify their presence (e.g., search for both nodes first).
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns null (handled by loop termination).
  + p or q equals the root: Returns root (since p.val == cur.val or q.val == cur.val triggers the else case).
  + p and q in same subtree: Correctly traverses to the LCA.
  + p is ancestor of q (or vice versa): Returns p (or q) when one value matches cur.val.
  + **Note**: Add input validation (e.g., if (root == null || p == null || q == null) return null;) for robustness.
* **Assumptions**:
  + The input tree is a valid BST with unique node values.
  + Nodes p and q exist in the tree (per problem constraints).
  + Node values are integers, and the LCA is the lowest node whose subtree contains both p and q.
  + If these assumptions don’t hold (e.g., duplicate values, nodes not in tree), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Recursive Solution**: Recursively traverse the BST with the same logic (go left, right, or return current node). Time: O(h), Space: O(h) due to recursion stack. Less space-efficient but equally clear.
  + **General Binary Tree Approach**: For non-BSTs, search both subtrees to find paths to p and q, then find the last common node. Time: O(n), Space: O(h). Overkill for BSTs due to lack of ordering.
  + The iterative BST approach is preferred for its O(1) space and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**:
    - Add checks for null inputs or invalid cases:

if (root == null || p == null || q == null) return null;

* + - Optionally verify p and q exist in the tree (though guaranteed by constraints).
  + **Clarity**: Add comments explaining the BST property (e.g., “Use BST ordering to traverse toward LCA”).
  + **Edge Case Discussion**: In an interview, mention handling of root as LCA, same-subtree cases, and ancestor cases to show thoroughness.
  + **Recursive Option**: Be prepared to code a recursive version if asked:

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

if (root == null) return null;

if (p.val > root.val && q.val > root.val) {

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

} else if (p.val < root.val && q.val < root.val) {

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

} else {

return root;

}

}

This achieves O(h) time but uses O(h) space.

* + **Optimization**: The iterative approach is optimal for space (O(1)). For clarity, you could handle edge cases like p.val == q.val explicitly, though it’s handled correctly by the current logic.
* **Note on LCA Logic**: The algorithm leverages the BST property: if both p and q are greater than the current node, they lie in the right subtree; if both are less, they lie in the left subtree; otherwise, the current node is the LCA (either p or q equals the node, or they’re in different subtrees). The iterative approach avoids recursion overhead, making it space-efficient.
* Binary Tree Level Order Traversal

Commented Code

*/\*\**

*\* 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 {

*// Method to perform level-order traversal of a binary tree*

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

List<List<Integer>> ans = new ArrayList<>(); *// List to store level-wise node values*

Queue<TreeNode> q = new LinkedList<>(); *// Queue for BFS*

*// Handle empty tree*

if (root == null) return ans;

*// Start with root node*

q.offer(root);

*// Process each level*

while (!q.isEmpty()) {

List<Integer> row = new ArrayList<>(); *// List for current level's values*

int n = q.size(); *// Number of nodes in current level*

*// Process all nodes in current level*

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

TreeNode front = q.poll(); *// Dequeue node*

row.add(front.val); *// Add node value to current level*

*// Enqueue left child if exists*

if (front.left != null) q.offer(front.left);

*// Enqueue right child if exists*

if (front.right != null) q.offer(front.right);

}

ans.add(row); *// Add current level's values to result*

}

return ans; *// Return level-order traversal*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm performs a breadth-first search (BFS), visiting each node in the tree exactly once.
  + For a tree with n nodes, each node is enqueued and dequeued once (O(1) per operation), and its value is added to the result list (O(1)).
  + The loop processes each level, and the total number of iterations across all levels equals the number of nodes n.
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the tree.

**Space Complexity**

* **Space Complexity: O(w)**
  + The space complexity is determined by the queue and the output list.
  + The queue stores nodes at the current level, which is at most the maximum width w of the tree. In a balanced binary tree, w = n/2 in the worst case (bottom level), leading to O(n). In a skewed tree, w = 1, leading to O(1).
  + The row list stores values for the current level, also up to O(w) space.
  + The output ans list stores all node values, but this is part of the required output and not counted as extra space per problem conventions.
  + Therefore, the auxiliary space complexity is **O(w)**, where w is the maximum width of the tree, ranging from O(1) for skewed trees to O(n) for balanced trees.

**Concise Summary of the Approach**

The solution performs a level-order (breadth-first) traversal of a binary tree using a queue. It processes nodes level by level, adding each level’s values to a list and enqueueing children for the next level. The result is a list of lists, where each inner list contains the node values at a given level. The approach achieves **O(n)** time complexity to visit all nodes and **O(w)** space complexity, where w is the maximum tree width, efficiently producing the level-order traversal.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited to produce the level-order traversal.
  + **Clarity**: The BFS approach using a queue is intuitive and standard for level-order traversal, making it interview-friendly.
  + **Correctness**: Correctly handles all tree configurations, including empty, single-node, balanced, and skewed trees.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a queue for level-order traversal?
    - **Response**: A queue ensures nodes are processed in the order they appear level by level (FIFO), which is necessary for level-order traversal. A stack (DFS) would not guarantee level-wise processing.
  + **Interviewer might ask**: Can you solve it without a queue (e.g., recursively)?
    - **Response**: Yes, a recursive solution can track levels using a depth parameter, but it’s less intuitive and requires O(n) space for the recursion stack. BFS with a queue is the standard approach.
  + **Interviewer might ask**: How does this handle very wide trees?
    - **Response**: The queue’s size is bounded by the maximum width w, which could be O(n) in a balanced tree. This is unavoidable for level-order traversal, as we must store all nodes at a level.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns empty list.
  + Single node: Returns [[val]] (one level with one value).
  + Skewed tree (e.g., linked list): Returns a list of single-element lists.
  + Balanced tree: Returns lists corresponding to each level’s nodes.
  + **Note**: The code is robust, but explicit validation (e.g., handling invalid node values) could be added if required.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers (irrelevant to the logic, as values are simply collected).
  + If these assumptions don’t hold (e.g., cyclic trees), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Recursive DFS**: Use recursion with a depth parameter to build level lists, adding nodes to the appropriate level. Time: O(n), Space: O(h) for recursion stack plus O(n) for output. Less intuitive and harder to manage levels.
  + **Iterative with Two Queues**: Use two queues to separate current and next levels. Time: O(n), Space: O(w). More complex than a single queue.
  + The BFS queue-based approach is preferred for its clarity and efficiency.
* **Suggestions for Improvement**:
  + **Input Validation**: The code handles null inputs well, but you could emphasize this in an interview:

if (root == null) return ans;

(Already present, but highlight in discussion.)

* + **Use ArrayDeque**: Replace LinkedList with ArrayDeque for better performance:

Queue<TreeNode> q = new ArrayDeque<>();

ArrayDeque is more efficient for queue operations than LinkedList.

* + **Clarity**: Add comments explaining the level-processing logic (e.g., “Process all nodes in current level, enqueue children for next level”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and wide trees to show thoroughness.
  + **Recursive Option**: Be prepared to code a recursive version if asked:

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

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

if (root == null) return ans;

dfs(root, 0, ans);

return ans;

}

private void dfs(TreeNode node, int level, List<List<Integer>> ans) {

if (node == null) return;

if (level == ans.size()) ans.add(new ArrayList<>());

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

dfs(node.left, level + 1, ans);

dfs(node.right, level + 1, ans);

}

This uses DFS with level tracking, achieving O(n) time and O(h) space (excluding output).

* + **Optimization**: The BFS approach is optimal, but ArrayDeque improves constant factors.
* **Note on Level-Order Logic**: The algorithm uses BFS to process nodes level by level. The inner loop processes all nodes in the current level (tracked by q.size()), adding their values to a row list and enqueueing their children for the next level. This ensures the output lists are ordered by level, top to bottom.
* Binary Tree Right Side View

Commented Code

*/\*\**

*\* 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 {

*// Method to return the rightmost node values at each level of a binary tree*

public List<Integer> rightSideView(TreeNode root) {

List<Integer> res = new ArrayList<>(); *// List to store rightmost node values*

Queue<TreeNode> q = new LinkedList<>(); *// Queue for BFS*

q.offer(root); *// Start with root*

*// Process each level using BFS*

while (!q.isEmpty()) {

TreeNode rightSide = null; *// Track rightmost non-null node in current level*

int qLen = q.size(); *// Number of nodes in current level*

*// Process all nodes in current level*

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

TreeNode node = q.poll(); *// Dequeue node*

if (node != null) {

rightSide = node; *// Update rightmost node*

q.offer(node.left); *// Enqueue left child*

q.offer(node.right); *// Enqueue right child*

}

}

*// Add rightmost node's value to result if it exists*

if (rightSide != null) {

res.add(rightSide.val);

}

}

return res; *// Return right side view*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm performs a breadth-first search (BFS), visiting each node in the tree exactly once.
  + For a tree with n nodes, each node is enqueued and dequeued once (O(1) per operation), and its value is processed (O(1)).
  + The loop processes each level, and the total number of iterations across all levels equals n.
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the tree.

**Space Complexity**

* **Space Complexity: O(w)**
  + The space complexity is determined by the queue and the output list.
  + The queue stores nodes at the current level, which is at most the maximum width w of the tree. In a balanced binary tree, w = n/2 in the worst case (bottom level), leading to O(n). In a skewed tree, w = 1, leading to O(1).
  + The res list stores one value per level, which is O(h) (where h is the height), but this is part of the output and not counted as extra space per problem conventions.
  + The rightSide variable and loop counters use O(1) space.
  + Therefore, the auxiliary space complexity is **O(w)**, where w is the maximum width, ranging from O(1) for skewed trees to O(n) for balanced trees.

**Concise Summary of the Approach**

The solution returns the rightmost node value at each level of a binary tree using a breadth-first search (BFS). It processes nodes level by level with a queue, tracking the last non-null node in each level as the rightmost visible node. These values are added to the result list. The approach achieves **O(n)** time complexity to visit all nodes and **O(w)** space complexity, where w is the maximum tree width, efficiently producing the right side view.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited to determine the rightmost node per level.
  + **Clarity**: The BFS approach is intuitive for level-order problems, and tracking the last node per level is straightforward, making it interview-friendly.
  + **Correctness**: Correctly captures the rightmost node at each level, handling null nodes and all tree configurations.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why track the last non-null node per level?
    - **Response**: The rightmost node at each level is the last non-null node processed in BFS, as BFS visits nodes left to right. Tracking rightSide ensures we capture the value visible from the right side.
  + **Interviewer might ask**: Can you solve it with DFS?
    - **Response**: Yes, a DFS approach (pre-order, right-to-left) can track the first node at each depth, updating the result list. It achieves O(n) time and O(h) space but requires depth tracking, making BFS simpler for this problem.
  + **Interviewer might ask**: How does this handle sparse trees with many null nodes?
    - **Response**: The code checks for null nodes before processing children, ensuring only valid nodes are added to the result. The rightSide != null check prevents adding values from empty levels.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns empty list (no nodes enqueued).
  + Single node: Returns [val] (one level with one value).
  + Skewed tree (e.g., right-only): Returns values of all nodes in order.
  + Balanced tree: Returns rightmost node value per level.
  + **Note**: Add explicit validation for null root for clarity, though handled correctly.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers, and the right side view includes the rightmost non-null node per level.
  + If these assumptions don’t hold (e.g., cyclic trees), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **DFS with Right-to-Left Traversal**: Use recursive DFS, prioritizing right children and tracking depth to update the first node seen at each level. Time: O(n), Space: O(h). Simpler space complexity but less intuitive for level-order problems.
  + **Recursive Level Tracking**: Use recursion to build level lists, taking the last node per level. Time: O(n), Space: O(h) for recursion plus O(n) for output. More complex than BFS.
  + The BFS approach is preferred for its clarity and natural level-order processing.
* **Suggestions for Improvement**:
  + **Input Validation**: Explicitly handle null root (already done, but emphasize):

if (root == null) return res;

* + **Use ArrayDeque**: Replace LinkedList with ArrayDeque for better performance:

Queue<TreeNode> q = new ArrayDeque<>();

ArrayDeque is more efficient for queue operations.

* + **Clarity**: Add comments explaining the rightmost node logic (e.g., “Track last non-null node per level as right side view”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and sparse trees to show thoroughness.
  + **DFS Option**: Be prepared to code a DFS version if asked:

public List<Integer> rightSideView(TreeNode root) {

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

dfs(root, 0, res);

return res;

}

private void dfs(TreeNode node, int depth, List<Integer> res) {

if (node == null) return;

if (depth == res.size()) res.add(node.val); *// First node at this depth*

dfs(node.right, depth + 1, res); *// Right first for rightmost view*

dfs(node.left, depth + 1, res);

}

This uses DFS, prioritizing right children, achieving O(n) time and O(h) space.

* + **Optimization**: The BFS approach is optimal, but you could optimize by adding only the last node’s value per level directly (current code already does this efficiently).
* **Note on Right Side View Logic**: The algorithm uses BFS to process nodes level by level, updating rightSide to the last non-null node in each level. This ensures the rightmost node’s value is captured, as BFS processes nodes left to right. The null check (rightSide != null) prevents adding values from empty levels.
* Count Good Nodes In Binary Tree

Commented Code

*/\*\**

*\* 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 {

*// Method to count good nodes in a binary tree*

public int goodNodes(TreeNode root) {

*// Start with smallest possible max value*

return countGoodNodes(root, Integer.MIN\_VALUE);

}

*// Helper method to count good nodes recursively*

private int countGoodNodes(TreeNode node, int maxSoFar) {

*// Base case: null node contributes 0 good nodes*

if (node == null) {

return 0;

}

int count = 0;

*// Node is good if its value is >= max value on path from root*

if (node.val >= maxSoFar) {

count = 1; *// Count current node as good*

maxSoFar = node.val; *// Update max value for descendants*

}

*// Recursively count good nodes in left and right subtrees*

count += countGoodNodes(node.left, maxSoFar);

count += countGoodNodes(node.right, maxSoFar);

return count; *// Return total good nodes in this subtree*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm performs a depth-first search (DFS), visiting each node in the tree exactly once.
  + For a tree with n nodes, each node involves O(1) operations (null check, comparison, addition, and updating maxSoFar).
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the tree.

**Space Complexity**

* **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 binary tree, h = log n, leading to O(log n) space.
  + In a skewed tree (e.g., a linked list), h = n, leading to O(n) space.
  + No additional data structures are used beyond the recursion stack and a few variables (count, maxSoFar) that use O(1) space per call.
  + Therefore, the space complexity is **O(h)**, ranging from O(log n) for balanced trees to O(n) for skewed trees.

**Concise Summary of the Approach**

The solution counts "good" nodes in a binary tree, where a node is good if its value is greater than or equal to the maximum value on the path from the root to that node. It uses a recursive DFS, tracking the maximum value seen (maxSoFar) along the path. For each node, it checks if its value is at least maxSoFar, incrementing a counter if true and updating maxSoFar. The counts from left and right subtrees are added to the total. The approach achieves **O(n)** time complexity to visit all nodes and **O(h)** space complexity due to the recursion stack, efficiently counting good nodes.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be visited to check if it’s good.
  + **Clarity**: The recursive DFS is concise and intuitive, tracking maxSoFar naturally, making it interview-friendly.
  + **Correctness**: Correctly identifies good nodes by maintaining the maximum value along each path.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use Integer.MIN\_VALUE as the initial maxSoFar?
    - **Response**: Integer.MIN\_VALUE ensures the root node is always considered good, as its value will be ≥ Integer.MIN\_VALUE. It avoids edge cases where the tree has negative values.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative DFS or BFS can track the maximum value along paths using a stack or queue with node-value pairs. It achieves O(n) time and O(h) or O(w) space but is more complex than recursion.
  + **Interviewer might ask**: What if the tree has duplicate values?
    - **Response**: The condition node.val >= maxSoFar handles duplicates correctly, counting nodes with equal values as good if they meet the path condition.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns 0 (no good nodes).
  + Single node: Returns 1 (node is good, as it’s ≥ Integer.MIN\_VALUE).
  + Skewed tree: Correctly counts good nodes along the single path.
  + Negative values: Handled by initializing maxSoFar to Integer.MIN\_VALUE.
  + **Note**: The code is robust, but explicit validation (e.g., null checks) is already handled.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers, possibly negative, and a node is good if its value is ≥ the maximum on its path from the root.
  + If these assumptions don’t hold (e.g., cyclic trees), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative DFS with Stack**: Use a stack to store pairs of nodes and the maximum value seen on their path. Process nodes in-order or pre-order, updating counts. Time: O(n), Space: O(h). More complex but avoids recursion.
  + **BFS with Level Tracking**: Use a queue to track nodes and their path maxima, processing level by level. Time: O(n), Space: O(w) where w is the maximum width. Less intuitive for path-based problems.
  + The recursive DFS approach is preferred for its simplicity and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: The code handles null nodes well, but you could emphasize this in an interview:

if (root == null) return 0; *// Implicitly handled by countGoodNodes*

* + **Clarity**: Add comments explaining the good node condition (e.g., “Node is good if its value is ≥ max value on path from root”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, negative values, and duplicates to show thoroughness.
  + **Iterative Option**: Be prepared to code an iterative DFS version if asked:

public int goodNodes(TreeNode root) {

if (root == null) return 0;

Deque<Pair<TreeNode, Integer>> stack = new ArrayDeque<>();

stack.push(new Pair<>(root, Integer.MIN\_VALUE));

int count = 0;

while (!stack.isEmpty()) {

Pair<TreeNode, Integer> pair = stack.pop();

TreeNode node = pair.getKey();

int maxSoFar = pair.getValue();

if (node.val >= maxSoFar) {

count++;

maxSoFar = node.val;

}

if (node.right != null) stack.push(new Pair<>(node.right, maxSoFar));

if (node.left != null) stack.push(new Pair<>(node.left, maxSoFar));

}

return count;

}

This uses a stack with Pair (assuming a simple Pair class) to track nodes and path maxima, achieving O(n) time and O(h) space.

* + **Optimization**: The recursive approach is optimal, but using ArrayDeque for iterative solutions improves performance over legacy Stack.
* **Note on Good Nodes Logic**: A node is good if its value is at least the maximum value on the path from the root to that node. The recursive function tracks maxSoFar, updating it when a good node is found (since descendants must compare against the new maximum). The count accumulates good nodes from all subtrees.
* Validate Binary Search Tree

Commented Code

*/\*\**

*\* 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 {

*// Method to check if a binary tree is a valid BST*

public boolean isValidBST(TreeNode root) {

*// Start with the widest possible range for the root*

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

}

*// Helper method to validate BST with range constraints*

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

*// Base case: null node is valid*

if (node == null) {

return true;

}

*// Check if current node's value is within the valid range*

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

return false;

}

*// Recursively validate left and right subtrees with updated ranges*

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

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

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm performs a depth-first search (DFS), visiting each node in the tree exactly once to check if its value is within the valid range.
  + For a tree with n nodes, each recursive call performs O(1) operations (null check, range comparison, logical AND).
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the tree.

**Space Complexity**

* **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 binary tree, h = log n, leading to O(log n) space.
  + In a skewed tree (e.g., a linked list), h = n, leading to O(n) space.
  + No additional data structures are used beyond the recursion stack and parameters (left, right), which use O(1) space per call.
  + Therefore, the space complexity is **O(h)**, ranging from O(log n) for balanced trees to O(n) for skewed trees.

**Concise Summary of the Approach**

The solution checks if a binary tree is a valid binary search tree (BST) by recursively ensuring each node’s value lies within a valid range. The valid helper function takes a node and a range (left, right) and checks if the node’s value is strictly between these bounds. It then recursively validates the left subtree (values < node’s value) and right subtree (values > node’s value). The initial range uses Long.MIN\_VALUE and Long.MAX\_VALUE to accommodate all possible integer values. The approach achieves **O(n)** time complexity to visit all nodes and **O(h)** space complexity due to the recursion stack, efficiently validating the BST property.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be checked to validate the BST property.
  + **Clarity**: The recursive approach with range constraints is intuitive and clearly enforces BST rules, making it interview-friendly.
  + **Correctness**: Using long for ranges avoids integer overflow issues and handles edge cases with duplicate or extreme values.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use long instead of int for range bounds?
    - **Response**: Using long prevents issues with edge cases like Integer.MIN\_VALUE or Integer.MAX\_VALUE as node values. For example, if a node’s value is Integer.MIN\_VALUE, the left subtree needs a bound less than it, which int cannot represent.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative in-order traversal with a stack can check if values are strictly increasing, achieving O(n) time and O(h) space. The recursive range-based approach is simpler and equally efficient.
  + **Interviewer might ask**: What if the tree has duplicate values?
    - **Response**: The code assumes a BST has no duplicates (per standard BST definition). The condition left < node.val && node.val < right ensures strict ordering. If duplicates are allowed, clarify the BST definition with the interviewer.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns true (empty tree is a valid BST).
  + Single node: Returns true (single node is always valid).
  + Skewed tree: Validates correctly by checking ranges along the path.
  + Extreme values (e.g., Integer.MIN\_VALUE, Integer.MAX\_VALUE): Handled by using long for bounds.
  + **Note**: The code is robust, but explicit validation (e.g., cyclic tree checks) could be added if required.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers, and a valid BST has no duplicate values (all left subtree values < node value < all right subtree values).
  + If these assumptions don’t hold (e.g., duplicates allowed), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **In-Order Traversal**: Perform an in-order traversal (iterative or recursive) and check if values are strictly increasing. Time: O(n), Space: O(h). Simple but requires tracking the previous value.
  + **Iterative Range-Based**: Use a stack to simulate the recursive range-checking approach, storing nodes with their valid ranges. Time: O(n), Space: O(h). More complex but avoids recursion.
  + The recursive range-based approach is preferred for its clarity and robustness.
* **Suggestions for Improvement**:
  + **Input Validation**: The code handles null nodes well, but you could emphasize this in an interview:

if (root == null) return true; *// Implicitly handled by valid*

* + **Clarity**: Add comments explaining the range logic (e.g., “Ensure node value is strictly between left and right bounds”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and extreme values to show thoroughness.
  + **Iterative Option**: Be prepared to code an iterative in-order version if asked:

public boolean isValidBST(TreeNode root) {

Deque<TreeNode> stack = new ArrayDeque<>();

TreeNode curr = root;

Long prev = Long.MIN\_VALUE;

while (curr != null || !stack.isEmpty()) {

while (curr != null) {

stack.push(curr);

curr = curr.left;

}

curr = stack.pop();

if (curr.val <= prev) return false;

prev = (long) curr.val;

curr = curr.right;

}

return true;

}

This uses in-order traversal to check strictly increasing values, achieving O(n) time and O(h) space.

* + **Optimization**: The recursive approach is optimal, but using ArrayDeque for iterative solutions improves performance over legacy Stack.
* **Note on BST Validation Logic**: A valid BST requires all nodes in the left subtree to have values less than the node’s value and all nodes in the right subtree to have values greater. The range-based approach enforces this by updating bounds (left, right) at each node, ensuring strict ordering (left < node.val < right). Using long avoids overflow issues with int bounds.
* Kth Smallest Element In a Bst

Commented Code

*/\*\**

*\* 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 method for in-order traversal to collect node values*

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

*// Base case: if node is null, return the array*

if (root == null) {

return arr;

}

*// Recursively traverse left subtree*

inOrder(root.left, arr);

*// Add current node's value*

arr.add(root.val);

*// Recursively traverse right subtree*

inOrder(root.right, arr);

return arr;

}

*// Method to find the kth smallest element in a BST*

public int kthSmallest(TreeNode root, int k) {

*// Get in-order traversal of BST*

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

*// Return the (k-1)th element (0-based index)*

return nums.get(k - 1);

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The inOrder method performs an in-order traversal of the BST, visiting each node exactly once to add its value to the list. This takes O(n) time, where n is the number of nodes.
  + The kthSmallest method calls inOrder and accesses the (k-1)th element in the resulting list, which is O(1).
  + Therefore, the total time complexity is **O(n)**, dominated by the in-order traversal.

**Space Complexity**

* **Space Complexity: O(n + h)**
  + The inOrder method uses O(n) space for the ArrayList to store all node values.
  + The recursion stack for the in-order traversal depends on the height h of the tree. In a balanced BST, h = log n, leading to O(log n) space; in a skewed tree, h = n, leading to O(n) space.
  + The output list nums is not counted as extra space per problem conventions, but the arr parameter in inOrder is.
  + Therefore, the space complexity is **O(n + h)**, where n is for the list and h is the tree height, ranging from O(n + log n) for balanced trees to O(n) for skewed trees. Practically, this is often simplified to **O(n)** due to the list dominating.

**Concise Summary of the Approach**

The solution finds the kth smallest element in a binary search tree (BST) by performing an in-order traversal to collect all node values in sorted order into an ArrayList. It then returns the (k-1)th element (0-based index) from the list. The approach leverages the BST property that in-order traversal yields values in ascending order. It achieves **O(n)** time complexity to traverse the tree and **O(n + h)** space complexity due to the list and recursion stack, providing a straightforward way to find the kth smallest element.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Clarity**: The in-order traversal is intuitive for BSTs, as it naturally produces sorted values, making the solution easy to understand and implement.
  + **Correctness**: Correctly finds the kth smallest element by leveraging the BST’s in-order property.
  + **Simplicity**: The two-method structure separates traversal and kth element retrieval, making it interview-friendly.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use in-order traversal?
    - **Response**: In a BST, in-order traversal (left, root, right) visits nodes in ascending order of their values, so the kth node visited is the kth smallest element.
  + **Interviewer might ask**: Can you optimize space complexity?
    - **Response**: Yes, an iterative in-order traversal can track the kth node without storing all values, reducing space to O(h). Stop after finding the kth node to avoid traversing the entire tree in some cases.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative in-order traversal using a stack can count nodes and return the kth value, achieving O(n) time and O(h) space, improving space efficiency.
* **Edge Cases Handled**:
  + Empty tree (root = null): Returns empty list from inOrder, but k is guaranteed valid, so no issue.
  + Single node: Returns the node’s value if k = 1.
  + Skewed tree: Correctly processes nodes in-order (e.g., ascending values for right-skewed).
  + Large k: Problem guarantees 1 ≤ k ≤ n, so nums.get(k-1) is safe.
  + **Note**: Add input validation (e.g., if (root == null || k <= 0)) for robustness.
* **Assumptions**:
  + The input tree is a valid BST with unique node values.
  + 1 ≤ k ≤ n, where n is the number of nodes (per problem constraints).
  + Node values are integers.
  + If these assumptions don’t hold (e.g., invalid k, duplicates), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative In-Order with Early Stopping**: Use a stack for in-order traversal, counting nodes until the kth is found. Time: O(h + k) (stops after k nodes), Space: O(h). More space-efficient.
  + **Morris Traversal**: Perform in-order traversal without recursion or stack, using temporary links. Time: O(n), Space: O(1) excluding output. Complex and rarely needed.
  + **Recursive with Counter**: Modify the recursive approach to count nodes in-order and stop at the kth. Time: O(h + k), Space: O(h). Similar to iterative but recursive.
  + The iterative in-order approach is often preferred for better space efficiency.
* **Suggestions for Improvement**:
  + **Optimize Space**: Modify to avoid storing all values:

public int kthSmallest(TreeNode root, int k) {

Deque<TreeNode> stack = new ArrayDeque<>();

TreeNode curr = root;

int count = 0;

while (curr != null || !stack.isEmpty()) {

while (curr != null) {

stack.push(curr);

curr = curr.left;

}

curr = stack.pop();

count++;

if (count == k) return curr.val;

curr = curr.right;

}

return -1; *// Should not reach here given valid k*

}

This achieves O(n) time (O(h + k) if stopping early) and O(h) space.

* + **Input Validation**: Add checks for null root or invalid k:

if (root == null || k <= 0) throw new IllegalArgumentException("Invalid input");

(Though guaranteed by constraints.)

* + **Clarity**: Add comments explaining in-order traversal (e.g., “In-order traversal yields sorted BST values”).
  + **Edge Case Discussion**: In an interview, mention handling of single nodes, skewed trees, and valid k to show thoroughness.
  + **Use ArrayDeque**: For iterative solutions, use ArrayDeque instead of Stack for better performance.
* **Note on In-Order Logic**: In a BST, in-order traversal visits nodes in ascending order. The kth node visited has the kth smallest value. The current approach collects all values, which is simple but space-intensive. The iterative version stops after k nodes, improving efficiency for small k.
* Construct Binary Tree From Preorder And Inorder Traversal

Commented Code

*/\*\**

*\* 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 {

int pre\_idx = 0; *// Track current index in preorder array*

HashMap<Integer, Integer> indices = new HashMap<>(); *// Map inorder values to indices*

*// Method to build binary tree from preorder and inorder traversals*

public TreeNode buildTree(int[] preorder, int[] inorder) {

*// Store inorder value-to-index mappings for O(1) lookup*

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

indices.put(inorder[i], i);

}

*// Start recursive construction with full inorder range*

return dfs(preorder, 0, inorder.length - 1);

}

*// Helper method for recursive tree construction*

private TreeNode dfs(int[] preorder, int l, int r) {

*// Base case: if left index exceeds right, return null*

if (l > r) return null;

*// Get root value from preorder and increment index*

int root\_val = preorder[pre\_idx++];

TreeNode root = new TreeNode(root\_val);

*// Find root's index in inorder to split left and right subtrees*

int mid = indices.get(root\_val);

*// Recursively build left and right subtrees*

root.left = dfs(preorder, l, mid - 1);

root.right = dfs(preorder, mid + 1, r);

return root; *// Return constructed subtree*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + Building the indices HashMap takes O(n) time, where n is the length of the inorder array (number of nodes).
  + The dfs method visits each node exactly once to construct the tree. Each recursive call performs O(1) operations: accessing pre\_idx, creating a TreeNode, HashMap lookup, and recursive calls.
  + The total number of recursive calls is n, as each node is processed once.
  + Therefore, the total time complexity is **O(n)** for the HashMap construction plus O(n) for the tree construction, resulting in **O(n)**.

**Space Complexity**

* **Space Complexity: O(n + h)**
  + The indices HashMap stores n key-value pairs (inorder values to indices), using O(n) space.
  + The recursion stack for dfs depends on the height h of the tree. In a balanced tree, h = log n, leading to O(log n) space; in a skewed tree, h = n, leading to O(n) space.
  + The pre\_idx variable uses O(1) space.
  + The output tree is not counted as extra space per problem conventions.
  + Therefore, the space complexity is **O(n + h)**, where n is for the HashMap and h is the recursion stack, ranging from O(n + log n) for balanced trees to O(n) for skewed trees. Practically, this is simplified to **O(n)** due to the HashMap dominating.

**Concise Summary of the Approach**

The solution constructs a binary tree from its preorder and inorder traversals using a recursive approach. It uses a HashMap to store inorder value-to-index mappings for O(1) lookup. The dfs method builds the tree by selecting the next root from the preorder array, finding its position in the inorder array to determine left and right subtree boundaries, and recursively constructing subtrees. The approach achieves **O(n)** time complexity to process all nodes and **O(n + h)** space complexity for the HashMap and recursion stack, efficiently reconstructing the binary tree.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be processed to build the tree, and the HashMap reduces lookup time from O(n) to O(1).
  + **Clarity**: The recursive approach mirrors the tree’s structure, and the HashMap simplifies splitting subtrees, making it interview-friendly.
  + **Correctness**: Correctly reconstructs the tree using preorder (root-first) and inorder (left-root-right) properties, handling all valid inputs.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a HashMap?
    - **Response**: The HashMap allows O(1) lookup of a node’s index in the inorder array, avoiding linear search (O(n) per node). Without it, the time complexity would be O(n²).
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative solution using a stack can mimic the recursive process, tracking subtree boundaries. It achieves O(n) time and O(h) space but is more complex and less intuitive.
  + **Interviewer might ask**: What if there are duplicate values?
    - **Response**: The problem guarantees unique values in the arrays. If duplicates were allowed, the inorder traversal would be ambiguous, requiring clarification on the BST definition.
* **Edge Cases Handled**:
  + Empty tree (preorder and inorder empty): Returns null (handled by l > r).
  + Single node: Correctly builds a single-node tree.
  + Skewed tree: Correctly processes one-sided trees using inorder boundaries.
  + Balanced tree: Efficiently handles full subtrees.
  + **Note**: Add input validation (e.g., check array lengths) for robustness.
* **Assumptions**:
  + The input arrays preorder and inorder are valid, have equal length n, and represent a valid binary tree with unique values.
  + 1 ≤ n ≤ 3000 (per typical problem constraints).
  + If these assumptions don’t hold (e.g., mismatched arrays, duplicates), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative with Stack**: Use a stack to track nodes and boundaries, processing preorder and inorder iteratively. Time: O(n), Space: O(h). More complex but reduces recursion stack usage.
  + **Without HashMap**: Search for the root value in the inorder array for each node. Time: O(n²) due to linear search, Space: O(h). Inefficient for large trees.
  + The HashMap-based recursive approach is preferred for its O(n) time and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid inputs:

if (preorder == null || inorder == null || preorder.length != inorder.length) {

return null;

}

* + **Clarity**: Add comments explaining preorder/inorder roles (e.g., “Preorder gives root, inorder splits subtrees”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and skewed trees to show thoroughness.
  + **Iterative Option**: Be prepared to code an iterative version if asked:

public TreeNode buildTree(int[] preorder, int[] inorder) {

if (preorder.length == 0) return null;

Deque<TreeNode> stack = new ArrayDeque<>();

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

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

indices.put(inorder[i], i);

}

TreeNode root = new TreeNode(preorder[0]);

stack.push(root);

int pre\_idx = 1;

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

TreeNode node = new TreeNode(preorder[i]);

TreeNode parent = stack.peek();

int mid = indices.get(preorder[i]);

int parentMid = indices.get(parent.val);

if (mid < parentMid) {

parent.left = node;

} else {

while (!stack.isEmpty() && indices.get(stack.peek().val) < mid) {

parent = stack.pop();

}

parent.right = node;

}

stack.push(node);

}

return root;

}

This achieves O(n) time and O(h) space but is more complex.

* + **Optimization**: The recursive approach with HashMap is optimal for time (O(n)). Consider ArrayDeque for iterative solutions to improve performance.
* **Note on Construction Logic**: Preorder traversal gives the root node first, while inorder traversal splits the tree into left and right subtrees around the root’s index. The HashMap enables O(1) lookup of the root’s inorder index (mid), allowing efficient subtree boundary calculation (l to mid-1 for left, mid+1 to r for right). The pre\_idx tracks the next root in preorder.
* Binary Tree Maximum Path Sum

Commented Code

*/\*\**

*\* 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 {

int ans = Integer.MIN\_VALUE; *// Global variable to track maximum path sum*

*// Method to find the maximum path sum in a binary tree*

public int maxPathSum(TreeNode root) {

helper(root); *// Compute max path sum*

return ans; *// Return global maximum*

}

*// Helper method to compute max gain and update global max*

private int helper(TreeNode node) {

*// Base case: null node contributes 0*

if (node == null) return 0;

*// Get max path sum from left and right subtrees, ignore negative paths*

int left = Math.max(helper(node.left), 0);

int right = Math.max(helper(node.right), 0);

*// Compute path sum through current node (node + left + right)*

int pathSum = node.val + left + right;

*// Update global maximum path sum*

ans = Math.max(ans, pathSum);

*// Return max gain for parent: node value + max of left or right path*

return node.val + Math.max(left, right);

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The helper method performs a depth-first search (DFS), visiting each node exactly once.
  + For a tree with n nodes, each node involves O(1) operations: null check, Math.max calls, addition, and updating ans.
  + Therefore, the total time complexity is **O(n)**, where n is the number of nodes in the tree.

**Space Complexity**

* **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 binary tree, h = log n, leading to O(log n) space.
  + In a skewed tree (e.g., a linked list), h = n, leading to O(n) space.
  + The ans variable uses O(1) space, and no additional data structures are used.
  + Therefore, the space complexity is **O(h)**, ranging from O(log n) for balanced trees to O(n) for skewed trees.

**Concise Summary of the Approach**

The solution finds the maximum path sum in a binary tree, where a path’s sum is the total value of nodes along it. It uses a recursive DFS via the helper method, which computes the maximum gain (node value plus the maximum path from one subtree) for each node and updates a global ans with the maximum path sum through the node (node value plus left and right paths). Negative paths are ignored by taking Math.max(0, path). The approach achieves **O(n)** time complexity to visit all nodes and **O(h)** space complexity due to the recursion stack, efficiently finding the maximum path sum.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each node must be considered to find the maximum path sum.
  + **Clarity**: The recursive approach is concise and naturally handles the path sum logic, making it interview-friendly.
  + **Correctness**: Correctly computes the maximum path sum, including paths that may not pass through the root, and handles negative values.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why ignore negative paths with Math.max(0, path)?
    - **Response**: Negative path sums reduce the total sum, so excluding them (by choosing 0) maximizes the path sum. This allows flexibility to include only the current node or positive subtree paths.
  + **Interviewer might ask**: Why use a global variable for ans?
    - **Response**: The maximum path sum may occur in a subtree, not necessarily including the root. A global variable tracks the maximum across all nodes without needing to return it through the recursion stack.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: An iterative solution using a stack is possible but complex, requiring tracking of path sums and maximum gains for each node. It achieves O(n) time and O(h) space but is less intuitive than recursion.
* **Edge Cases Handled**:
  + Empty tree (root = null): Not applicable (problem guarantees non-empty tree), but helper returns 0 for null nodes.
  + Single node: Returns the node’s value (e.g., node.val with left = right = 0).
  + Negative values: Handled by ignoring negative paths with Math.max(0, path).
  + Skewed tree: Correctly computes path sums along the single path.
  + **Note**: Add input validation (e.g., if (root == null)) for robustness, though not strictly needed.
* **Assumptions**:
  + The input tree is a valid binary tree with at least one node.
  + Node values are integers, possibly negative.
  + The maximum path sum is the sum of node values along any path (may not include root or leaves).
  + If these assumptions don’t hold (e.g., empty tree), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative with Stack**: Use a stack for post-order DFS, tracking maximum path sums and gains for each node. Time: O(n), Space: O(h). More complex and rarely preferred.
  + **Brute Force**: Check all possible paths between pairs of nodes. Time: O(n²), Space: O(h). Inefficient and impractical.
  + The recursive DFS approach is preferred for its O(n) time and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add check for null root:

if (root == null) return Integer.MIN\_VALUE; *// Or throw exception*

* + **Clarity**: Add comments explaining path sum logic (e.g., “Path sum through node includes left and right subtrees; return max gain to parent”).
  + **Edge Case Discussion**: In an interview, mention handling of single nodes, negative values, and skewed trees to show thoroughness.
  + **Iterative Option**: Be prepared to discuss an iterative approach, though the recursive version is typically sufficient:

public int maxPathSum(TreeNode root) {

Deque<TreeNode> stack = new ArrayDeque<>();

Map<TreeNode, Integer> gains = new HashMap<>();

int ans = Integer.MIN\_VALUE;

TreeNode curr = root, last = null;

while (curr != null || !stack.isEmpty()) {

while (curr != null) {

stack.push(curr);

curr = curr.left;

}

curr = stack.peek();

if (curr.right == null || curr.right == last) {

stack.pop();

int left = gains.getOrDefault(curr.left, 0);

int right = gains.getOrDefault(curr.right, 0);

left = Math.max(left, 0);

right = Math.max(right, 0);

ans = Math.max(ans, curr.val + left + right);

gains.put(curr, curr.val + Math.max(left, right));

last = curr;

curr = null;

} else {

curr = curr.right;

}

}

return ans;

}

This uses a stack and HashMap for O(n) time and O(h) space, but is more complex.

* + **Optimization**: The recursive approach is optimal for time (O(n)). Consider ArrayDeque for iterative solutions.
* **Note on Path Sum Logic**: The maximum path sum can be any path in the tree (straight or through a node). For each node, helper computes the maximum gain (node value plus the best subtree path) to return to the parent, while updating ans with the maximum path through the node (node value plus left and right paths). Ignoring negative paths (Math.max(0, path)) ensures optimal sums.
* Serialize And Deserialize Binary Tree

Commented Code

*/\*\**

*\* 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 Codec {

*// Encodes a tree to a single string.*

public String serialize(TreeNode root) {

if (root == null) return "null"; *// Handle empty tree*

StringBuilder sb = new StringBuilder(); *// Build serialized string*

Queue<TreeNode> q = new LinkedList<>(); *// Queue for level-order traversal*

q.add(root);

*// Perform BFS to serialize tree*

while (!q.isEmpty()) {

TreeNode curr = q.poll();

if (curr == null) {

sb.append("null,"); *// Append null for missing nodes*

continue;

}

sb.append(curr.val).append(","); *// Append node value*

q.add(curr.left); *// Enqueue left child*

q.add(curr.right); *// Enqueue right child*

}

return sb.toString(); *// Return serialized string*

}

*// Decodes your encoded data to tree.*

public TreeNode deserialize(String data) {

if (data.equals("null")) return null; *// Handle empty tree*

String[] arr = data.split(","); *// Split string into array*

TreeNode root = new TreeNode(Integer.parseInt(arr[0])); *// Create root node*

Queue<TreeNode> q = new LinkedList<>(); *// Queue for BFS reconstruction*

q.add(root);

int i = 1; *// Index for array traversal*

*// Reconstruct tree using BFS*

while (!q.isEmpty() && i < arr.length) {

TreeNode curr = q.poll(); *// Dequeue parent node*

*// Process left child*

if (!arr[i].equals("null")) {

curr.left = new TreeNode(Integer.parseInt(arr[i]));

q.add(curr.left);

}

i++;

*// Process right child*

if (i < arr.length && !arr[i].equals("null")) {

curr.right = new TreeNode(Integer.parseInt(arr[i]));

q.add(curr.right);

}

i++;

}

return root; *// Return reconstructed tree*

}

}

**Time Complexity**

* **Serialize: O(n)**
  + The serialize method performs a breadth-first search (BFS), visiting each node exactly once.
  + For a tree with n nodes, each node is enqueued/dequeued (O(1)), and its value (or "null") is appended to the StringBuilder (O(1)).
  + The total time is **O(n)**, where n is the number of nodes.
* **Deserialize: O(n)**
  + The deserialize method splits the input string (O(n), where n is the number of nodes, as the string length is proportional to n) and processes each node via BFS.
  + Each node is enqueued/dequeued (O(1)), and its value is parsed to create a TreeNode (O(1)).
  + The total time is **O(n)** for splitting and processing nodes.
* **Overall**: Both methods are **O(n)**.

**Space Complexity**

* **Serialize: O(w)**
  + The queue stores at most the maximum width w of the tree (O(n) in a balanced tree, O(1) in a skewed tree).
  + The StringBuilder stores the serialized string, but this is part of the output and not counted as extra space.
  + Therefore, the auxiliary space complexity is **O(w)**, where w is the maximum width, ranging from O(1) to O(n).
* **Deserialize: O(w)**
  + The queue stores at most w nodes (O(n) in a balanced tree, O(1) in a skewed tree).
  + The arr array stores the split string (O(n)), but this is necessary for processing.
  + The output tree is not counted as extra space.
  + Therefore, the auxiliary space complexity is **O(w)**, though the array adds O(n) if considered temporary.
* **Overall**: Both methods are **O(w)** for queue space, practically O(n) in worst-case balanced trees.

**Concise Summary of the Approach**

The solution serializes a binary tree into a string using level-order (BFS) traversal, appending node values or "null" for missing nodes. It deserializes the string by splitting it into an array and reconstructing the tree using BFS, assigning left and right children to nodes in order. The approach achieves **O(n)** time complexity for both serialization and deserialization and **O(w)** space complexity (where w is the maximum tree width), efficiently encoding and decoding the tree structure.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as all nodes must be processed for serialization and deserialization.
  + **Clarity**: The BFS approach is intuitive for preserving tree structure, and the queue-based logic is straightforward, making it interview-friendly.
  + **Correctness**: Handles all tree configurations, including sparse or skewed trees, by explicitly encoding null nodes.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use BFS instead of DFS?
    - **Response**: BFS ensures a level-by-level encoding, making deserialization simpler by processing nodes in the same order. DFS (e.g., preorder) is also viable but requires tracking indices differently.
  + **Interviewer might ask**: Why include "null" in the serialized string?
    - **Response**: Null nodes are included to preserve the tree’s structure, ensuring unambiguous reconstruction (e.g., distinguishing between a node with no children vs. one with null children).
  + **Interviewer might ask**: Can you optimize space?
    - **Response**: BFS requires O(w) queue space, which is necessary. A DFS-based approach (preorder) could use O(h) recursion stack space, but the difference is minor, and BFS is clearer for level-order reconstruction.
* **Edge Cases Handled**:
  + Empty tree (root = null): Serializes to "null", deserializes to null.
  + Single node: Serializes as "val,null,null", deserializes correctly.
  + Skewed tree: Correctly encodes and decodes one-sided trees.
  + Sparse tree: Handles null children by including "null" in the string.
  + **Note**: Add input validation for malformed strings in deserialize.
* **Assumptions**:
  + The input tree is a valid binary tree, possibly empty.
  + Node values are integers, and the serialized string is well-formed during deserialization.
  + If these assumptions don’t hold (e.g., invalid string format), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **DFS (Preorder)**: Serialize using preorder traversal (root, left, right), encoding nulls. Deserialize by recursively parsing the string. Time: O(n), Space: O(h) for recursion stack. Simpler space complexity but requires careful index tracking.
  + **Postorder DFS**: Similar to preorder but uses postorder (left, right, root). Time: O(n), Space: O(h). Less intuitive for reconstruction.
  + The BFS approach is preferred for its clarity and straightforward deserialization.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for malformed input in deserialize:

if (data == null || data.isEmpty()) return null;

* + **Use ArrayDeque**: Replace LinkedList with ArrayDeque for better performance:

Queue<TreeNode> q = new ArrayDeque<>();

* + **Trim String**: Remove trailing "null"s in serialize to optimize string length, but ensure deserialization can handle it.
  + **Clarity**: Add comments explaining BFS logic (e.g., “Serialize level by level, including nulls to preserve structure”).
  + **Edge Case Discussion**: In an interview, mention handling of empty trees, single nodes, and sparse trees to show thoroughness.
  + **DFS Option**: Be prepared to code a preorder DFS version if asked:

public class Codec {

*// Serialize using preorder DFS*

public String serialize(TreeNode root) {

if (root == null) return "null";

return root.val + "," + serialize(root.left) + "," + serialize(root.right);

}

*// Deserialize using preorder DFS*

public TreeNode deserialize(String data) {

String[] arr = data.split(",");

Queue<String> q = new LinkedList<>(Arrays.asList(arr));

return dfsDeserialize(q);

}

private TreeNode dfsDeserialize(Queue<String> q) {

String val = q.poll();

if (val.equals("null")) return null;

TreeNode node = new TreeNode(Integer.parseInt(val));

node.left = dfsDeserialize(q);

node.right = dfsDeserialize(q);

return node;

}

}

This achieves O(n) time and O(h) space but requires careful string parsing.

* **Note on Serialization Logic**: The BFS approach serializes the tree level by level, ensuring null nodes are explicitly included to preserve structure. Deserialization mirrors this by reconstructing nodes in the same order, using a queue to track parents and assign children. The comma-separated string format is simple and unambiguous.
* **Date and Time Context**: The date and time (09:02 PM EDT, September 2, 2025) are irrelevant, as the problem is a static algorithm with no time-based logic. In an interview, you could note the code’s portability across contexts.

8. Tries (3 problems)

* Implement Trie Prefix Tree

Commented Code

*// Definition of TrieNode class*

public class TrieNode {

HashMap<Character, TrieNode> children = new HashMap<>(); *// Map of child nodes*

boolean endOfWord = false; *// Flag to mark end of a word*

}

public class PrefixTree {

private TrieNode root; *// Root node of the Trie*

*// Constructor: Initialize empty Trie with root node*

public PrefixTree() {

root = new TrieNode();

}

*// Insert a word into the Trie*

public void insert(String word) {

TrieNode cur = root; *// Start at root*

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

*// Add new node for character if it doesn't exist*

cur.children.putIfAbsent(c, new TrieNode());

cur = cur.children.get(c); *// Move to child node*

}

cur.endOfWord = true; *// Mark end of word*

}

*// Search for an exact word in the Trie*

public boolean search(String word) {

TrieNode cur = root; *// Start at root*

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

*// If character not found, word doesn't exist*

if (!cur.children.containsKey(c)) {

return false;

}

cur = cur.children.get(c); *// Move to child node*

}

return cur.endOfWord; *// Return true if word ends here*

}

*// Check if any word in the Trie starts with the given prefix*

public boolean startsWith(String prefix) {

TrieNode cur = root; *// Start at root*

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

*// If character not found, no word with this prefix*

if (!cur.children.containsKey(c)) {

return false;

}

cur = cur.children.get(c); *// Move to child node*

}

return true; *// Prefix exists in Trie*

}

}

**Time Complexity**

* **Constructor (PrefixTree): O(1)**
  + Initializes an empty TrieNode, which takes constant time.
* **insert Method: O(m)**
  + Iterates through each character of the input word of length m.
  + For each character:
    - Checking and adding to children (putIfAbsent) is O(1) on average for HashMap.
    - Retrieving the child node (get) is O(1) on average.
    - Setting endOfWord is O(1).
  + Total time per insertion is **O(m)**, where m is the length of the word.
* **search Method: O(m)**
  + Iterates through each character of the input word of length m.
  + For each character:
    - Checking containsKey is O(1) on average.
    - Retrieving the child node (get) is O(1) on average.
    - Checking endOfWord is O(1).
  + Total time is **O(m)**, where m is the length of the word.
* **startsWith Method: O(m)**
  + Similar to search, iterates through each character of the input prefix of length m.
  + For each character:
    - Checking containsKey is O(1) on average.
    - Retrieving the child node (get) is O(1) on average.
  + Total time is **O(m)**, where m is the length of the prefix.

**Space Complexity**

* **Space Complexity: O(T), where T is the total number of characters across all inserted words**
  + **Constructor**: Creates one TrieNode with an empty HashMap, using O(1) space.
  + **Trie Storage**: Each inserted word contributes nodes for its characters. In the worst case (no shared prefixes), each character requires a new TrieNode with a HashMap entry. For k words with average length m, the space is O(k \* m) in the worst case, but shared prefixes reduce this significantly in practice (e.g., words like "cat" and "car" share the "ca" nodes). Thus, the space is **O(T)**, where T is the total number of characters across all words.
  + **Auxiliary Space**:
    - insert: Uses a single cur pointer and iterates over the input string, requiring O(1) auxiliary space (excluding the nodes added to the Trie).
    - search and startsWith: Use a single cur pointer, requiring O(1) auxiliary space.
  + The input strings are not counted as extra space.
  + Therefore, the space complexity for the Trie structure is **O(T)**, and auxiliary space for each method is **O(1)**.

**Concise Summary of the Approach**

The solution implements a Trie (prefix tree) to store and query strings efficiently. The TrieNode class uses a HashMap to store child nodes and a boolean to mark word endings. The insert method adds a word by creating nodes for each character, marking the final node as a word end. The search method checks if a word exists by traversing its characters and verifying the final node’s endOfWord flag. The startsWith method checks if a prefix exists by traversing its characters. The approach achieves **O(m)** time complexity for each operation (m is the string length) and **O(T)** space complexity for the Trie, where T is the total characters across all words, making it efficient for string prefix operations.

* Design Add And Search Words Data Structure

Commented Code

*// Definition of TrieNode class*

class TrieNode {

Map<Character, TrieNode> children = new HashMap(); *// Map of child nodes*

boolean word = false; *// Flag to mark end of a word*

public TrieNode() {} *// Constructor*

}

class WordDictionary {

TrieNode trie; *// Root node of the Trie*

*// Constructor: Initialize empty Trie*

public WordDictionary() {

trie = new TrieNode();

}

*// Add a word to the Trie*

public void addWord(String word) {

TrieNode node = trie; *// Start at root*

for (char ch : word.toCharArray()) {

*// Add new node for character if it doesn't exist*

if (!node.children.containsKey(ch)) {

node.children.put(ch, new TrieNode());

}

node = node.children.get(ch); *// Move to child node*

}

node.word = true; *// Mark end of word*

}

*// Helper method to search for a word with wildcards starting from a node*

public boolean searchInNode(String word, TrieNode node) {

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

char ch = word.charAt(i);

*// If character not in children and not '.', word/prefix doesn't exist*

if (!node.children.containsKey(ch)) {

if (ch == '.') {

*// Try all children for wildcard*

for (char x : node.children.keySet()) {

TrieNode child = node.children.get(x);

if (searchInNode(word.substring(i + 1), child)) {

return true; *// Found a match in one branch*

}

}

return false; *// No match found for wildcard*

}

return false; *// Character not found, no match*

} else {

node = node.children.get(ch); *// Move to child node*

}

}

return node.word; *// Return true if word ends here*

}

*// Search for a word (with wildcards) in the Trie*

public boolean search(String word) {

return searchInNode(word, trie);

}

}

**Time Complexity**

* **Constructor (WordDictionary): O(1)**
  + Initializes an empty TrieNode, which takes constant time.
* **addWord Method: O(m)**
  + Iterates through each character of the input word of length m.
  + For each character:
    - Checking and adding to children (containsKey and put) is O(1) on average for HashMap.
    - Retrieving the child node (get) is O(1) on average.
    - Setting word = true is O(1).
  + Total time is **O(m)**, where m is the length of the word.
* **search and searchInNode Methods: O(min(C^m, T))**
  + For a word of length m, a regular character search (no .) is O(m), similar to addWord, as it performs O(1) HashMap operations per character.
  + When the word contains a wildcard (.), at each ., the algorithm explores all children of the current node. In the worst case, with an alphabet size C (e.g., 26 for lowercase letters), each . branches to C children, leading to O(C^m) complexity for a word with many wildcards (exponential in the number of . characters).
  + However, the search is bounded by the total number of nodes in the Trie (T), as it only explores valid paths. Thus, the complexity is **O(min(C^m, T))**, where T is the total number of nodes.
  + In practice, for typical inputs with few wildcards and shared prefixes, the complexity is closer to O(m) for searches without wildcards and O(k \* C) for k wildcards with limited branching.

**Space Complexity**

* **Space Complexity: O(T), where T is the total number of characters across all inserted words**
  + **Constructor**: Creates one TrieNode with an empty HashMap, using O(1) space.
  + **Trie Storage**: Each inserted word contributes nodes for its characters. In the worst case (no shared prefixes), each character requires a new TrieNode with a HashMap entry. For k words with average length m, the space is O(k \* m) in the worst case, but shared prefixes reduce this to **O(T)**, where T is the total number of characters across all words.
  + **Auxiliary Space**:
    - addWord: Uses a single node pointer, requiring O(1) auxiliary space (excluding nodes added to the Trie).
    - search and searchInNode: Use O(m) space due to the recursion stack in the worst case (when handling wildcards, as searchInNode is recursive). For non-wildcard searches, it’s O(1) since it’s iterative within the loop.
    - The word.substring(i + 1) operation in searchInNode creates a new string of length m - i - 1, but this is temporary and bounded by O(m).
  + The input strings are not counted as extra space.
  + Therefore, the space complexity for the Trie structure is **O(T)**, and auxiliary space for search is **O(m)** in the worst case due to recursion.

**Concise Summary of the Approach**

The solution implements a Trie to store words and support searches with a wildcard character (.). The TrieNode uses a HashMap for children and a boolean to mark word endings. addWord inserts a word by creating nodes for each character, marking the final node. search uses a recursive helper searchInNode to traverse the Trie, handling regular characters with direct lookups and wildcards by exploring all children recursively. The approach achieves **O(m)** time for addWord and non-wildcard searches, **O(min(C^m, T))** for wildcard searches, and **O(T)** space for the Trie, efficiently supporting flexible word searches.

* Word Search II

9. Heap / Priority Queue (7 problems)

* Kth Largest Element In a Stream

Commented Code

class KthLargest {

private PriorityQueue<Integer> minHeap; *// Min-heap to store k largest elements*

private int k; *// Size of heap to maintain kth largest*

*// Constructor to initialize with k and initial numbers*

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

this.k = k;

this.minHeap = new PriorityQueue<>(); *// Initialize min-heap*

*// Add each number to heap*

for (int num : nums) {

minHeap.offer(num); *// Insert number*

*// Keep heap size at k by removing smallest if size exceeds k*

if (minHeap.size() > k) {

minHeap.poll();

}

}

}

*// Method to add a new value and return kth largest*

public int add(int val) {

minHeap.offer(val); *// Insert new value*

*// Keep heap size at k by removing smallest if size exceeds k*

if (minHeap.size() > k) {

minHeap.poll();

}

return minHeap.peek(); *// Return smallest in heap (kth largest)*

}

}

**Time Complexity**

* **Constructor: O(n log k)**
  + Initializing the min-heap takes O(1).
  + For each of the n numbers in nums, offer (insert) takes O(log k) in a min-heap of size at most k.
  + If the heap size exceeds k, poll (remove minimum) takes O(log k).
  + In the worst case, each of the n numbers triggers an offer, and up to n - k numbers trigger a poll.
  + Total time is O(n log k) for n insertions, as each operation is O(log k).
* **add Method: O(log k)**
  + Inserting a value (offer) takes O(log k) in a min-heap of size at most k.
  + If the heap size exceeds k, removing the minimum (poll) takes O(log k).
  + Peeking at the minimum (peek) is O(1).
  + Total time per add call is O(log k).
* **Overall**: Constructor is **O(n log k)**, and each add call is **O(log k)**.

**Space Complexity**

* **Space Complexity: O(k)**
  + The min-heap stores at most k elements at any time, using O(k) space.
  + The k variable and loop variables use O(1) space.
  + The input array nums is not stored, so it doesn’t contribute to auxiliary space.
  + Therefore, the space complexity is **O(k)**, where k is the size of the heap.

**Concise Summary of the Approach**

The solution tracks the kth largest element in a stream using a min-heap of size k. The constructor initializes the heap by adding numbers from the input array, maintaining only the k largest by removing the smallest when the size exceeds k. The add method inserts a new value and ensures the heap size remains k, returning the smallest element in the heap (the kth largest). The approach achieves **O(n log k)** time for initialization and **O(log k)** per add call, with **O(k)** space, efficiently maintaining the kth largest element.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: Using a min-heap ensures O(log k) time per add, which is optimal for maintaining the kth largest element dynamically.
  + **Clarity**: The min-heap approach is intuitive, as it naturally keeps the k largest elements, with the kth largest at the root.
  + **Correctness**: Correctly handles streaming data by updating the heap and returning the kth largest element.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a min-heap instead of a max-heap?
    - **Response**: A min-heap of size k keeps the k largest elements, with the kth largest as the smallest (root). A max-heap could track the k smallest, but we’d need to track all elements to find the kth largest, using more space.
  + **Interviewer might ask**: Why keep the heap size at k?
    - **Response**: Maintaining exactly k elements ensures the heap’s root is the kth largest. If the size exceeds k, we remove the smallest to keep only the k largest elements.
  + **Interviewer might ask**: Can you optimize for space or time?
    - **Response**: The min-heap of size k is space-optimal (O(k)). Time could be improved for specific cases (e.g., sorted input), but O(log k) per add is generally optimal for dynamic updates.
* **Edge Cases Handled**:
  + Empty input array (nums = []): Heap remains empty; add will build up to k elements.
  + Single element (nums.length = 1): Returns that element if k = 1.
  + Large k (k > nums.length): Heap stores all elements; add continues building.
  + Negative or duplicate values: Handled correctly by the heap’s comparison.
  + **Note**: Add validation for k <= 0 or null nums for robustness.
* **Assumptions**:
  + 1 ≤ k ≤ nums.length (per typical problem constraints).
  + Input array nums is valid, possibly empty.
  + Node values are integers, possibly negative or duplicates.
  + If these assumptions don’t hold (e.g., invalid k), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Max-Heap for k Smallest**: Use a max-heap to track all elements, but this requires O(n) space and O(log n) per add, less efficient for large streams.
  + **Sorted Array**: Maintain a sorted array of k largest elements. Time: O(n) per add (insertion in sorted array), Space: O(k). Inefficient for dynamic updates.
  + **QuickSelect for Initialization**: Use QuickSelect to find the kth largest initially, then maintain with insertions. Complex and not better than heap for streaming.
  + The min-heap approach is preferred for its balance of time (O(log k) per add) and space (O(k)).
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid inputs:

if (k <= 0 || nums == null) throw new IllegalArgumentException("Invalid input");

* + **Use ArrayDeque for Stack-Based Alternatives**: If implementing an alternative (e.g., for specific constraints), prefer ArrayDeque over legacy Stack.
  + **Clarity**: Add comments explaining heap logic (e.g., “Min-heap maintains k largest elements, root is kth largest”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, large k, and duplicates to show thoroughness.
  + **Optimization**: Consider early stopping in constructor if nums is sorted, but this is rare and not worth the complexity.
  + **Alternative Implementation**: Be prepared to explain a max-heap approach if asked, though it’s less efficient:

class KthLargest {

private PriorityQueue<Integer> maxHeap;

private int k;

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

this.k = k;

this.maxHeap = new PriorityQueue<>((a, b) -> b - a);

for (int num : nums) maxHeap.offer(num);

this.maxHeap = new PriorityQueue<>((a, b) -> b - a); *// Rebuild to store all*

for (int num : nums) maxHeap.offer(num);

*// Extract kth largest by polling k-1 times (not ideal for add)*

}

public int add(int val) {

maxHeap.offer(val);

PriorityQueue<Integer> temp = new PriorityQueue<>((a, b) -> b - a);

for (int i = 0; i < k && !maxHeap.isEmpty(); i++) {

temp.offer(maxHeap.poll());

}

int result = temp.peek();

while (!temp.isEmpty()) maxHeap.offer(temp.poll());

return result;

}

}

This is inefficient (O(n log n) for add), reinforcing the min-heap’s superiority.

* **Note on Min-Heap Logic**: The min-heap maintains the k largest elements, with the smallest of these (the kth largest) at the root. Inserting a new value and removing the smallest when size exceeds k ensures the heap always contains the k largest elements seen. peek returns the kth largest efficiently.
* Last Stone Weight

Commented Code

**e Complexity**

* **Time Complexity: O(n log n)**
  + Initializing the max-heap takes O(1).
  + Adding n stones to the max-heap requires O(log n) per insertion, totaling O(n log n).
  + The while loop runs until at most one stone remains. In the worst case, each iteration reduces the heap size by 1 (or 2 if stones are equal), so there are up to O(n) iterations.
  + Each iteration involves two poll operations (O(log n) each) and potentially one add operation (O(log n)), contributing O(log n) per iteration.
  + Total loop time is O(n log n) for up to n iterations.
  + Therefore, the overall time complexity is **O(n log n)**, dominated by heap operations.

**Space Complexity**

* **Space Complexity: O(n)**
  + The max-heap stores at most n stones, using O(n) space.
  + Variables x, y, and loop counters use O(1) space.
  + The input array stones is not counted as extra space.
  + Therefore, the space complexity is **O(n)**, where n is the number of stones.

**Concise Summary of the Approach**

The solution finds the weight of the last remaining stone after repeatedly smashing the two heaviest stones. It uses a max-heap to efficiently retrieve the two largest stones in each iteration. If the stones differ in weight, their difference is added back to the heap. The process continues until at most one stone remains, returning its weight or 0 if none remain. The approach achieves **O(n log n)** time complexity for heap operations and **O(n)** space complexity for the heap, efficiently simulating the stone-smashing process.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log n) is optimal for this problem, as heap operations ensure efficient access to the largest stones.
  + **Clarity**: The max-heap approach is intuitive, mirroring the problem’s requirement to repeatedly select and smash the heaviest stones.
  + **Correctness**: Correctly handles all cases, including equal stones, single stones, and empty results.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a max-heap?
    - **Response**: A max-heap ensures O(log n) access to the two largest stones, which is critical for efficiency. Sorting the array repeatedly would take O(n log n) per iteration, leading to O(n² log n) overall.
  + **Interviewer might ask**: Why add y - x back to the heap?
    - **Response**: The problem states that if stones x and y (x ≤ y) are smashed, y - x is the resulting weight if they differ. Adding this back simulates the smashing process.
  + **Interviewer might ask**: Can you solve it without a heap?
    - **Response**: Yes, sorting the array initially and processing from the end is possible, but each insertion of the difference requires O(n) time, leading to O(n²) overall. The heap is more efficient.
* **Edge Cases Handled**:
  + Empty array: Returns 0 (heap empty after initialization).
  + Single stone: Returns the stone’s weight (no smashing needed).
  + Two equal stones: Both removed, returns 0.
  + All equal stones: All pairs cancel out, returns 0.
  + **Note**: Add validation for stones null or empty for robustness.
* **Assumptions**:
  + The input array stones is valid, non-null, and contains at least one stone.
  + Stone weights are positive integers.
  + If these assumptions don’t hold (e.g., negative weights), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Sorting-Based**: Sort the array initially (O(n log n)), then process from the end, inserting differences in sorted order. Time: O(n²) due to insertions, Space: O(1) excluding input. Inefficient.
  + **Array with Two Pointers**: Maintain a sorted array and use two pointers to track the largest stones. Time: O(n²) for shifting elements, Space: O(1). Less efficient.
  + The max-heap approach is preferred for its O(n log n) time and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid inputs:

if (stones == null || stones.length == 0) return 0;

* + **Clarity**: Add comments explaining heap logic (e.g., “Max-heap ensures largest stones are accessed efficiently”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single stones, and equal stones to show thoroughness.
  + **Optimization**: Consider using offer instead of add for heap consistency:

maxHeap.offer(stone);

(add and offer are equivalent in PriorityQueue, but offer is preferred for queues.)

* + **Alternative Implementation**: Be prepared to explain a sorting-based approach if asked, though it’s less efficient:

public int lastStoneWeight(int[] stones) {

Arrays.sort(stones);

int i = stones.length - 1;

while (i > 0) {

int y = stones[i], x = stones[i - 1];

if (x == y) {

i -= 2;

} else {

stones[i - 1] = y - x;

i--;

Arrays.sort(stones, 0, i + 1); *// Re-sort subarray*

}

}

return i >= 0 ? stones[i] : 0;

}

This is O(n² log n) due to repeated sorting, making the heap approach superior.

* **Note on Max-Heap Logic**: The max-heap ensures the two largest stones (x and y, with x ≤ y) are efficiently retrieved in O(log n). If x != y, the difference y - x is added back, simulating smashing. The heap maintains at most n elements, shrinking with each iteration, and the final peek or poll gives the last stone’s weight or 0 if empty.
* K Closest Points to Origin

Commented Code

**Commented Code**

class Solution {

*// Method to find the k closest points to the origin (0, 0)*

public int[][] kClosest(int[][] points, int k) {

*// Initialize max-heap to store points, comparing by squared Euclidean distance*

PriorityQueue<int[]> maxHeap = new PriorityQueue<>(

(a, b) -> Integer.compare(b[0] \* b[0] + b[1] \* b[1], a[0] \* a[0] + a[1] \* a[1])

);

*// Process each point*

for (int[] point : points) {

maxHeap.offer(point); *// Add point to heap*

*// Keep heap size at k by removing point with largest distance*

if (maxHeap.size() > k) {

maxHeap.poll();

}

}

*// Build result array from remaining k points*

int[][] res = new int[k][2];

int i = 0;

while (!maxHeap.isEmpty()) {

res[i] = maxHeap.poll(); *// Extract points in ascending distance order*

i++;

}

return res;

}

}

**Time Complexity**

* **Time Complexity: O(n log k)**
  + Initializing the max-heap takes O(1).
  + For each of the n points, inserting into the max-heap (offer) takes O(log k) since the heap size is at most k.
  + If the heap size exceeds k, removing the maximum (poll) takes O(log k).
  + Processing n points with up to n - k removals results in O(n log k) total time for heap operations.
  + Extracting k points from the heap takes O(k log k) (each poll is O(log k)).
  + The dominant term is O(n log k) for n insertions, as k ≤ n.
  + Therefore, the total time complexity is **O(n log k)**.

**Space Complexity**

* **Space Complexity: O(k)**
  + The max-heap stores at most k points, using O(k) space.
  + The result array res stores k points, but this is part of the output and not counted as auxiliary space.
  + Variables like i and loop variables use O(1) space.
  + The input array points is not counted as extra space.
  + Therefore, the auxiliary space complexity is **O(k)**, where k is the number of points to return.

**Concise Summary of the Approach**

The solution finds the k closest points to the origin (0, 0) by maintaining a max-heap of size k, ordered by squared Euclidean distance (x² + y²). For each point, it adds it to the heap and removes the farthest if the size exceeds k. After processing all points, the heap contains the k closest points, which are extracted into a result array. The approach achieves **O(n log k)** time complexity for heap operations and **O(k)** space complexity, efficiently identifying the k closest points.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log k) is optimal for selecting the k smallest distances without sorting all points (which would be O(n log n)).
  + **Clarity**: The max-heap approach is intuitive, maintaining only the k closest points, making it interview-friendly.
  + **Correctness**: Correctly selects the k closest points by comparing squared distances, avoiding floating-point issues with square roots.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a max-heap instead of a min-heap?
    - **Response**: A max-heap of size k keeps the k closest points by removing the farthest when size exceeds k. A min-heap would need to store all n points to find the k smallest, using O(n) space, which is less efficient.
  + **Interviewer might ask**: Why compute squared distance instead of actual distance?
    - **Response**: Squared Euclidean distance (x² + y²) avoids square roots, which are computationally expensive and introduce floating-point precision issues, while preserving the same ordering.
  + **Interviewer might ask**: Can you solve it without a heap?
    - **Response**: Yes, sorting all points by distance takes O(n log n) time, then select the first k. This is simpler but less efficient for large n and small k. Another approach is QuickSelect, averaging O(n) but with O(n²) worst case.
* **Edge Cases Handled**:
  + Empty array (points = []): Returns empty array if k = 0.
  + Single point: Returns the point if k = 1.
  + k = points.length: Returns all points.
  + Points at origin ([0,0]): Correctly handles zero distance.
  + Duplicate points: Heap handles duplicates correctly.
  + **Note**: Add validation for points null or k invalid.
* **Assumptions**:
  + 1 ≤ k ≤ points.length and points is non-null.
  + Each point is a valid [x, y] array with integer coordinates.
  + If these assumptions don’t hold (e.g., invalid k), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Sorting-Based**: Sort points by squared distance and take the first k. Time: O(n log n), Space: O(1) excluding output. Simpler but slower for small k.
  + **QuickSelect**: Use QuickSelect to find the k smallest distances. Average time: O(n), Worst case: O(n²), Space: O(1). Unstable performance but good for one-time selection.
  + **Min-Heap for All Points**: Store all points in a min-heap, extract k smallest. Time: O(n log n + k log n), Space: O(n). Less efficient for large n.
  + The max-heap approach is preferred for its O(n log k) time and O(k) space.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid inputs: if (points == null || k <= 0 || k > points.length) throw new IllegalArgumentException("Invalid input");
  + **Use ArrayDeque for Alternatives**: If implementing a stack-based alternative, prefer ArrayDeque over legacy Stack.
  + **Clarity**: Add comments explaining heap logic (e.g., “Max-heap maintains k closest points by removing farthest”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single points, and k = n to show thoroughness.
  + **Optimization**: Consider a QuickSelect-based approach for one-time selection:

This averages O(n) but has O(n²) worst case and modifies the input array.

* **Note on Max-Heap Logic**: The max-heap maintains the k closest points by comparing squared Euclidean distances. By keeping the size at k and removing the farthest point when needed, the heap ensures the k smallest distances remain. Extracting all points at the end gives the result in ascending distance order due to polling the max-heap.
* Kth Largest Element In An Array

Commented Code

**Commented Code**

class Solution {

*// Method to find the kth largest element in an array*

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

PriorityQueue<Integer> minHeap = new PriorityQueue<>(); *// Min-heap to store k largest elements*

*// Process each number in the array*

for (int num : nums) {

minHeap.offer(num); *// Add number to heap*

*// Keep heap size at k by removing smallest element if size exceeds k*

if (minHeap.size() > k) {

minHeap.poll();

}

}

*// Return the smallest element in heap (kth largest)*

return minHeap.peek();

}

}

**Time Complexity**

* **Time Complexity: O(n log k)**
  + Initializing the min-heap takes O(1).
  + For each of the n numbers in nums, inserting into the min-heap (offer) takes O(log k) since the heap size is at most k.
  + If the heap size exceeds k, removing the smallest element (poll) takes O(log k).
  + Processing n numbers with up to n - k removals results in O(n log k) total time for heap operations.
  + The final peek operation is O(1).
  + Therefore, the total time complexity is **O(n log k)**, where n is the length of the array.

**Space Complexity**

* **Space Complexity: O(k)**
  + The min-heap stores at most k elements, using O(k) space.
  + Loop variables and temporary storage use O(1) space.
  + The input array nums is not counted as extra space.
  + Therefore, the space complexity is **O(k)**, where k is the number of elements to track.

**Concise Summary of the Approach**

The solution finds the kth largest element in an array using a min-heap of size k. It iterates through the array, adding each number to the heap and removing the smallest element if the heap size exceeds k. At the end, the heap’s smallest element (accessed via peek) is the kth largest in the array. The approach achieves **O(n log k)** time complexity for heap operations and **O(k)** space complexity, efficiently identifying the kth largest element.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log k) is efficient for large arrays when k is significantly smaller than n, avoiding full sorting (O(n log n)).
  + **Clarity**: The min-heap approach is straightforward, maintaining the k largest elements with the kth largest at the root, making it interview-friendly.
  + **Correctness**: Correctly identifies the kth largest element, handling duplicates and edge cases.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a min-heap instead of a max-heap?
    - **Response**: A min-heap of size k keeps the k largest elements, with the kth largest as the smallest (root). A max-heap would need to store all n elements to find the kth largest, using O(n) space and O(n log n) time for construction.
  + **Interviewer might ask**: Why maintain a heap of size k?
    - **Response**: Keeping exactly k elements ensures the heap’s root is the kth largest. Removing the smallest element when the size exceeds k maintains the k largest elements efficiently.
  + **Interviewer might ask**: Can you solve it without a heap?
    - **Response**: Yes, sorting the array takes O(n log n) time, then select the (n-k)th element. Alternatively, QuickSelect has an average time of O(n) but O(n²) worst case. The heap approach balances efficiency and simplicity.
* **Edge Cases Handled**:
  + Empty array: Not applicable (problem guarantees 1 ≤ k ≤ nums.length).
  + Single element (nums.length = 1, k = 1): Returns the only element.
  + k = nums.length: Returns the smallest element after processing all numbers.
  + Duplicates: Handled correctly by the heap’s comparison.
  + Negative numbers: Handled seamlessly as heap compares values numerically.
  + **Note**: Add validation for nums null or k invalid for robustness.
* **Assumptions**:
  + 1 ≤ k ≤ nums.length and nums is non-null.
  + Array elements are integers, possibly negative or duplicates.
  + If these assumptions don’t hold (e.g., invalid k), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Sorting-Based**: Sort the array in descending order and return the kth element. Time: O(n log n), Space: O(1) excluding input. Simple but slower for small k.
  + **QuickSelect**: Use QuickSelect to partition the array and find the kth largest element. Average time: O(n), Worst case: O(n²), Space: O(1). Faster on average but less predictable.
  + **Max-Heap for All Elements**: Store all elements in a max-heap and extract the kth largest by polling k times. Time: O(n log n + k log n), Space: O(n). Less efficient for large n.
  + The min-heap approach is preferred for its O(n log k) time and O(k) space, especially when k is small.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid inputs, such as:
    - Null array or invalid k (k <= 0 or k > nums.length).
    - Example condition: “If nums is null or k is invalid, throw an exception.”
  + **Clarity**: Add comments explaining heap logic (e.g., “Min-heap maintains k largest elements, root is kth largest”).
  + **Edge Case Discussion**: In an interview, mention handling of single elements, duplicates, and k = n to show thoroughness.
  + **Use Offer/Poll**: The code already uses offer and poll, which is consistent with PriorityQueue conventions.
* **Note on Min-Heap Logic**: The min-heap maintains the k largest elements by adding each number and removing the smallest when the size exceeds k. The root of the heap is the smallest of the k largest elements, which is the kth largest overall. This approach avoids sorting the entire array, making it efficient for small k.
* Task Scheduler

Commented Code

**Commented Code**

class Solution {

*// Method to find the least number of time units to schedule all tasks*

public int leastInterval(char[] tasks, int n) {

*// Step 1: Count the frequency of each task*

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

for (char task : tasks) {

freqMap.put(task, freqMap.getOrDefault(task, 0) + 1);

}

*// Step 2: Build a max-heap based on task frequencies*

PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

maxHeap.addAll(freqMap.values());

*// Step 3: Process tasks with cooling period*

int time = 0;

while (!maxHeap.isEmpty()) {

List<Integer> temp = new ArrayList<>(); *// Store frequencies temporarily*

*// Process up to n+1 tasks (one cycle)*

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

if (!maxHeap.isEmpty()) {

temp.add(maxHeap.poll()); *// Remove task frequency*

}

}

*// Decrease frequencies and add back non-zero frequencies*

for (int freq : temp) {

if (--freq > 0) {

maxHeap.add(freq); *// Add back for next cycle*

}

}

*// Step 4: Update time*

*// If heap is empty, add number of tasks processed; else add full cycle (n+1)*

time += maxHeap.isEmpty() ? temp.size() : n + 1;

}

return time; *// Return total time units*

}

}

**Time Complexity**

* **Time Complexity: O(m log m + n)**
  + **Step 1**: Building the frequency map takes O(n), where n is the length of the tasks array.
  + **Step 2**: Creating the max-heap from m unique task frequencies (where m ≤ n and m ≤ 26 for letters A-Z) takes O(m log m) using addAll.
  + **Step 3**: The while loop processes tasks in cycles. In each cycle:
    - Up to n + 1 tasks are polled from the heap (O(log m) per poll), and their frequencies are stored in temp (O(1) per task).
    - Frequencies are decremented and added back if non-zero (O(log m) per add).
    - Each cycle processes up to n + 1 tasks, and each task is processed at most f times, where f is the maximum frequency (f ≤ n).
    - In the worst case, there are O(n) total heap operations (since each task is processed until its frequency is 0), each taking O(log m).
    - Thus, the loop contributes O(n log m).
  + **Step 4**: Updating time is O(1) per cycle.
  + Total time is O(n) for the frequency map + O(m log m) for heap initialization + O(n log m) for heap operations. Since m ≤ 26 in this problem (tasks are letters A-Z), log m is effectively constant (log 26 ≈ 4.7), so the time simplifies to **O(n)** practically, but formally **O(m log m + n)**.

**Space Complexity**

* **Space Complexity: O(m)**
  + The frequency map stores at most m unique tasks (m ≤ 26 for letters A-Z), using O(m) space.
  + The max-heap stores at most m frequencies, using O(m) space.
  + The temp list stores up to n + 1 frequencies per cycle, but since n + 1 is at most the number of tasks processed and temp is reused, it’s bounded by O(m) in practice (as only unique task frequencies are processed).
  + Variables like time and loop counters use O(1) space.
  + The input array tasks is not counted as extra space.
  + Therefore, the space complexity is **O(m)**, where m is the number of unique tasks (bounded by 26 in this problem).

**Concise Summary of the Approach**

The solution calculates the minimum time units to schedule tasks with a cooling period of n units between same-task executions. It uses a max-heap to prioritize tasks with the highest remaining frequencies. In each cycle, it processes up to n + 1 tasks, decrements their frequencies, and adds non-zero frequencies back to the heap. Time is incremented by n + 1 per cycle (or fewer if the heap is empty). The approach achieves **O(m log m + n)** time complexity (practically O(n) since m ≤ 26) and **O(m)** space complexity, efficiently scheduling tasks while respecting the cooling period.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) practical time (due to m ≤ 26) is efficient for large task arrays, and the max-heap ensures high-frequency tasks are prioritized.
  + **Clarity**: The heap-based approach clearly models the scheduling process with cooling periods, making it interview-friendly.
  + **Correctness**: Correctly accounts for cooling periods and handles all task distributions.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a max-heap?
    - **Response**: A max-heap prioritizes tasks with the highest frequencies, ensuring the most frequent tasks are scheduled first to minimize idle time and total duration.
  + **Interviewer might ask**: Why increment time by n + 1 or temp.size()?
    - **Response**: Each cycle processes up to n + 1 tasks (to account for cooling periods). If the heap is empty, only the remaining tasks (temp.size()) are needed; otherwise, a full cycle (n + 1) accounts for tasks plus idle time.
  + **Interviewer might ask**: Can you solve it without a heap?
    - **Response**: Yes, a mathematical approach can compute the result using the maximum frequency and idle time formula: max((maxFreq - 1) \* (n + 1) + countMaxFreq, tasks.length). This is O(n) time and O(1) space but requires deriving the formula, which is less intuitive.
* **Edge Cases Handled**:
  + Single task (tasks.length = 1): Returns 1 (no cooling needed).
  + n = 0: Returns tasks.length (no cooling period, tasks executed sequentially).
  + All same tasks: Requires significant idle time (e.g., tasks = [A,A,A], n = 2 returns 7: A \_ \_ A \_ \_ A).
  + No idle time needed: When tasks are diverse enough (e.g., tasks = [A,B,C], n = 2 returns 3).
  + Empty array: Not applicable (problem guarantees non-empty array).
  + **Note**: Add validation for tasks null or n < 0 for robustness.
* **Assumptions**:
  + tasks is non-null with at least one task, and tasks are uppercase letters (A-Z, so m ≤ 26).
  + n ≥ 0 is the cooling period.
  + If these assumptions don’t hold (e.g., invalid n), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Mathematical Formula**: Calculate the result using the formula max((maxFreq - 1) \* (n + 1) + countMaxFreq, tasks.length), where maxFreq is the highest task frequency and countMaxFreq is the number of tasks with that frequency. Time: O(n), Space: O(1) excluding frequency map. Faster but less intuitive and harder to derive in an interview.
  + **Array-Based Greedy**: Sort frequencies and schedule tasks manually, tracking cooling periods. Time: O(n log n), Space: O(1). More complex and less efficient.
  + The max-heap approach is preferred for its clarity and guaranteed O(n) practical performance.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid inputs, such as null array or negative n.
  + **Clarity**: Add comments explaining cooling period logic (e.g., “Each cycle processes up to n + 1 tasks to account for cooling”).
  + **Edge Case Discussion**: In an interview, mention handling of single tasks, n = 0, and all-same tasks to show thoroughness.
  + **Efficiency**: The code is efficient given m ≤ 26, but note that addAll could be replaced with individual offer calls for clarity, though it doesn’t change complexity.
* **Note on Heap Logic**: The max-heap prioritizes tasks with the highest frequencies to minimize idle time. Each cycle processes up to n + 1 tasks to respect the cooling period, reducing frequencies and re-adding non-zero ones. Time increments by n + 1 for full cycles or fewer if tasks are exhausted, ensuring the minimum time is computed.
* Design Twitter

Commented Code

public class Twitter {

private static int timeStamp = 0; *// Global timestamp for tweets*

*// User class to represent each user in Twitter*

private class User {

int id; *// User ID*

Set<Integer> followed; *// Set of followed user IDs*

Tweet tweetHead; *// Head of user's tweet linked list*

public User(int id) {

this.id = id;

followed = new HashSet<>();

follow(id); *// User follows themselves*

tweetHead = null;

}

public void follow(int id) {

followed.add(id); *// Add user to followed set*

}

public void unfollow(int id) {

if (id != this.id) {

followed.remove(id); *// Remove user from followed set, except self*

}

}

public void post(int id) {

Tweet newTweet = new Tweet(id); *// Create new tweet*

newTweet.next = tweetHead; *// Prepend to tweet list*

tweetHead = newTweet;

}

}

*// Tweet class to represent each tweet*

private class Tweet {

int id; *// Tweet ID*

int time; *// Timestamp of tweet*

Tweet next; *// Next tweet in user's tweet list*

public Tweet(int id) {

this.id = id;

time = timeStamp++; *// Assign global timestamp*

next = null;

}

}

private Map<Integer, User> userMap; *// Map of user ID to User object*

*/\*\* Initialize your data structure here. \*/*

public Twitter() {

userMap = new HashMap<>(); *// Initialize user map*

}

*/\*\* Compose a new tweet. \*/*

public void postTweet(int userId, int tweetId) {

if (!userMap.containsKey(userId)) {

User newUser = new User(userId); *// Create new user if not exists*

userMap.put(userId, newUser);

}

userMap.get(userId).post(tweetId); *// Post tweet for user*

}

*/\*\**

*\* Retrieve the 10 most recent tweet IDs in the user's news feed.*

*\* Each item in the news feed must be posted by users who the user followed or by the user themselves.*

*\* Tweets must be ordered from most recent to least recent.*

*\*/*

public List<Integer> getNewsFeed(int userId) {

List<Integer> newsFeed = new LinkedList<>(); *// List for recent tweets*

if (!userMap.containsKey(userId)) return newsFeed; *// Return empty if user doesn't exist*

Set<Integer> followedUsers = userMap.get(userId).followed; *// Get followed users*

PriorityQueue<Tweet> tweetHeap = new PriorityQueue<>(

followedUsers.size(), (a, b) -> b.time - a.time); *// Max-heap by timestamp*

*// Add most recent tweet of each followed user to heap*

for (int user : followedUsers) {

Tweet tweet = userMap.get(user).tweetHead;

if (tweet != null) {

tweetHeap.add(tweet);

}

}

*// Extract up to 10 most recent tweets*

int count = 0;

while (!tweetHeap.isEmpty() && count < 10) {

Tweet tweet = tweetHeap.poll(); *// Get most recent tweet*

newsFeed.add(tweet.id); *// Add to news feed*

count++;

if (tweet.next != null) {

tweetHeap.add(tweet.next); *// Add next tweet from same user*

}

}

return newsFeed;

}

*/\*\* Follower follows a followee. \*/*

public void follow(int followerId, int followeeId) {

if (!userMap.containsKey(followerId)) {

User newUser = new User(followerId); *// Create follower if not exists*

userMap.put(followerId, newUser);

}

if (!userMap.containsKey(followeeId)) {

User newUser = new User(followeeId); *// Create followee if not exists*

userMap.put(followeeId, newUser);

}

userMap.get(followerId).follow(followeeId); *// Add followee to follower's set*

}

*/\*\* Follower unfollows a followee. \*/*

public void unfollow(int followerId, int followeeId) {

if (userMap.containsKey(followerId) && followerId != followeeId) {

userMap.get(followerId).unfollow(followeeId); *// Remove followee if valid*

}

}

}

**Time Complexity**

* **Constructor: O(1)**
  + Initializing the HashMap takes constant time.
* **postTweet: O(1)**
  + Checking and adding a user to userMap is O(1) (HashMap operations).
  + Creating a new Tweet and prepending it to the user’s tweet list is O(1).
  + Total: **O(1)**.
* **getNewsFeed: O(f log f + 10 log f)**
  + Retrieving the followed users set is O(1).
  + Adding the most recent tweet of each followed user (f users) to the max-heap takes O(f log f) for f insertions.
  + Extracting up to 10 tweets involves poll (O(log f)) and potentially adding the next tweet (O(log f)). This happens up to 10 times, contributing O(10 log f).
  + Total: **O(f log f)**, where f is the number of followed users (simplified, as the constant 10 is small).
* **follow: O(1)**
  + Creating new users (if needed) and adding to userMap is O(1).
  + Adding a followee to the HashSet is O(1).
  + Total: **O(1)**.
* **unfollow: O(1)**
  + Checking user existence and removing from the HashSet is O(1).
  + Total: **O(1)**.

**Space Complexity**

* **Space Complexity: O(U + T + F)**
  + userMap stores U users, each with a User object containing a HashSet of followed users and a linked list of tweets. Total space for users is O(U).
  + Each user’s HashSet stores up to F followees (total follow relationships across all users).
  + The tweet linked list stores T tweets across all users, with each Tweet object using O(1) space.
  + **Constructor**: O(1) for the empty HashMap.
  + **postTweet**: O(1) per call, but accumulates O(T) for all tweets over time.
  + **getNewsFeed**: O(f) for the max-heap (storing up to f tweets) and O(1) for the output list (up to 10 tweets). Output is not counted as auxiliary space.
  + **follow/unfollow**: O(1) per call, but accumulates O(F) for all follow relationships.
  + Total: **O(U + T + F)**, where U is the number of users, T is the total number of tweets, and F is the total number of follow relationships.

**Concise Summary of the Approach**

The solution implements a Twitter system with user follow, tweet posting, and news feed retrieval. It uses a HashMap to store users, each with a HashSet of followed users and a linked list of their tweets. The postTweet method prepends a tweet with a global timestamp. The getNewsFeed method uses a max-heap to merge the most recent tweets from followed users, returning up to 10 in descending time order. The follow and unfollow methods manage follow relationships. The approach achieves **O(1)** time for postTweet, follow, and unfollow, **O(f log f)** for getNewsFeed, and **O(U + T + F)** space, efficiently modeling Twitter’s functionality.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(1) for most operations (postTweet, follow, unfollow) and O(f log f) for getNewsFeed are efficient for typical use cases.
  + **Clarity**: The data structure (HashMap, HashSet, linked list, max-heap) clearly models users, follows, and tweets, making it interview-friendly.
  + **Correctness**: Correctly handles tweet ordering, follow relationships, and edge cases like non-existent users.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a max-heap for getNewsFeed?
    - **Response**: A max-heap efficiently merges the most recent tweets from multiple users’ tweet lists, ensuring the top 10 tweets are retrieved in descending time order with O(log f) per operation.
  + **Interviewer might ask**: Why store tweets in a linked list?
    - **Response**: A linked list allows O(1) prepending of new tweets and easy traversal for accessing recent tweets, which is ideal for chronological ordering.
  + **Interviewer might ask**: Can you optimize getNewsFeed for frequent calls?
    - **Response**: Caching recent news feeds or precomputing merged tweet lists could help, but it increases space complexity and requires invalidation on new tweets or follows. The current heap-based approach balances time and space.
* **Edge Cases Handled**:
  + Non-existent user: Handled by returning empty news feed or creating users in postTweet and follow.
  + Self-follow/unfollow: Users follow themselves by default; unfollow prevents self-removal.
  + No tweets: Empty news feed returned for users with no tweets or followed users.
  + Single user/tweet: Correctly handles minimal cases.
  + Large number of followed users: Heap handles efficiently with O(f log f).
  + **Note**: Add validation for invalid user IDs or tweet IDs if required.
* **Assumptions**:
  + User IDs and tweet IDs are valid integers.
  + getNewsFeed returns up to 10 tweets from followed users or the user themselves.
  + No duplicate tweets or invalid follow relationships (e.g., negative IDs).
  + If these assumptions don’t hold, clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **ArrayList for Tweets**: Store tweets in an ArrayList per user instead of a linked list. Time: Same, but appending takes O(1) at the end, requiring reverse traversal for recent tweets, which is less intuitive.
  + **Sorted List for News Feed**: Maintain a sorted list of all tweets from followed users, updated on postTweet or follow. Time: O(T log T) for updates, Space: O(T). Inefficient for dynamic updates.
  + **Database-Like Approach**: Store tweets and follows in a relational structure and query for news feed. Time: Depends on query optimization, Space: O(U + T + F). More complex and not suited for in-memory design.
  + The current approach is preferred for its balance of simplicity and efficiency.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid user IDs or tweet IDs (e.g., negative values).
  + **Clarity**: Add comments explaining the max-heap’s role (e.g., “Max-heap merges recent tweets from followed users”).
  + **Edge Case Discussion**: In an interview, mention handling of non-existent users, self-follows, and empty tweet lists to show thoroughness.
  + **Efficiency**: Note that PriorityQueue is efficient, but ArrayDeque could be considered for alternative stack-based implementations (though not needed here).
* **Note on Design Logic**: The User class encapsulates follow relationships and tweets, with a linked list for chronological tweet storage. The Tweet class uses a global timestamp to order tweets. The getNewsFeed method leverages a max-heap to efficiently merge and sort tweets from multiple users, ensuring the 10 most recent are returned. The HashMap and HashSet enable fast user and follow lookups.
* Find Median From Data Stream

Commented Code

public class MedianFinder {

*// Max-heap for the lower half of numbers*

private PriorityQueue<Integer> lo = new PriorityQueue<>((a, b) -> b - a);

*// Min-heap for the upper half of numbers*

private PriorityQueue<Integer> hi = new PriorityQueue<>();

*// Adds a number into the data structure*

public void addNum(int num) {

lo.offer(num); *// Add to max-heap (lower half)*

*// Balance by moving largest from max-heap to min-heap*

hi.offer(lo.poll());

*// Ensure max-heap has equal or one more element than min-heap*

if (lo.size() < hi.size()) {

lo.offer(hi.poll()); *// Move smallest from min-heap to max-heap*

}

}

*// Returns the median of current data stream*

public double findMedian() {

*// If uneven size, max-heap has one extra element (return its top)*

*// If even size, average the tops of max-heap and min-heap*

return lo.size() > hi.size() ? lo.peek() : (lo.peek() + hi.peek()) \* 0.5;

}

}

**Time Complexity**

* **addNum: O(log n)**
  + Inserting into the max-heap (lo.offer) takes O(log n), where n is the total number of elements in both heaps.
  + Removing the largest from the max-heap (lo.poll) takes O(log n).
  + Inserting into the min-heap (hi.offer) takes O(log n).
  + If balancing is needed, removing from the min-heap (hi.poll) and inserting into the max-heap (lo.offer) each take O(log n).
  + Total: O(log n) per call, as there are at most three heap operations.
* **findMedian: O(1)**
  + Peeking at the top of the max-heap (lo.peek) is O(1).
  + If sizes are equal, peeking at the min-heap (hi.peek) and averaging is O(1).
  + Total: O(1).
* **Overall**: addNum is **O(log n)**, and findMedian is **O(1)**.

**Space Complexity**

* **Space Complexity: O(n)**
  + The max-heap (lo) stores up to ⌈n/2⌉ elements, and the min-heap (hi) stores up to ⌊n/2⌋ elements, where n is the total number of elements added.
  + The heaps together use O(n) space.
  + No additional data structures are used beyond the heaps.
  + Therefore, the space complexity is **O(n)**.

**Concise Summary of the Approach**

The solution maintains the median of a data stream using two heaps: a max-heap (lo) for the lower half of numbers and a min-heap (hi) for the upper half. The addNum method adds a number to the max-heap, balances it by moving the largest to the min-heap, and ensures the max-heap has at most one more element than the min-heap. The findMedian method returns the top of the max-heap if sizes are unequal or the average of both heap tops if equal. The approach achieves **O(log n)** time for addNum, **O(1)** for findMedian, and **O(n)** space, efficiently tracking the median.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(log n) for adding numbers and O(1) for finding the median are optimal for a dynamic stream.
  + **Clarity**: The two-heap approach clearly separates lower and upper halves, making it intuitive and interview-friendly.
  + **Correctness**: Maintains the median accurately by ensuring balanced heaps and correct ordering.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use two heaps?
    - **Response**: Two heaps ensure the median is always accessible in O(1) time (top of one or both heaps). The max-heap keeps the lower half (smaller numbers), and the min-heap keeps the upper half, with sizes balanced to maintain the median.
  + **Interviewer might ask**: Why add to max-heap first?
    - **Response**: Adding to the max-heap first simplifies balancing. The largest element is moved to the min-heap to ensure the upper half contains larger numbers, and size adjustments keep the heaps balanced.
  + **Interviewer might ask**: Can you solve it without heaps?
    - **Response**: A sorted array or list allows O(n) insertion and O(1) median retrieval, but it’s inefficient for large streams. A balanced BST could achieve O(log n) for both operations but is more complex to implement.
* **Edge Cases Handled**:
  + Single number: Max-heap has one element, median is that element.
  + Even number of elements: Averages tops of both heaps.
  + Odd number of elements: Returns top of max-heap (which has one extra element).
  + Negative numbers or duplicates: Handled correctly by heap ordering.
  + **Note**: The code assumes at least one number exists for findMedian (problem guarantees non-empty stream for median queries).
* **Assumptions**:
  + The stream contains valid integers, possibly negative or duplicates.
  + findMedian is called after at least one addNum.
  + If these assumptions don’t hold (e.g., empty stream), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Sorted List**: Maintain a sorted list, insert new numbers in sorted order (O(n)), and find the median in O(1). Time: O(n) for insertion, Space: O(n). Inefficient for large streams.
  + **Balanced BST**: Use a self-balancing BST (e.g., AVL or Red-Black tree) to maintain sorted order. Time: O(log n) for insertion and median, Space: O(n). More complex to implement.
  + **Multiset with Iterators**: Use a multiset (sorted container) to store numbers and track the median with iterators. Time: O(log n) for insertion, O(1) for median, Space: O(n). Complex to maintain median pointers.
  + The two-heap approach is preferred for its simplicity and optimal time complexity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for findMedian on empty heaps, though the problem typically guarantees valid calls.
  + **Clarity**: Add comments explaining heap roles (e.g., “Max-heap stores lower half, min-heap stores upper half for median access”).
  + **Edge Case Discussion**: In an interview, mention handling of single numbers, even/odd counts, and duplicates to show thoroughness.
  + **Efficiency**: Note that PriorityQueue is efficient, but the comparator for lo could be written as (a, b) -> b - a for clarity (already done).
* **Note on Two-Heap Logic**: The max-heap (lo) stores the lower half of numbers (largest at the root), and the min-heap (hi) stores the upper half (smallest at the root). New numbers are added to lo, and balancing ensures lo has at most one more element than hi. The median is either lo.peek() (odd count) or the average of lo.peek() and hi.peek() (even count), ensuring O(1) retrieval.

10. Backtracking (9 problems)

* Subsets

Commented Code  
class Solution {

*// Method to generate all possible subsets of the input array*

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

List<List<Integer>> result = new ArrayList<>(); *// List to store all subsets*

generateSubsets(0, nums, new ArrayList<>(), result); *// Start backtracking*

return result; *// Return all subsets*

}

*// Helper method for backtracking to generate subsets*

private void generateSubsets(int index, int[] nums, List<Integer> current, List<List<Integer>> result) {

result.add(new ArrayList<>(current)); *// Add current subset to result*

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

current.add(nums[i]); *// Include current element*

generateSubsets(i + 1, nums, current, result); *// Recurse with next index*

current.remove(current.size() - 1); *// Backtrack by removing element*

}

}

}

**Time Complexity**

* **Time Complexity: O(2^n)**
  + The algorithm generates all possible subsets of an array of length n. There are 2^n subsets, as each element can either be included or excluded.
  + The backtracking process explores each subset:
    - At each index i, the algorithm decides to include or exclude nums[i], leading to a binary tree of decisions with depth n.
    - Each subset is added to result in O(k) time, where k is the subset’s size (up to n), for copying the current list.
  + The total number of subsets is 2^n, and the work per subset (copying current) is O(n) in the worst case.
  + Thus, the total time complexity is **O(2^n)** for generating all subsets and copying them.
  + This is optimal, as generating all 2^n subsets is required.

**Space Complexity**

* **Space Complexity: O(2^n \* n)**
  + **Output Space**: The result list stores all 2^n subsets, with each subset having up to n elements. This requires O(2^n \* n) space for the output, not counted as auxiliary space in some analyses.
  + **Auxiliary Space**:
    - The recursion stack can go up to depth n (one recursive call per element), requiring O(n) space.
    - The current list stores up to n elements, requiring O(n) space.
    - The input nums array is not counted as extra space.
  + No additional data structures scale beyond the recursion stack and current list.
  + Therefore, the auxiliary space complexity is **O(n)** for the recursion stack and current list, and the total space including output is **O(2^n \* n)**.

**Concise Summary of the Approach**

The solution generates all possible subsets of an integer array using backtracking. It starts with an empty subset and, for each element, recursively decides to include or exclude it, adding each valid subset to the result. The generateSubsets helper method tracks the current index and builds subsets incrementally, backtracking by removing elements after exploration. The approach achieves **O(2^n)** time complexity to generate all subsets and **O(n)** auxiliary space, efficiently producing the power set of the input array.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(2^n) time is optimal, as all 2^n subsets must be generated, and O(n) auxiliary space is minimal.
  + **Clarity**: The backtracking approach is intuitive, mimicking the decision to include or exclude each element, making it interview-friendly for demonstrating recursion.
  + **Correctness**: Correctly generates all subsets, including the empty set, without duplicates.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why add the subset at the start of generateSubsets?
    - **Response**: Adding the current subset at the start captures the state before including any further elements, ensuring all subsets (including the empty one) are included. This covers the "exclude" choice implicitly for remaining elements.
  + **Interviewer might ask**: Why use i + 1 in the recursive call?
    - **Response**: Using i + 1 ensures elements are considered in order and prevents revisiting earlier elements, avoiding duplicates and maintaining subset order (e.g., [1,2] but not [2,1]).
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative approach can use a binary counter from 0 to 2^n - 1, where each bit represents including/excluding an element. Time: O(2^n), Space: O(2^n \* n). The recursive approach is often clearer for interviews.
* **Edge Cases Handled**:
  + Empty array: Returns [[]] (empty subset).
  + Single element: Returns [[], [1]] for nums = [1].
  + Duplicate elements: Handled correctly, as the algorithm processes elements by index (e.g., nums = [1,1] → [[], [1], [1], [1,1]], though problem typically assumes distinct elements).
  + Large array: Scales within constraints (e.g., n ≤ 10).
  + **Note**: Add validation for null input for robustness.
* **Assumptions**:
  + nums is non-null and contains integers (typically distinct, per problem constraints).
  + Result must include all possible subsets in any order.
  + If these assumptions don’t hold (e.g., null array), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative (Bitmask)**: Use a loop from 0 to 2^n - 1, where each bit determines whether to include an element. Time: O(2^n), Space: O(2^n \* n). Simple but less intuitive.
  + **Cascading**: Start with an empty subset and, for each element, duplicate existing subsets and add the element to the copies. Time: O(2^n), Space: O(2^n \* n). Iterative but memory-intensive.
  + **BFS**: Use a queue to build subsets level by level. Time: O(2^n), Space: O(2^n \* n). More complex and unnecessary.
  + The current recursive approach is preferred for its clarity and elegance.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null nums, e.g., “If nums is null, return [[]] or throw an exception.”
  + **Clarity**: Add comments explaining backtracking choices (e.g., “Include/exclude each element to generate all subsets”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single elements, and duplicates (if applicable) to show thoroughness.
  + **Efficiency**: The approach is optimal, but note that the iterative bitmask approach could be discussed as an alternative.
* **Note on Backtracking Logic**: The algorithm builds subsets by making a decision at each index: include the current element and recurse, then backtrack by removing it to explore excluding it. Adding the current subset at the start of each call captures all possible combinations up to that point, ensuring all 2^n subsets are generated systematically.
* Combination Sum

Commented Code

class Solution {

*// Method to find all unique combinations of candidates that sum to target*

public List<List<Integer>> combinationSum(int[] candidates, int target) {

List<List<Integer>> res = new ArrayList<List<Integer>>(); *// List to store all valid combinations*

List<Integer> combination = new ArrayList<>(); *// Current combination being built*

backTrack(target, res, combination, 0, candidates); *// Start backtracking*

return res; *// Return all valid combinations*

}

*// Helper method for backtracking to generate combinations*

public void backTrack(int target, List<List<Integer>> res, List<Integer> combination,

int start, int[] candidates) {

*// Base case: if target is reached, add current combination to result*

if (target == 0) {

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

}

*// Base case: if target is negative, stop this branch*

else if (target < 0) {

return;

}

*// Try each candidate starting from index 'start'*

for (int i = start; i < candidates.length; i++) {

combination.add(candidates[i]); *// Include candidate*

backTrack(target - candidates[i], res, combination, i, candidates); *// Recurse with reduced target*

combination.remove(combination.size() - 1); *// Backtrack by removing candidate*

}

}

}

**Time Complexity**

* **Time Complexity: O(N^(T/M))**
  + Let N be the length of candidates, T be the target, and M be the minimum value in candidates.
  + The backtracking process forms a tree where each node represents choosing a candidate, and the depth is bounded by the number of candidates needed to reach target.
  + In the worst case, the tree’s depth is approximately T/M (e.g., if all candidates are M, you need T/M selections to reach or exceed target).
  + At each level, there are up to N choices (candidates from index start onward), leading to a branching factor of N.
  + The total number of nodes in the tree is roughly N^(T/M) in the worst case, as each path explores combinations of candidates.
  + For each valid combination, copying the combination list to res takes O(T/M) (maximum combination length), but this is dominated by the number of nodes explored.
  + Thus, the time complexity is approximately **O(N^(T/M))**.
  + For typical inputs (e.g., T up to 30, N up to 40, M ≥ 1), the complexity is manageable within constraints but exponential in nature.

**Space Complexity**

* **Space Complexity: O(N \* N^(T/M))**
  + **Output Space**: The res list stores all valid combinations. In the worst case, the number of combinations is bounded by the number of ways to partition T using N candidates, roughly O(N^(T/M)), with each combination having up to T/M elements. This requires O(N^(T/M) \* T/M) space for the output, not counted as auxiliary space in some analyses.
  + **Auxiliary Space**:
    - The recursion stack can go up to depth T/M (maximum number of candidates in a combination), requiring O(T/M) space.
    - The combination list stores up to T/M elements, requiring O(T/M) space.
    - The input candidates array is not counted as extra space.
  + No additional data structures scale beyond these.
  + Therefore, the auxiliary space complexity is **O(T/M)** for the recursion stack and combination list, and the total space including output is **O(N^(T/M) \* T/M)**.

**Concise Summary of the Approach**

The solution finds all unique combinations of numbers from candidates that sum to target, allowing reuse of elements. It uses backtracking to build combinations incrementally, starting from an index to ensure ordered selections. When the target is reached, the current combination is added to the result; if the target becomes negative, the branch is pruned. The approach achieves **O(N^(T/M))** time complexity to explore all possible combinations and **O(T/M)** auxiliary space, efficiently handling reuse and producing all valid combinations.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(N^(T/M)) is optimal for exploring all possible combinations, given the exponential nature of the problem.
  + **Clarity**: The backtracking approach is intuitive, systematically trying each candidate with reuse, making it interview-friendly for demonstrating recursion.
  + **Correctness**: Correctly generates all combinations that sum to target, allowing reuse and respecting the problem constraints.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why allow reuse of candidates (i instead of i+1 in recursive call)?
    - **Response**: The problem allows using the same candidate multiple times in a combination. Using i in the recursive call allows reusing the current candidate, while start ensures we don’t revisit earlier indices unnecessarily.
  + **Interviewer might ask**: Why check target < 0?
    - **Response**: If target becomes negative, the current combination overshoots the target, so the branch is invalid and can be pruned to avoid unnecessary exploration.
  + **Interviewer might ask**: Can you optimize it further?
    - **Response**: The backtracking approach is near-optimal, but sorting candidates and pruning branches where candidates[i] > target could reduce exploration. This is a minor optimization, as the exponential nature remains.
* **Edge Cases Handled**:
  + Empty candidates array: Returns [] (handled by loop not executing).
  + Target = 0: Returns [[]] if no candidates are added (handled by base case).
  + No valid combinations: Returns [] (e.g., candidates = [2], target = 1).
  + Single candidate: Handles reuse (e.g., candidates = [2], target = 4 → [[2,2]]).
  + Large target: Scales within constraints (e.g., target ≤ 30, candidates.length ≤ 40).
  + **Note**: Add validation for null input or invalid target for robustness.
* **Assumptions**:
  + candidates is non-null, contains positive integers, and may contain duplicates.
  + target is a positive integer.
  + Elements can be reused, and combinations must sum exactly to target.
  + If these assumptions don’t hold (e.g., null array, negative target), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Dynamic Programming**: Use a DP table to build combinations for each sum up to target. Time: O(N \* T), Space: O(T + output). Less intuitive and may use more space for large T.
  + **Iterative BFS**: Use a queue to build combinations level by level, tracking remaining target and start index. Time: O(N^(T/M)), Space: O(N^(T/M)). More complex and memory-intensive.
  + **Sorted Pruning**: Sort candidates and skip candidates where candidates[i] > target. Time: O(N^(T/M)), Space: O(T/M). Minor optimization but similar logic.
  + The current approach is preferred for its clarity and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null candidates or invalid target, e.g., “If candidates is null or target ≤ 0, return [] or throw an exception.”
  + **Clarity**: Add comments explaining reuse (e.g., “Allow reuse by passing i in recursive call”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, zero target, and no valid combinations to show thoroughness.
  + **Efficiency**: The approach is near-optimal, but sorting candidates could enable early pruning for large values.
* **Note on Backtracking Logic**: The algorithm builds combinations by selecting candidates from index start onward, allowing reuse by passing i in recursive calls. It tracks the remaining target and adds valid combinations when target = 0. Negative targets prune invalid branches. The start index ensures ordered combinations, avoiding duplicates in the result.
* Permutations

Commented Code

public class Solution {

*// Method to generate all possible permutations of the input array*

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

List<List<Integer>> result = new ArrayList<>(); *// List to store all permutations*

boolean[] used = new boolean[nums.length]; *// Track used elements*

backtrack(result, new ArrayList<>(), nums, used); *// Start backtracking*

return result; *// Return all permutations*

}

*// Helper method for backtracking to generate permutations*

private void backtrack(List<List<Integer>> result, List<Integer> current, int[] nums, boolean[] used) {

*// Base case: if current permutation is complete, add to result*

if (current.size() == nums.length) {

result.add(new ArrayList<>(current));

return;

}

*// Try each unused element*

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

if (!used[i]) { *// If element is not used*

current.add(nums[i]); *// Include element*

used[i] = true; *// Mark as used*

backtrack(result, current, nums, used); *// Recurse*

used[i] = false; *// Backtrack: unmark as used*

current.remove(current.size() - 1); *// Backtrack: remove element*

}

}

}

}

**Time Complexity**

* **Time Complexity: O(n!)**
  + The algorithm generates all possible permutations of an array of length n. There are n! permutations for n distinct elements.
  + The backtracking process explores a decision tree:
    - At the first position, there are n choices.
    - At the second position, n-1 choices, and so on, down to 1 choice.
    - This results in n \* (n-1) \* ... \* 1 = n! permutations.
  + For each permutation:
    - Adding to result involves copying the current list of size up to n, which is O(n).
    - Other operations (checking used, adding/removing from current, updating used) are O(1) per step.
  + The total work is n! permutations \* O(n) per permutation, giving **O(n \* n!)**. However, the dominant factor is the number of permutations, often simplified to **O(n!)** in permutation problems.
  + This is optimal, as generating all n! permutations is required.

**Space Complexity**

* **Space Complexity: O(n \* n!)**
  + **Output Space**: The result list stores all n! permutations, each of length n, requiring O(n \* n!) space. This is part of the output and not counted as auxiliary space in some analyses.
  + **Auxiliary Space**:
    - The recursion stack can go up to depth n (one call per element in a permutation), requiring O(n) space.
    - The current list stores up to n elements, requiring O(n) space.
    - The used array is of size n, requiring O(n) space.
    - The input nums array is not counted as extra space.
  + No additional data structures scale beyond these.
  + Therefore, the auxiliary space complexity is **O(n)** for the recursion stack, current list, and used array, and the total space including output is **O(n \* n!)**.

**Concise Summary of the Approach**

The solution generates all possible permutations of an integer array using backtracking. It maintains a used array to track which elements are included in the current permutation and builds permutations incrementally in the current list. For each position, it tries each unused element, marks it as used, recurses, and backtracks by unmarking and removing the element. Complete permutations are added to the result. The approach achieves **O(n!)** time complexity to generate all permutations and **O(n)** auxiliary space, efficiently producing the full set of permutations.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n!) time is optimal, as all n! permutations must be generated, and O(n) auxiliary space is minimal.
  + **Clarity**: The backtracking approach with a used array is a standard, clear way to generate permutations, making it interview-friendly for demonstrating recursion and state management.
  + **Correctness**: Correctly generates all permutations without duplicates, handling distinct elements efficiently.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a used array?
    - **Response**: The used array tracks which elements have been included in the current permutation, preventing duplicates and ensuring each element is used exactly once per permutation.
  + **Interviewer might ask**: Why add to result only when current.size() == nums.length?
    - **Response**: This ensures only complete permutations (using all n elements) are added, as partial permutations are not valid for the result.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative approach (e.g., Heap’s algorithm or next-permutation method) can generate permutations in O(n!) time and O(n) space. However, the recursive backtracking approach is often clearer and more intuitive for interviews.
* **Edge Cases Handled**:
  + Empty array: Returns [[]] (empty permutation).
  + Single element: Returns [[1]] for nums = [1].
  + Two elements: Returns [[1,2], [2,1]] for nums = [1,2].
  + Large array: Scales within constraints (e.g., n ≤ 6).
  + **Note**: Add validation for null input for robustness. The problem typically assumes distinct elements, but duplicates would require clarification.
* **Assumptions**:
  + nums is non-null and contains distinct integers.
  + Result must include all possible permutations in any order.
  + If these assumptions don’t hold (e.g., null array or duplicates), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Heap’s Algorithm**: Iteratively generate permutations by swapping elements systematically. Time: O(n!), Space: O(n). Iterative but less intuitive.
  + **Next Permutation**: Generate permutations in lexicographic order by finding the next permutation iteratively. Time: O(n!), Space: O(n). More complex for generating all permutations.
  + **DFS without Used Array**: Swap elements in the array to generate permutations, restoring after each recursion. Time: O(n!), Space: O(n). Slightly different backtracking style.
  + The current approach is preferred for its clarity and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null nums, e.g., “If nums is null, return [[]] or throw an exception.”
  + **Clarity**: Add comments explaining backtracking choices (e.g., “Try each unused element to build permutations”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single elements, and potential duplicates to show thoroughness.
  + **Efficiency**: The approach is optimal, but note that iterative methods like Heap’s algorithm could be discussed as alternatives.
* **Note on Backtracking Logic**: The algorithm builds permutations by selecting each unused element at the current position, marking it as used, and recursing to fill the next position. Backtracking restores the state by unmarking and removing the element, ensuring all n! permutations are generated systematically without duplicates.
* Subsets II

Commented Code

class Solution {

*// Method to generate all unique subsets of the input array with duplicates*

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

List<List<Integer>> result = new ArrayList<>(); *// List to store all unique subsets*

Arrays.sort(nums); *// Sort the array to handle duplicates*

backtrack(result, new ArrayList<>(), nums, 0); *// Start backtracking*

return result; *// Return all unique subsets*

}

*// Helper method for backtracking to generate subsets*

private void backtrack(List<List<Integer>> result, List<Integer> current, int[] nums, int start) {

result.add(new ArrayList<>(current)); *// Add current subset to result*

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

*// Skip duplicates to avoid duplicate subsets*

if (i > start && nums[i] == nums[i - 1]) {

continue;

}

current.add(nums[i]); *// Include current element*

backtrack(result, current, nums, i + 1); *// Recurse with next index*

current.remove(current.size() - 1); *// Backtrack by removing element*

}

}

}

**Time Complexity**

* **Time Complexity: O(2^n)**
  + The algorithm generates all possible subsets of an array of length n. In the worst case (no duplicates), there are 2^n subsets, as each element can be included or excluded.
  + **Sorting**: Sorting the input array (Arrays.sort(nums)) takes O(n log n).
  + **Backtracking**:
    - The backtracking process explores a decision tree where each element is included or excluded, but duplicates are skipped to avoid duplicate subsets.
    - For each subset, the algorithm performs O(n) work to copy the current list to result.
    - With duplicates, the number of unique subsets is less than 2^n (e.g., for nums = [1,1], subsets are [], [1]). In the worst case (all distinct elements), it’s still 2^n.
  + The total work is O(2^n) for generating and copying subsets, dominating the O(n log n) sorting step.
  + Thus, the total time complexity is **O(2^n)**, which is optimal for generating all unique subsets.

**Space Complexity**

* **Space Complexity: O(2^n \* n)**
  + **Output Space**: The result list stores all unique subsets, up to 2^n in the worst case (all distinct elements), each with up to n elements. This requires O(2^n \* n) space for the output, not counted as auxiliary space in some analyses.
  + **Auxiliary Space**:
    - The recursion stack can go up to depth n (one call per element in a subset), requiring O(n) space.
    - The current list stores up to n elements, requiring O(n) space.
    - The sorting operation (Arrays.sort) uses O(log n) space for the recursion stack in Timsort, but this is typically not counted as auxiliary space.
    - The input nums array is modified in-place during sorting and not counted as extra space.
  + No additional data structures scale beyond these.
  + Therefore, the auxiliary space complexity is **O(n)** for the recursion stack and current list, and the total space including output is **O(2^n \* n)**.

**Concise Summary of the Approach**

The solution generates all unique subsets of an integer array that may contain duplicates using backtracking. It first sorts the array to group duplicates together, then uses backtracking to build subsets, skipping duplicate elements at the same level to ensure unique subsets. Each subset is added to the result before exploring further elements. The approach achieves **O(2^n)** time complexity to generate all unique subsets and **O(n)** auxiliary space, efficiently handling duplicates to produce the power set without repetitions.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(2^n) time is optimal for generating all unique subsets, and O(n) auxiliary space is minimal.
  + **Clarity**: The backtracking approach with sorting to handle duplicates is a standard, clear solution, making it interview-friendly for demonstrating recursion and duplicate handling.
  + **Correctness**: Correctly generates all unique subsets by skipping duplicates, ensuring no repetitions in the result.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why sort the array?
    - **Response**: Sorting groups duplicate elements together, allowing the algorithm to skip duplicates (nums[i] == nums[i-1]) at the same recursion level, preventing duplicate subsets (e.g., for [1,1], ensures only one [1] is generated).
  + **Interviewer might ask**: Why check i > start in the duplicate condition?
    - **Response**: The condition i > start ensures duplicates are skipped only within the same recursion level, not across levels. This allows including the same value in different positions of a subset (e.g., [1,1] for nums = [1,1]), but avoids duplicate subsets like [1\_a] and [1\_b].
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative bitmask approach can generate subsets, filtering duplicates by tracking seen subsets (e.g., using a set). Time: O(2^n), Space: O(2^n \* n). The recursive approach is clearer and avoids explicit duplicate tracking.
* **Edge Cases Handled**:
  + Empty array: Returns [[]] (empty subset).
  + Single element: Returns [[], [1]] for nums = [1].
  + All duplicates: Correctly handles (e.g., nums = [1,1] → [[], [1], [1,1]]).
  + Mixed duplicates: Correctly handles (e.g., nums = [1,2,2] → [[], [1], [1,2], [1,2,2], [2], [2,2]]).
  + **Note**: Add validation for null input for robustness.
* **Assumptions**:
  + nums is non-null and may contain duplicates.
  + Result must include all unique subsets in any order.
  + If these assumptions don’t hold (e.g., null array), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative Bitmask**: Use a loop from 0 to 2^n - 1, where each bit represents including an element, and use a set to filter duplicate subsets. Time: O(2^n), Space: O(2^n \* n). Less intuitive and requires extra space for duplicate checking.
  + **Cascading**: Start with an empty subset, duplicate subsets for each element, and skip duplicates by checking sorted elements. Time: O(2^n), Space: O(2^n \* n). Iterative but memory-intensive.
  + **DFS without Sorting**: Track used elements and handle duplicates explicitly (e.g., with a set). Time: O(2^n), Space: O(2^n \* n). More complex and less efficient.
  + The current approach is preferred for its clarity and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null nums, e.g., “If nums is null, return [[]] or throw an exception.”
  + **Clarity**: Add comments explaining duplicate handling (e.g., “Skip duplicates at same recursion level to avoid duplicate subsets”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, all duplicates, and mixed duplicates to show thoroughness.
  + **Efficiency**: The approach is optimal, but note that sorting is critical for efficient duplicate handling.
* **Note on Backtracking Logic**: The algorithm sorts the array to group duplicates, then uses backtracking to build subsets. At each index, it includes the current element and recurses, skipping duplicates at the same level (i > start && nums[i] == nums[i-1]) to avoid generating duplicate subsets. Each subset is added before exploring further elements, ensuring all unique combinations are captured.
* Combination Sum II

Commented Code

class Solution {

*// Method to find all unique combinations of candidates that sum to target, no reuse*

public List<List<Integer>> combinationSum2(int[] candidates, int target) {

List<List<Integer>> result = new ArrayList<>(); *// List to store all unique combinations*

Arrays.sort(candidates); *// Sort the array to handle duplicates*

backtrack(result, new ArrayList<>(), candidates, target, 0); *// Start backtracking*

return result; *// Return all valid combinations*

}

*// Helper method for backtracking to generate combinations*

private void backtrack(List<List<Integer>> result, List<Integer> current, int[] candidates, int target, int start) {

*// Base case: if target is reached, add current combination to result*

if (target == 0) {

result.add(new ArrayList<>(current));

return;

}

*// Try each candidate starting from index 'start'*

for (int i = start; i < candidates.length; i++) {

*// Skip duplicates to avoid duplicate combinations*

if (i > start && candidates[i] == candidates[i - 1]) {

continue;

}

*// Early termination if the remaining sum becomes negative*

if (target - candidates[i] < 0) {

break;

}

current.add(candidates[i]); *// Include candidate*

backtrack(result, current, candidates, target - candidates[i], i + 1); *// Recurse with next index*

current.remove(current.size() - 1); *// Backtrack by removing candidate*

}

}

}

**Time Complexity**

* **Time Complexity: O(2^n)**
  + Let n be the length of candidates.
  + **Sorting**: Sorting the candidates array takes O(n log n).
  + **Backtracking**: The algorithm generates all possible unique subsets that sum to target, without reusing elements.
    - In the worst case (no duplicates, all distinct elements), it explores up to 2^n subsets, as each element can be included or excluded.
    - With duplicates, sorting allows skipping duplicate elements at the same recursion level, reducing the number of unique subsets.
    - For each subset, copying the current list to result takes O(k), where k is the subset size (up to n).
  + The total number of unique subsets is bounded by 2^n (all subsets in the worst case), and the work per subset is O(n) for copying.
  + Early termination (target - candidates[i] < 0) and duplicate skipping reduce the actual number of explored paths, but the worst-case complexity remains **O(2^n)** for generating all valid combinations.
  + The sorting cost (O(n log n)) is dominated by the exponential backtracking, so the total time complexity is **O(2^n)**.
  + This is optimal, as all possible unique combinations must be considered.

**Space Complexity**

* **Space Complexity: O(2^n \* n)**
  + **Output Space**: The result list stores all valid combinations. In the worst case (all distinct elements), the number of subsets summing to target is bounded by 2^n, each with up to n elements, requiring O(2^n \* n) space. This is not counted as auxiliary space in some analyses.
  + **Auxiliary Space**:
    - The recursion stack can go up to depth n (one call per element in a combination), requiring O(n) space.
    - The current list stores up to n elements, requiring O(n) space.
    - The sorting operation (Arrays.sort) uses O(log n) space for the recursion stack in Timsort, typically not counted as auxiliary space.
    - The input candidates array is modified in-place during sorting and not counted as extra space.
  + No additional data structures scale beyond these.
  + Therefore, the auxiliary space complexity is **O(n)** for the recursion stack and current list, and the total space including output is **O(2^n \* n)**.

**Concise Summary of the Approach**

The solution finds all unique combinations of numbers from candidates that sum to target, with each number used at most once. It sorts the array to handle duplicates and enable early termination, then uses backtracking to build combinations, skipping duplicate elements at the same recursion level and pruning branches where the target becomes negative. When the target is reached, the combination is added to the result. The approach achieves **O(2^n)** time complexity to explore all unique combinations and **O(n)** auxiliary space, efficiently handling duplicates and non-reusable elements.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(2^n) time is optimal for exploring all unique combinations, and O(n) auxiliary space is minimal.
  + **Clarity**: The backtracking approach with sorting for duplicate handling and early pruning is standard and interview-friendly, demonstrating recursion and optimization techniques.
  + **Correctness**: Correctly generates unique combinations by skipping duplicates and respecting the no-reuse constraint.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why sort the array?
    - **Response**: Sorting groups duplicate elements together, allowing the algorithm to skip duplicates (candidates[i] == candidates[i-1]) at the same recursion level to avoid duplicate combinations. It also enables early termination when target - candidates[i] < 0, as sorted values ensure later candidates are larger.
  + **Interviewer might ask**: Why use i + 1 in the recursive call?
    - **Response**: Using i + 1 ensures each candidate is used at most once by moving to the next index, preventing reuse of the same element in a combination.
  + **Interviewer might ask**: Why check i > start for duplicates?
    - **Response**: The condition i > start skips duplicates only within the same recursion level, allowing the same value to appear in different levels (e.g., [1\_a, 1\_b] for candidates = [1,1]), but prevents duplicate combinations like [1\_a] and [1\_b].
* **Edge Cases Handled**:
  + Empty array: Returns [] (handled by loop not executing).
  + Target = 0: Returns [[]] if no candidates are added (handled by base case).
  + No valid combinations: Returns [] (e.g., candidates = [2], target = 1).
  + All duplicates: Correctly handles (e.g., candidates = [1,1], target = 2 → [[1,1]]).
  + Mixed duplicates: Correctly handles (e.g., candidates = [1,2,2], target = 4 → [[1,2,2]]).
  + **Note**: Add validation for null input or invalid target for robustness.
* **Assumptions**:
  + candidates is non-null, contains positive integers, and may contain duplicates.
  + target is a positive integer.
  + Each candidate can be used at most once.
  + If these assumptions don’t hold (e.g., null array, negative target), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Dynamic Programming**: Use a DP table to build combinations for each sum up to target, handling duplicates with a set or sorting. Time: O(n \* target), Space: O(target + output). Less intuitive for this problem.
  + **Iterative BFS**: Use a queue to build combinations, tracking remaining target and start index, skipping duplicates. Time: O(2^n), Space: O(2^n \* n). More complex and memory-intensive.
  + **Backtracking without Sorting**: Track used elements and duplicates explicitly (e.g., with a set). Time: O(2^n), Space: O(n). Less efficient due to duplicate handling.
  + The current approach is preferred for its clarity and pruning optimizations.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null candidates or invalid target, e.g., “If candidates is null or target ≤ 0, return [] or throw an exception.”
  + **Clarity**: Add comments explaining duplicate handling and pruning (e.g., “Skip duplicates and prune negative targets for efficiency”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, zero target, duplicates, and no valid combinations to show thoroughness.
  + **Efficiency**: The approach is near-optimal, with sorting and early termination providing practical improvements.
* **Note on Backtracking Logic**: The algorithm sorts candidates to group duplicates and enable pruning. It builds combinations by selecting candidates from start onward, ensuring no reuse (i + 1). Duplicates are skipped at the same level to avoid duplicate combinations, and negative targets prune invalid branches. When target = 0, the combination is added, ensuring all unique solutions are captured.
* Word Search

Commented Code

public class Solution {

*// Method to check if a word exists in the board*

public boolean exist(char[][] board, String word) {

int m = board.length; *// Number of rows*

int n = board[0].length; *// Number of columns*

boolean[][] visited = new boolean[m][n]; *// Track visited cells*

boolean result = false;

*// Check each cell as a potential starting point*

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

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

if (board[i][j] == word.charAt(0)) { *// Start if first character matches*

result = backtrack(board, word, visited, i, j, 0);

if (result) return true; *// Return true if word is found*

}

}

}

return false; *// Word not found*

}

*// Helper method for backtracking to find the word*

private boolean backtrack(char[][] board, String word, boolean[][] visited, int i, int j, int index) {

*// Base case: if all characters are matched, word is found*

if (index == word.length()) {

return true;

}

*// Check if current position is invalid or doesn't match*

if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || visited[i][j] || board[i][j] != word.charAt(index)) {

return false;

}

visited[i][j] = true; *// Mark cell as visited*

*// Explore four directions: down, up, right, left*

if (backtrack(board, word, visited, i + 1, j, index + 1) ||

backtrack(board, word, visited, i - 1, j, index + 1) ||

backtrack(board, word, visited, i, j + 1, index + 1) ||

backtrack(board, word, visited, i, j - 1, index + 1)) {

return true; *// Word found in one of the directions*

}

visited[i][j] = false; *// Backtrack: unmark cell*

return false; *// Word not found in any direction*

}

}

**Time Complexity**

* **Time Complexity: O(M \* N \* 3^L)**
  + Let M be the number of rows and N be the number of columns in board, and L be the length of word.
  + **Starting Points**: The algorithm checks each of the M \* N cells as a potential starting point for the word.
  + **Backtracking**:
    - For each starting cell, the algorithm performs a depth-first search (DFS) to match the word of length L.
    - At each step, it explores up to 4 directions (down, up, right, left), but after the first step, it avoids backtracking to the previous cell, effectively exploring up to 3 directions per step.
    - The maximum depth of the recursion is L, leading to a branching factor of approximately 3 per step after the first, resulting in up to 3^L paths in the worst case (e.g., board filled with the same letter, allowing many valid paths).
    - Each recursive call performs O(1) operations (checking bounds, visited status, character match, marking/unmarking).
  + Thus, for each starting cell, the backtracking takes O(3^L), and across M \* N cells, the total time is **O(M \* N \* 3^L)**.
  + This is near-optimal, as exploring all possible paths for a word of length L is necessary.

**Space Complexity**

* **Space Complexity: O(M \* N + L)**
  + **Auxiliary Space**:
    - The visited array is M x N, requiring O(M \* N) space to track visited cells.
    - The recursion stack can go up to depth L (one call per character in the word), requiring O(L) space.
    - Other variables (i, j, index, etc.) use O(1) space.
    - The input board and word are not counted as extra space.
  + No additional data structures scale beyond these.
  + Therefore, the auxiliary space complexity is **O(M \* N + L)** for the visited array and recursion stack. Since there is no output storage beyond the boolean result, the total space complexity is **O(M \* N + L)**.

**Concise Summary of the Approach**

The solution determines if a word can be formed by a path of adjacent letters (horizontally or vertically) on an M x N board using backtracking. It iterates over each cell, starting a DFS if the cell matches the word’s first character. The backtracking explores four directions, marking cells as visited to avoid reuse, and backtracks by unmarking them. If all characters are matched, it returns true. The approach achieves **O(M \* N \* 3^L)** time complexity to explore all paths and **O(M \* N + L)** space complexity for the visited array and recursion stack, efficiently finding the word.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(M \* N \* 3^L) is near-optimal, as it explores all valid paths starting from each cell, pruning invalid ones early.
  + **Clarity**: The backtracking approach with a visited array is standard and intuitive, making it interview-friendly for demonstrating DFS and state management.
  + **Correctness**: Correctly checks all possible paths, ensuring the word is found if it exists while respecting the no-reuse constraint.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a visited array?
    - **Response**: The visited array prevents reusing cells in the same path, as required by the problem (each cell can be used at most once). It ensures valid paths without cycles.
  + **Interviewer might ask**: Why explore four directions?
    - **Response**: The problem allows moving up, down, left, or right to form the word. Exploring all four directions ensures all possible paths are checked, with visited preventing backtracking to the same cell.
  + **Interviewer might ask**: Can you optimize it further?
    - **Response**: The approach is near-optimal, but minor optimizations include checking the board’s character set against the word’s to prune early (e.g., if word contains letters not in board). This doesn’t change the worst-case complexity but helps in practice.
* **Edge Cases Handled**:
  + Empty board or word: Returns false (handled by loop conditions and base case).
  + Single cell: Checks for single-character word (e.g., board = [['a']], word = "a" → true).
  + Word not found: Returns false (e.g., board = [['a']], word = "b" → false).
  + Long word: Scales within constraints (e.g., M, N ≤ 6, word.length ≤ 15).
  + Board with repeated letters: Correctly handles (e.g., board filled with 'a', word = "aaa").
  + **Note**: Add validation for null inputs for robustness.
* **Assumptions**:
  + board is non-null, M x N, with uppercase/lowercase letters.
  + word is non-null and contains uppercase/lowercase letters.
  + Each cell can be used at most once in a path.
  + If these assumptions don’t hold (e.g., null inputs), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **DFS without Visited Array**: Modify the board in-place (e.g., mark with '#') to track visited cells, restoring after backtracking. Time: O(M \* N \* 3^L), Space: O(L). Saves space but modifies input.
  + **BFS**: Use a queue to explore paths level by level, tracking visited cells. Time: O(M \* N \* 3^L), Space: O(M \* N + L). More complex and unnecessary.
  + **Precheck Characters**: Validate if all word characters exist in the board before DFS. Time: O(M \* N \* 3^L), Space: O(M \* N). Minor optimization for early pruning.
  + The current approach is preferred for its clarity and balance of space usage.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null board or word, e.g., “If board or word is null, return false or throw an exception.”
  + **Clarity**: Add comments explaining backtracking flow (e.g., “Explore four directions, marking cells to prevent reuse”).
  + **Edge Case Discussion**: In an interview, mention handling of empty inputs, single cells, and repeated letters to show thoroughness.
  + **Efficiency**: The approach is near-optimal, but prechecking character availability could be mentioned as a practical optimization.
* **Note on Backtracking Logic**: The algorithm starts from each cell matching the word’s first character, using DFS to match subsequent characters in four directions. The visited array prevents reusing cells, and backtracking unmarks cells to explore other paths. If the word’s length is reached, the word is found. Pruning occurs for invalid positions, visited cells, or mismatched characters, ensuring efficiency.
* Palindrome Partitioning

Commented Code

class Solution {

*// Method to find all possible palindrome partitions of the input string*

public List<List<String>> partition(String s) {

List<List<String>> result = new ArrayList<>(); *// List to store all partitions*

backtrack(result, new ArrayList<>(), s, 0); *// Start backtracking*

return result; *// Return all valid partitions*

}

*// Helper method for backtracking to generate partitions*

private void backtrack(List<List<String>> result, List<String> current, String s, int start) {

*// Base case: if entire string is processed, add current partition to result*

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

result.add(new ArrayList<>(current));

return;

}

*// Try all possible substrings starting from 'start'*

for (int end = start; end < s.length(); end++) {

if (isPalindrome(s, start, end)) { *// If substring is a palindrome*

current.add(s.substring(start, end + 1)); *// Include palindrome*

backtrack(result, current, s, end + 1); *// Recurse with next position*

current.remove(current.size() - 1); *// Backtrack by removing substring*

}

}

}

*// Helper method to check if substring from start to end is a palindrome*

private boolean isPalindrome(String s, int start, int end) {

while (start < end) {

if (s.charAt(start++) != s.charAt(end--)) {

return false; *// Not a palindrome if characters don't match*

}

}

return true; *// Is a palindrome*

}

}

**Time Complexity**

* **Time Complexity: O(N \* 2^N)**
  + Let N be the length of the input string s.
  + **Backtracking**:
    - The algorithm generates all possible partitions by considering each substring starting from index start as a potential palindrome.
    - For each starting position, it tries all possible ending positions (end = start to N-1), leading to a binary decision tree where each position can be the start of a new substring or continue an existing one.
    - In the worst case (e.g., string of all identical characters like "aaa"), there are 2^(N-1) possible partitions, as each position can either be a cut or not (except the last position).
    - For each partition, copying the current list to result takes O(N) time, as a partition can have up to N substrings (e.g., all single characters).
  + **Palindrome Check**:
    - The isPalindrome method checks if a substring from start to end is a palindrome, taking O(N) time in the worst case (e.g., checking the entire string).
    - For each starting position start, there are up to N - start possible end positions, and palindrome checks are performed for each.
    - The total cost of palindrome checks across all backtracking steps is bounded by the number of substring checks, which is amortized within the backtracking tree.
  + The total number of partitions is O(2^N), and each involves O(N) work for copying and palindrome checks, leading to **O(N \* 2^N)** time complexity.
  + This is near-optimal, as generating all possible partitions is inherently exponential.

**Space Complexity**

* **Space Complexity: O(N \* 2^N)**
  + **Output Space**: The result list stores all valid partitions. In the worst case (e.g., s = "aaa"), there are 2^(N-1) partitions, each containing up to N characters (or O(N) for substring storage). This requires O(N \* 2^N) space for the output, not counted as auxiliary space in some analyses.
  + **Auxiliary Space**:
    - The recursion stack can go up to depth N (one call per character in a partition of single characters), requiring O(N) space.
    - The current list stores up to N substrings (in the worst case, each character is a separate substring), but the total character count is bounded by O(N), as substrings reference the input string.
    - The s.substring(start, end + 1) operation creates a new string, but this is temporary and bounded by O(N) per recursive call.
    - The input string s is not counted as extra space.
  + No additional data structures scale beyond these.
  + Therefore, the auxiliary space complexity is **O(N)** for the recursion stack and current list, and the total space including output is **O(N \* 2^N)**.

**Concise Summary of the Approach**

The solution finds all possible ways to partition a string into palindromic substrings using backtracking. It tries all substrings starting from the current index, checks if they are palindromes, and recursively builds partitions with the remaining string. Valid partitions are added to the result when the entire string is processed. The approach achieves **O(N \* 2^N)** time complexity to generate all partitions and **O(N)** auxiliary space, efficiently handling palindrome checks and producing all valid partitions.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(N \* 2^N) is near-optimal for generating all possible partitions, given the exponential number of partitions in the worst case.
  + **Clarity**: The backtracking approach with a separate palindrome check is intuitive and modular, making it interview-friendly for demonstrating recursion and string manipulation.
  + **Correctness**: Correctly generates all valid palindrome partitions without duplicates.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why check all possible substrings in backtracking?
    - **Response**: To find all partitions, we must consider every possible substring starting from the current position and check if it’s a palindrome. This ensures no valid partition is missed, as each substring could be part of a solution.
  + **Interviewer might ask**: Why use a separate isPalindrome method?
    - **Response**: Separating the palindrome check improves code readability and modularity. It also allows potential optimizations (e.g., memoizing palindrome results) without changing the backtracking logic.
  + **Interviewer might ask**: Can you optimize the palindrome checks?
    - **Response**: Yes, memoizing palindrome results for substring ranges (start, end) using a 2D boolean array can reduce repeated checks, lowering the palindrome check cost from O(N) to O(1) per substring after preprocessing. This changes the complexity to O(2^N + N^2) with O(N^2) preprocessing.
* **Edge Cases Handled**:
  + Empty string: Returns [[]] (handled by base case when start == s.length()).
  + Single character: Returns [[s]] (e.g., s = "a" → [["a"]]).
  + All identical characters: Handles correctly (e.g., s = "aaa" → [["a","a","a"], ["a","aa"], ["aa","a"], ["aaa"]]).
  + No palindromes: Handles correctly (e.g., s = "ab" → [["a","b"]]).
  + **Note**: Add validation for null input for robustness.
* **Assumptions**:
  + s is non-null and contains lowercase letters (or as specified by constraints).
  + Result must include all possible palindrome partitions in any order.
  + If these assumptions don’t hold (e.g., null string), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Dynamic Programming with Backtracking**: Precompute a 2D boolean array for palindrome substrings in O(N^2) time, then use backtracking to generate partitions in O(2^N) time. Total: O(N^2 + 2^N), Space: O(N^2 + N). Reduces palindrome check overhead but increases space.
  + **Iterative Approach**: Build partitions iteratively using a queue or stack, tracking current partitions. Time: O(N \* 2^N), Space: O(N \* 2^N). More complex and memory-intensive.
  + **Divide and Conquer**: Split the string and recursively solve for subproblems. Time: O(N \* 2^N), Space: O(N). Similar to backtracking but less intuitive.
  + The current approach is preferred for its clarity and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty s, e.g., “If s is null, return []; if empty, return [[]].”
  + **Clarity**: Add comments explaining palindrome partitioning (e.g., “Try all palindromic substrings to build partitions”).
  + **Edge Case Discussion**: In an interview, mention handling of empty strings, single characters, and all identical characters to show thoroughness.
  + **Efficiency**: The approach is near-optimal, but memoizing palindrome checks could reduce the cost of repeated substring checks.
* **Note on Backtracking Logic**: The algorithm tries all substrings starting from start, checks if they are palindromes using isPalindrome, and recursively builds partitions with the remaining string. When start reaches the string’s length, a valid partition is added. Backtracking removes the last substring to explore other possibilities, ensuring all valid partitions are captured.
* Letter Combinations of a Phone Number

Commented Code

public class Solution {

*// List to store all valid letter combinations*

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

*// Mapping of digits to corresponding letters*

private String[] digitToChar = {

"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"

};

*// Method to generate all letter combinations for a given digit string*

public List<String> letterCombinations(String digits) {

if (digits.isEmpty()) return res; *// Return empty list for empty input*

backtrack(0, "", digits); *// Start backtracking*

return res; *// Return all combinations*

}

*// Helper method for backtracking to generate combinations*

private void backtrack(int i, String curStr, String digits) {

*// Base case: if current string length equals digits length, add to result*

if (curStr.length() == digits.length()) {

res.add(curStr);

return;

}

*// Get letters for current digit*

String chars = digitToChar[digits.charAt(i) - '0'];

*// Try each letter for the current digit*

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

backtrack(i + 1, curStr + c, digits); *// Recurse with next digit and updated string*

}

}

}

**Time Complexity**

* **Time Complexity: O(4^n)**
  + Let n be the length of the input digits string.
  + Each digit maps to a set of letters (3 for digits 2-6, 8; 4 for digits 7, 9; 0 for 0, 1). In the worst case, assume each digit maps to 4 letters (e.g., digits 7 or 9).
  + The backtracking process forms a tree where each of the n digits branches to up to 4 choices, leading to 4 \* 4 \* ... \* 4 = 4^n possible combinations.
  + For each combination:
    - Building the string (curStr + c) takes O(1) for concatenation in the recursive call.
    - Adding to res takes O(n) to copy the string of length n.
  + The total number of combinations is 4^n (in the worst case), and copying each combination takes O(n).
  + Thus, the total time complexity is **O(n \* 4^n)**, often simplified to **O(4^n)** as the exponential term dominates.
  + This is optimal, as all possible combinations must be generated.

**Space Complexity**

* **Space Complexity: O(n \* 4^n)**
  + **Output Space**: The res list stores all combinations, up to 4^n in the worst case, each of length n. This requires O(n \* 4^n) space for the output, not counted as auxiliary space in some analyses.
  + **Auxiliary Space**:
    - The recursion stack can go up to depth n (one call per digit), requiring O(n) space.
    - The curStr string is built incrementally, with up to n characters, requiring O(n) space per recursive call.
    - The digitToChar array is fixed-size (10 elements), using O(1) space.
    - The input digits string is not counted as extra space.
  + No additional data structures scale beyond these.
  + Therefore, the auxiliary space complexity is **O(n)** for the recursion stack and string building, and the total space including output is **O(n \* 4^n)**.

**Concise Summary of the Approach**

The solution generates all possible letter combinations for a given string of digits, where each digit maps to a set of letters (e.g., 2 → "abc"). It uses backtracking to explore all combinations, maintaining a current string and processing each digit’s letters recursively. When the string length matches the input length, it’s added to the result. The approach achieves **O(4^n)** time complexity to generate all combinations and **O(n)** auxiliary space, efficiently producing all valid letter combinations.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(4^n) time is optimal, as all possible combinations (up to 4^n for digits like 7, 9) must be generated, and O(n) auxiliary space is minimal.
  + **Clarity**: The backtracking approach is intuitive, mapping each digit to its letters and building combinations systematically, making it interview-friendly for demonstrating recursion.
  + **Correctness**: Correctly handles all valid digits (2-9) and produces all combinations without duplicates.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why check curStr.length() == digits.length()?
    - **Response**: This ensures the current combination is complete, covering all digits in the input. Only then is it a valid combination to add to the result.
  + **Interviewer might ask**: Why use digitToChar array?
    - **Response**: The digitToChar array provides a constant-time mapping from digits (0-9) to their corresponding letters, simplifying the lookup and ensuring clarity.
  + **Interviewer might ask**: Can you solve it iteratively?
    - **Response**: Yes, an iterative approach can use a queue or list to build combinations level by level, adding one letter per digit. Time: O(4^n), Space: O(n \* 4^n). The recursive approach is often clearer and more concise for interviews.
* **Edge Cases Handled**:
  + Empty string: Returns empty list (digits = "" → []).
  + Single digit: Returns all letters for that digit (e.g., digits = "2" → ["a", "b", "c"]).
  + Digits 7, 9: Handles 4-letter mappings correctly (e.g., 7 → "pqrs").
  + Invalid digits: Not applicable, as problem assumes digits 2-9.
  + **Note**: Add validation for invalid inputs (e.g., digits 0, 1) for robustness.
* **Assumptions**:
  + digits is non-null and contains characters '2' to '9'.
  + Result must include all possible combinations in any order.
  + If these assumptions don’t hold (e.g., null input or invalid digits), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Iterative BFS**: Use a queue to build combinations, starting with an empty string and adding one letter per digit. Time: O(4^n), Space: O(n \* 4^n). More memory-intensive but avoids recursion.
  + **Iterative Product**: Generate combinations by iterating over all possible letter choices for each digit (like a Cartesian product). Time: O(4^n), Space: O(n \* 4^n). Less intuitive.
  + **DFS with StringBuilder**: Use a StringBuilder for curStr to reduce string concatenation overhead. Time: O(4^n), Space: O(n \* 4^n). Minor optimization but similar logic.
  + The current approach is preferred for its clarity and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or invalid digits (e.g., '0', '1'), e.g., “If digits is null or contains invalid characters, return [] or throw an exception.”
  + **Clarity**: Add comments explaining backtracking flow (e.g., “Explore all letters for each digit to build combinations”).
  + **Edge Case Discussion**: In an interview, mention handling of empty strings, single digits, and 4-letter digits to show thoroughness.
  + **Efficiency**: The approach is optimal, but using a StringBuilder for curStr could reduce string concatenation overhead (minor in practice).
* **Note on Backtracking Logic**: The algorithm processes each digit sequentially, using the digitToChar array to get the corresponding letters. For each letter, it appends it to the current string and recurses to the next digit, building combinations. When the string length matches the input length, it’s added to the result. Backtracking implicitly explores all 4^n combinations by trying each letter at each position.
* N Queens

Commented Code

class Solution {

*// Method to find all valid N-Queens configurations*

public List<List<String>> solveNQueens(int n) {

char[][] board = new char[n][n]; *// Initialize n x n board*

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

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

board[i][j] = '.'; *// Set all cells to empty*

}

}

List<List<String>> result = new ArrayList<>(); *// List to store all solutions*

backtrack(board, 0, result); *// Start backtracking from first column*

return result; *// Return all valid configurations*

}

*// Helper method for backtracking to place queens*

private void backtrack(char[][] board, int col, List<List<String>> result) {

*// Base case: if all columns are filled, add board to result*

if (col == board.length) {

result.add(construct(board));

return;

}

*// Try placing a queen in each row of the current column*

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

if (isValid(board, i, col)) { *// Check if placement is valid*

board[i][col] = 'Q'; *// Place queen*

backtrack(board, col + 1, result); *// Recurse to next column*

board[i][col] = '.'; *// Backtrack by removing queen*

}

}

}

*// Helper method to check if a queen can be placed at (row, col)*

private boolean isValid(char[][] board, int row, int col) {

*// Check row for any queens in previous columns*

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

if (board[row][i] == 'Q') {

return false;

}

}

*// Check upper-left diagonal*

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board[i][j] == 'Q') {

return false;

}

}

*// Check lower-left diagonal*

for (int i = row, j = col; i < board.length && j >= 0; i++, j--) {

if (board[i][j] == 'Q') {

return false;

}

}

return true; *// Placement is valid*

}

*// Helper method to convert board to list of strings*

private List<String> construct(char[][] board) {

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

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

String row = new String(board[i]); *// Convert char array to string*

result.add(row);

}

return result; *// Return board as list of strings*

}

}

**Time Complexity**

* **Time Complexity: O(N!)**
  + Let N be the size of the board (n x n).
  + **Initialization**: Initializing the n x n board takes O(N^2).
  + **Backtracking**:
    - The algorithm places one queen per column, ensuring no two queens are in the same row or threaten each other.
    - For the first column, there are N possible rows to place a queen.
    - For the second column, there are up to N-1 valid rows (due to row and diagonal constraints), and so on, leading to a maximum of N \* (N-1) \* ... \* 1 = N! possible configurations in the worst case.
    - For each placement, the isValid check takes O(N) time:
      * Row check: O(N) to scan previous columns.
      * Upper-left diagonal check: O(N) in the worst case (diagonal to board edge).
      * Lower-left diagonal check: O(N) in the worst case.
    - For each valid configuration (when col == N), the construct method converts the board to a list of strings, taking O(N^2) to copy N rows of length N.
  + The total number of valid configurations is much less than N! due to constraints, but the worst-case exploration is O(N!) for the decision tree.
  + The work per node (O(N) for isValid and O(N^2) for construct) is dominated by the number of nodes explored, so the overall time complexity is **O(N!)**.
  + This is near-optimal, as exploring all valid queen placements is inherently factorial.

**Space Complexity**

* **Space Complexity: O(N^2 + N \* T)**
  + **Board Storage**: The board is an N x N char array, requiring O(N^2) space.
  + **Output Space**: The result list stores all valid configurations, where T is the number of valid solutions (T is much less than N!, e.g., T = 92 for N = 8). Each configuration has N strings of length N, requiring O(N^2 \* T) space. This is not counted as auxiliary space in some analyses.
  + **Auxiliary Space**:
    - The recursion stack can go up to depth N (one call per column), requiring O(N) space.
    - The construct method creates a list of N strings, each of length N, requiring O(N^2) space temporarily per configuration.
    - Other variables (i, j, col, etc.) use O(1) space.
    - The input n is not counted as extra space.
  + No additional data structures scale beyond these.
  + Therefore, the auxiliary space complexity is **O(N^2)** for the board and recursion stack, and the total space including output is **O(N^2 + N \* T)**.

**Concise Summary of the Approach**

The solution finds all valid N-Queens configurations on an n x n board using backtracking. It initializes an empty board and places one queen per column, ensuring no two queens share a row or threaten each other (via row and diagonal checks). Valid configurations are converted to string lists and added to the result. The approach achieves **O(N!)** time complexity to explore all possible placements and **O(N^2)** auxiliary space for the board and recursion, efficiently solving the N-Queens problem.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(N!) is near-optimal for exploring all valid queen placements, and O(N^2) auxiliary space is minimal.
  + **Clarity**: The backtracking approach with a clear validity check is standard and intuitive, making it interview-friendly for demonstrating recursion and board-based problem-solving.
  + **Correctness**: Correctly generates all valid configurations, respecting row, column, and diagonal constraints.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why check only previous columns in isValid?
    - **Response**: Since queens are placed column by column from left to right, only previous columns (i < col) can contain queens. Checking rows and diagonals in those columns ensures no conflicts with the current placement.
  + **Interviewer might ask**: Why not check columns explicitly?
    - **Response**: Placing one queen per column inherently ensures no two queens share a column, so explicit column checks are unnecessary.
  + **Interviewer might ask**: Can you optimize the isValid check?
    - **Response**: Yes, using three arrays to track occupied rows, upper diagonals, and lower diagonals can reduce the isValid check to O(1) per placement, maintaining O(N!) overall time but improving the constant factor.
* **Edge Cases Handled**:
  + n = 1: Returns [["Q"]] (single queen placement).
  + n = 2 or n = 3: Returns [] (no valid solutions, as queens cannot be placed without conflicts).
  + n = 4: Returns valid configurations (e.g., two solutions for 4x4 board).
  + Large n: Scales within constraints (e.g., n ≤ 9).
  + **Note**: Add validation for invalid n (e.g., n ≤ 0) for robustness.
* **Assumptions**:
  + n is a positive integer (typically 1 to 9 per problem constraints).
  + The board uses '.' for empty cells and 'Q' for queens in the output.
  + If these assumptions don’t hold (e.g., invalid n), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Optimized Validity Check**: Use arrays to track occupied rows, upper diagonals (row + col), and lower diagonals (row - col). Time: O(N!), Space: O(N^2). Reduces isValid to O(1) but adds minor complexity.
  + **Bit Manipulation**: Use bitsets to track occupied positions, reducing space for validity checks. Time: O(N!), Space: O(N^2). More complex and less intuitive.
  + **Iterative Approach**: Simulate backtracking iteratively with a stack. Time: O(N!), Space: O(N^2). Less clear and unnecessary.
  + The current approach is preferred for its clarity and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for invalid n, e.g., “If n ≤ 0, return [] or throw an exception.”
  + **Clarity**: Add comments explaining constraints (e.g., “Check row and diagonals for conflicts in previous columns”).
  + **Edge Case Discussion**: In an interview, mention handling of n = 1, n = 2, and large n to show thoroughness.
  + **Efficiency**: The approach is near-optimal, but using arrays for row/diagonal tracking could be mentioned as an optimization.
* **Note on Backtracking Logic**: The algorithm places one queen per column, trying each row and checking validity (no queens in the same row or diagonals). If valid, it places the queen, recurses to the next column, and backtracks by removing the queen. When all columns are filled, the board is converted to a string list and added to the result, ensuring all valid configurations are captured.

11. Graphs (13 problems)

* Number of Islands
* Clone Graph
* Max Area of Island
* Pacific Atlantic Water Flow
* Surrounded Regions
* Rotting Oranges
* Walls And Gates
* Course Schedule
* Course Schedule II
* Redundant Connection
* Number of Connected Components In An Undirected Graph
* Graph Valid Tree
* Word Ladder

12. Advanced Graphs (6 problems)

* Reconstruct Itinerary
* Min Cost to Connect All Points
* Network Delay Time
* Swim In Rising Water
* Alien Dictionary
* Cheapest Flights Within K Stops

13. 1-D Dynamic Programming (12 problems)

* Climbing Stairs
* Min Cost Climbing Stairs
* House Robber
* House Robber II
* Longest Palindromic Substring
* Palindromic Substrings
* Decode Ways
* Coin Change
* Maximum Product Subarray
* Word Break
* Longest Increasing Subsequence
* Partition Equal Subset Sum

14. 2-D Dynamic Programming (11 problems)

* Unique Paths
* Longest Common Subsequence
* Best Time to Buy And Sell Stock With Cooldown
* Coin Change 2
* Target Sum
* Interleaving String
* Longest Increasing Path In a Matrix
* Distinct Subsequences
* Edit Distance
* Burst Balloons
* Regular Expression Matching

15. Greedy (8 problems)

** Maximum Subarray**

Commented Code

**Commented Code**

class Solution {

*// Method to find the maximum sum of a contiguous subarray*

public int maxSubArray(int[] nums) {

int currSum = 0; *// Running sum of current subarray*

int maxSub = nums[0]; *// Maximum subarray sum, initialized with first element*

*// Iterate through the array*

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

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

if (currSum < 0) {

currSum = 0;

}

*// Add current element to running sum*

currSum += nums[i];

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

maxSub = Math.max(maxSub, currSum);

}

return maxSub; *// Return the maximum subarray sum*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the array nums exactly once, where n is the length of the array.
  + Each iteration performs O(1) operations: checking currSum, updating currSum, and updating maxSub with Math.max.
  + Therefore, the total time complexity is **O(n)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses only two integer variables (currSum and maxSub), which require O(1) space.
  + The input array nums is not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution finds the maximum sum of a contiguous subarray using Kadane’s algorithm. It maintains a running sum (currSum) and resets it to 0 if it becomes negative, effectively starting a new subarray. After adding each element, it updates the maximum sum (maxSub) if the current sum is larger. The approach achieves **O(n)** time complexity for a single pass through the array and **O(1)** space complexity, efficiently computing the maximum subarray sum.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each element must be considered, and O(1) space minimizes memory usage.
  + **Clarity**: Kadane’s algorithm is intuitive, tracking the maximum sum with minimal variables, making it interview-friendly.
  + **Correctness**: Correctly handles all cases, including negative numbers and single-element arrays, by initializing maxSub with the first element.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why reset currSum to 0 when negative?
    - **Response**: A negative currSum would reduce the sum of any subsequent subarray, so resetting to 0 starts a new subarray at the current element, potentially yielding a larger sum.
  + **Interviewer might ask**: Why initialize maxSub with nums[0]?
    - **Response**: Initializing with nums[0] ensures correctness for arrays with all negative numbers or a single element, as the maximum subarray could be a single element.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: A divide-and-conquer approach splits the array and considers left, right, and cross-boundary subarrays. It takes O(n log n) time and O(log n) space but is more complex. A brute-force approach checking all subarrays takes O(n²) time, which is inefficient.
* **Edge Cases Handled**:
  + Single element: Returns nums[0] (e.g., nums = [5] or nums = [-1]).
  + All negative numbers: Returns the largest element (e.g., nums = [-2, -1, -3] returns -1).
  + All positive numbers: Likely includes all elements (e.g., nums = [1, 2, 3] returns 6).
  + Mixed numbers: Correctly finds maximum subarray (e.g., nums = [-2, 1, -3, 4, -1, 2] returns 6 for [4, -1, 2]).
  + **Note**: Add validation for null or empty arrays for robustness.
* **Assumptions**:
  + nums is non-null and has at least one element.
  + Elements are integers, possibly negative.
  + If these assumptions don’t hold (e.g., empty array), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Divide-and-Conquer**: Recursively split the array, compute maximum subarrays for left, right, and cross-boundary cases. Time: O(n log n), Space: O(log n) for recursion stack. More complex and less efficient.
  + **Brute Force**: Check all possible subarrays by iterating over start and end indices. Time: O(n²), Space: O(1). Inefficient for large arrays.
  + **Dynamic Programming**: Use a DP array to store maximum subarray sums ending at each index. Time: O(n), Space: O(n). Equivalent to Kadane’s but uses more space.
  + Kadane’s algorithm is preferred for its O(n) time, O(1) space, and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty arrays, e.g., “If nums is null or empty, throw an exception or return a default value.”
  + **Clarity**: Add comments explaining Kadane’s logic (e.g., “Reset negative currSum to start new subarray for maximum sum”).
  + **Edge Case Discussion**: In an interview, mention handling of all-negative arrays, single elements, and mixed numbers to show thoroughness.
  + **Efficiency**: The approach is already optimal, but note that maxSub = nums[0] ensures correctness for edge cases like all-negative arrays.
* **Note on Kadane’s Logic**: The algorithm greedily builds subarrays by adding elements to currSum. If currSum becomes negative, it’s reset to 0, as continuing would reduce future sums. maxSub tracks the highest sum seen, updated after each element. This ensures the maximum subarray sum is found in a single pass.

** Jump Game**

Commented Code

class Solution {

*// Method to check if you can reach the last index*

public boolean canJump(int[] nums) {

int goal = nums.length - 1; *// Initialize goal as the last index*

*// Iterate backwards from second-to-last index*

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

*// If current index can reach or pass the goal, update goal*

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

goal = i;

}

}

*// Return true if goal reaches the starting index (0)*

return goal == 0;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the array from index nums.length - 2 to 0, covering n - 1 elements, where n is the length of nums.
  + Each iteration performs O(1) operations: checking i + nums[i] >= goal and updating goal if true.
  + Therefore, the total time complexity is **O(n)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses only one integer variable (goal) for tracking the reachable index, requiring O(1) space.
  + The input array nums is not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution determines if the last index of an array can be reached by jumping from index 0, where each index i allows a jump of up to nums[i] steps. Using a greedy approach, it starts with the last index as the goal and iterates backward, updating the goal to the current index i if it can reach or pass the current goal. The array is reachable if the goal becomes 0. The approach achieves **O(n)** time complexity for a single pass and **O(1)** space complexity, efficiently checking reachability.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each element is considered once, and O(1) space minimizes memory usage.
  + **Clarity**: The backward greedy approach is intuitive, progressively finding the earliest index that can reach the end, making it interview-friendly.
  + **Correctness**: Correctly determines reachability by ensuring a chain of jumps can connect index 0 to the last index.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why iterate backward instead of forward?
    - **Response**: Iterating backward starts with the target (last index) and finds the earliest index that can reach it, reducing the problem to checking if index 0 is reachable. A forward approach works but requires tracking the farthest reachable index, which is equivalent but less intuitive in some cases.
  + **Interviewer might ask**: Why update goal only when i + nums[i] >= goal?
    - **Response**: If i + nums[i] >= goal, index i can reach the current goal, so i becomes the new goal to check if earlier indices can reach it. This ensures the earliest possible index is tracked.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: A forward greedy approach tracks the farthest reachable index, updating it as you iterate. It also takes O(n) time and O(1) space. A dynamic programming approach could mark reachable indices but uses O(n) space and is less efficient.
* **Edge Cases Handled**:
  + Single element: Returns true (e.g., nums = [0]; already at the last index).
  + Two elements: Returns true if nums[0] ≥ 1, else false (e.g., nums = [1,0] vs. nums = [0,0]).
  + Unreachable last index: Returns false (e.g., nums = [2,0,0]; can’t reach index 2).
  + All zeros except last: Returns false unless starting at last index (e.g., nums = [0,0,0]).
  + **Note**: Add validation for null or empty arrays for robustness.
* **Assumptions**:
  + nums is non-null, has at least one element, and contains non-negative integers.
  + 0 ≤ nums[i] ≤ 10^5 (per typical problem constraints).
  + If these assumptions don’t hold (e.g., negative jumps), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Forward Greedy**: Iterate forward, tracking the farthest reachable index. If it reaches or exceeds the last index, return true. Time: O(n), Space: O(1). Equally efficient but tracks forward progress.
  + **Dynamic Programming**: Use a boolean array to mark reachable indices, updating based on jump ranges. Time: O(n²) in worst case, Space: O(n). Inefficient and unnecessary.
  + **BFS/DFS**: Treat indices as a graph and check if the last index is reachable from 0. Time: O(n²) due to multiple jumps, Space: O(n). Overly complex.
  + The backward greedy approach is preferred for its simplicity and efficiency.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty arrays, e.g., “If nums is null or empty, return false or throw an exception.”
  + **Clarity**: Add comments explaining the greedy logic (e.g., “Move goal backward if current index can reach it”).
  + **Edge Case Discussion**: In an interview, mention handling of single-element arrays, unreachable indices, and all-zero cases to show thoroughness.
  + **Efficiency**: The approach is already optimal, but note that the backward iteration simplifies the logic by focusing on the target index.
* **Note on Greedy Logic**: The algorithm greedily moves the goal backward to the earliest index that can reach the current goal. By iterating from the end, it ensures that if index 0 becomes the goal, a sequence of jumps exists from 0 to the last index. If the goal doesn’t reach 0, some index blocks the path, making the last index unreachable.

** Jump Game II**

Commented Code

class Solution {

*// Method to find the minimum number of jumps to reach the last index*

public int jump(int[] nums) {

int jump = 0; *// Number of jumps needed*

int currMax = 0; *// Farthest index reachable with current jumps*

int currEnd = 0; *// End of the range for the current jump*

*// Iterate until the second-to-last index*

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

*// Update farthest reachable index*

currMax = Math.max(currMax, i + nums[i]);

*// If reached the end of current jump range, make a jump*

if (i == currEnd) {

jump++;

currEnd = currMax; *// Update range to farthest reachable*

}

}

return jump; *// Return minimum jumps*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the array nums from index 0 to nums.length - 2, covering n - 1 elements, where n is the length of the array.
  + Each iteration performs O(1) operations: updating currMax with Math.max and checking/updating jump and currEnd when i == currEnd.
  + Therefore, the total time complexity is **O(n)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses three integer variables (jump, currMax, currEnd), which require O(1) space.
  + The input array nums is not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution finds the minimum number of jumps to reach the last index of an array, where each index i allows a jump of up to nums[i] steps. Using a greedy approach, it tracks the farthest reachable index (currMax) and the current jump’s range (currEnd). Each time the current index reaches currEnd, it increments the jump count and updates currEnd to currMax. The approach achieves **O(n)** time complexity for a single pass and **O(1)** space complexity, efficiently computing the minimum jumps.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each element is considered once, and O(1) space minimizes memory usage.
  + **Clarity**: The greedy approach is intuitive, tracking the farthest reachable index to minimize jumps, making it interview-friendly.
  + **Correctness**: Correctly finds the minimum jumps by always choosing the jump that maximizes reach within the current range.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why update currMax at every index?
    - **Response**: currMax tracks the farthest index reachable with the current number of jumps. Updating it at each index ensures we consider all possible jumps to maximize the range before committing to a new jump.
  + **Interviewer might ask**: Why increment jump when i == currEnd?
    - **Response**: When i reaches currEnd, we’ve exhausted the current jump’s range, so we must make a new jump to the farthest reachable index (currMax) to continue progressing toward the last index.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: A dynamic programming approach can compute minimum jumps to each index, taking O(n²) time and O(n) space. A BFS approach treats indices as a graph, also taking O(n²) in the worst case. The greedy approach is optimal for its O(n) time.
* **Edge Cases Handled**:
  + Single element: Returns 0 (e.g., nums = [1]; no jumps needed).
  + Two elements: Returns 1 if reachable (e.g., nums = [1,0]).
  + Large jumps: Handles cases where one jump reaches far (e.g., nums = [5,1,1,1] returns 1).
  + Unreachable indices: Problem guarantees reachability, so not applicable, but code works as long as currMax progresses.
  + **Note**: Add validation for null or empty arrays for robustness.
* **Assumptions**:
  + nums is non-null, has at least one element, and contains non-negative integers.
  + The last index is always reachable (per problem constraints).
  + If these assumptions don’t hold (e.g., null array), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Dynamic Programming**: Use an array to store minimum jumps to each index, updating based on reachable indices. Time: O(n²), Space: O(n). Inefficient for large arrays.
  + **BFS**: Treat indices as a graph and find the shortest path to the last index. Time: O(n²) due to multiple possible jumps, Space: O(n). Overly complex.
  + **Forward Greedy with Different Strategy**: Track jumps differently, e.g., choosing the next jump to maximize reach, but still O(n) time and O(1) space. Similar to the current approach but less streamlined.
  + The greedy approach is preferred for its O(n) time, O(1) space, and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty arrays, e.g., “If nums is null or empty, return 0 or throw an exception.”
  + **Clarity**: Add comments explaining the greedy logic (e.g., “Track farthest reachable index to minimize jumps”).
  + **Edge Case Discussion**: In an interview, mention handling of single-element arrays, large jumps, and the guaranteed reachability to show thoroughness.
  + **Efficiency**: The approach is already optimal, but note that stopping at nums.length - 1 avoids unnecessary checks since the last index requires no further jump.
* **Note on Greedy Logic**: The algorithm greedily minimizes jumps by extending the current jump’s range (currEnd) to the farthest reachable index (currMax) when the current range is exhausted. This ensures the minimum number of jumps by always choosing the jump that maximizes progress toward the last index.

** Gas Station**

Commented Code

public class Solution {

*// Method to find the starting gas station index for a complete circuit*

public int canCompleteCircuit(int[] gas, int[] cost) {

*// If total gas is less than total cost, no solution exists*

if (Arrays.stream(gas).sum() < Arrays.stream(cost).sum()) {

return -1;

}

int total = 0; *// Running sum of gas - cost*

int res = 0; *// Starting index of valid circuit*

*// Iterate through each station*

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

total += (gas[i] - cost[i]); *// Update running sum*

*// If total becomes negative, this starting point fails*

if (total < 0) {

total = 0; *// Reset running sum*

res = i + 1; *// Try next index as starting point*

}

}

return res; *// Return the valid starting index*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The initial check compares the sum of gas and cost using Arrays.stream().sum(), which takes O(n) time, where n is the length of the arrays.
  + The main loop iterates through the n stations exactly once, performing O(1) operations per iteration (addition, comparison, and assignment).
  + Total time complexity is O(n) for the sum check plus O(n) for the loop, resulting in **O(n)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses only two integer variables (total and res), which require O(1) space.
  + The input arrays gas and cost are not counted as extra space.
  + The Arrays.stream operation does not use additional data structures beyond temporary variables.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution determines if there exists a starting gas station to complete a circular route, given gas and cost arrays. It first checks if the total gas is at least the total cost; if not, it returns -1. Then, it iterates through the stations, tracking the running sum of gas[i] - cost[i]. If the sum becomes negative, the current starting point fails, so it resets the sum and tries the next index. The final starting index is returned. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently finding the valid starting point or determining none exists.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each station must be considered, and O(1) space minimizes memory usage.
  + **Clarity**: The greedy approach is intuitive, using a single pass to identify the starting point, making it interview-friendly.
  + **Correctness**: Correctly identifies the unique starting point (or -1) by leveraging the fact that a solution exists if total gas ≥ total cost.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why check total gas vs. total cost first?
    - **Response**: If total gas is less than total cost, it’s impossible to complete the circuit, as the net fuel would be negative, making the trip infeasible regardless of starting point.
  + **Interviewer might ask**: Why reset total and update res when total < 0?
    - **Response**: A negative total means the current starting point cannot reach the current station, so all stations up to the current one are invalid starting points. Resetting total and trying the next index (i + 1) tests a new potential starting point.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: A brute-force approach could try each starting index and simulate the circuit, taking O(n²) time. The greedy approach is more efficient, leveraging the problem’s property that a valid starting point is unique if total gas ≥ total cost.
* **Edge Cases Handled**:
  + Empty arrays: Not applicable (problem guarantees non-empty arrays).
  + Single station: Returns 0 if gas[0] ≥ cost[0], else -1.
  + Total gas < total cost: Returns -1 (e.g., gas = [1], cost = [2]).
  + All stations valid: Returns 0 if circuit can start anywhere (e.g., gas = [1,1], cost = [1,1]).
  + Negative net gas at some point: Correctly identifies the next valid starting point.
  + **Note**: Add validation for null or mismatched arrays for robustness.
* **Assumptions**:
  + gas and cost are non-null, have equal length, and contain non-negative integers.
  + 1 ≤ gas.length ≤ 10^5 (per typical problem constraints).
  + If these assumptions don’t hold (e.g., null arrays), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Brute Force**: Try each index as a starting point and simulate the circuit by iterating through all stations. Time: O(n²), Space: O(1). Inefficient for large arrays.
  + **Two-Pointer Simulation**: Use a sliding window to track the circuit, but this still requires multiple passes in the worst case. Time: O(n²), Space: O(1). Less efficient.
  + The greedy approach is preferred for its O(n) time and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or mismatched arrays, e.g., “If gas or cost is null or lengths differ, throw an exception.”
  + **Clarity**: Add comments explaining the greedy logic (e.g., “If total gas < total cost, no solution; else track valid starting point by resetting on negative sum”).
  + **Edge Case Discussion**: In an interview, mention handling of single stations, insufficient gas, and all-valid cases to show thoroughness.
  + **Efficiency**: The stream-based sum check could be done in the main loop to avoid two passes, but the impact is minimal (still O(n)).
* **Note on Greedy Logic**: The algorithm relies on the fact that if total gas ≥ total cost, there is exactly one valid starting point. By tracking the running sum of gas[i] - cost[i], it identifies when a starting point fails (negative sum) and moves to the next possible index. The total gas check ensures feasibility, and the final res is valid because any negative sum invalidates prior starting points.

** Hand of Straights**

Commented Code

class Solution {

*// Method to check if hand can be rearranged into groups of consecutive cards*

public boolean isNStraightHand(int[] hand, int groupSize) {

*// Early exit if hand length is not divisible by groupSize*

if (hand.length % groupSize != 0) return false;

*// Use TreeMap to store card counts, sorted by card value*

TreeMap<Integer, Integer> cardCounts = new TreeMap<>();

*// Count frequency of each card*

for (int card : hand) {

cardCounts.put(card, cardCounts.getOrDefault(card, 0) + 1);

}

*// Attempt to form groups of size groupSize*

while (!cardCounts.isEmpty()) {

int first = cardCounts.firstKey(); *// Get smallest card value*

*// Check for consecutive cards to form a group*

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

int currentCard = first + i; *// Next card in sequence*

if (!cardCounts.containsKey(currentCard)) {

return false; *// Missing card, cannot form group*

}

*// Decrease count or remove card if count becomes 0*

int count = cardCounts.get(currentCard);

if (count == 1) {

cardCounts.remove(currentCard);

} else {

cardCounts.put(currentCard, count - 1);

}

}

}

return true; *// All cards grouped successfully*

}

}

**Time Complexity**

* **Time Complexity: O(n log n)**
  + Checking if hand.length % groupSize != 0 is O(1).
  + Building the TreeMap involves iterating through n cards, with each put operation taking O(log m) where m is the number of unique cards (m ≤ n). Total: O(n log m).
  + The while loop runs until the TreeMap is empty. In each iteration:
    - Accessing the smallest key (firstKey) takes O(log m).
    - The inner loop runs for groupSize iterations, each performing:
      * A containsKey check (O(log m)).
      * A get operation (O(log m)).
      * A remove or put operation (O(log m)).
    - Each group processes groupSize cards, and there are n / groupSize groups, so the total number of card operations is O(n).
    - Each operation involves O(log m), so the while loop contributes O(n log m).
  + Since m ≤ n, the total time is O(n log n) for building the map and processing groups.
  + Therefore, the time complexity is **O(n log n)**.

**Space Complexity**

* **Space Complexity: O(m)**
  + The TreeMap stores at most m unique card values (m ≤ n), using O(m) space.
  + Variables like first, count, and loop counters use O(1) space.
  + The input array hand is not counted as extra space.
  + Therefore, the space complexity is **O(m)**, where m is the number of unique cards (bounded by n).

**Concise Summary of the Approach**

The solution checks if an array of cards can be grouped into sequences of groupSize consecutive cards. It first verifies that the array length is divisible by groupSize. A TreeMap stores card frequencies, sorted by value. The algorithm repeatedly takes the smallest card and checks for the next groupSize - 1 consecutive cards, decrementing their counts or removing them. If any required card is missing, it returns false. If all cards are grouped, it returns true. The approach achieves **O(n log n)** time complexity and **O(m)** space complexity, efficiently forming consecutive groups.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log n) is efficient given the need to sort or maintain ordered cards, and TreeMap provides fast access to the smallest card.
  + **Clarity**: The greedy approach using a sorted map is intuitive, ensuring consecutive cards are checked in order, making it interview-friendly.
  + **Correctness**: Correctly ensures all cards are grouped into valid sequences by checking consecutive values.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why use a TreeMap instead of a HashMap?
    - **Response**: A TreeMap keeps keys sorted, allowing easy access to the smallest card (firstKey) to start each group. A HashMap would require finding the minimum key each time (O(m)), increasing complexity.
  + **Interviewer might ask**: Why check hand.length % groupSize?
    - **Response**: If the length is not divisible by groupSize, it’s impossible to form complete groups of groupSize cards, so we can exit early.
  + **Interviewer might ask**: Can you solve it without a TreeMap?
    - **Response**: Yes, sorting the array and processing groups in order is possible, taking O(n log n) time and O(1) space (excluding input). However, it requires modifying the input or extra space for sorting, and TreeMap is clearer for dynamic updates.
* **Edge Cases Handled**:
  + Empty array: Returns false if groupSize > 0 (via length check).
  + Single group (hand.length = groupSize): Checks if cards form one consecutive sequence.
  + Non-divisible length: Returns false (e.g., hand = [1,2], groupSize = 3).
  + Non-consecutive cards: Returns false if a required card is missing (e.g., hand = [1,2,4], groupSize = 3).
  + Duplicate cards: Handled by tracking frequencies in TreeMap.
  + **Note**: Add validation for null array or invalid groupSize.
* **Assumptions**:
  + hand is non-null, contains integers, and 1 ≤ groupSize ≤ hand.length.
  + Cards can be duplicated, and groups must be consecutive sequences.
  + If these assumptions don’t hold (e.g., negative groupSize), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Sorting-Based**: Sort the array and process groups by checking consecutive sequences. Time: O(n log n), Space: O(1) excluding input. Simpler but may require input modification or extra space.
  + **HashMap with Min Tracking**: Use a HashMap for frequencies and track the minimum key manually. Time: O(n²) in worst case to find minimum each time, Space: O(m). Less efficient.
  + **Frequency Array**: If card values are bounded (e.g., 1 to max value), use an array to store frequencies. Time: O(n + maxVal), Space: O(maxVal). Only viable for small value ranges.
  + The TreeMap approach is preferred for its balance of efficiency and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null array, groupSize ≤ 0, or groupSize > hand.length.
  + **Clarity**: Add comments explaining the role of TreeMap (e.g., “TreeMap keeps cards sorted to access smallest value for group formation”).
  + **Edge Case Discussion**: In an interview, mention handling of non-divisible lengths, missing cards, and duplicates to show thoroughness.
  + **Efficiency**: The TreeMap is efficient, but note that getOrDefault and put are clear and standard for frequency counting.
* **Note on Greedy Logic**: The algorithm greedily forms groups starting with the smallest available card, ensuring consecutive sequences by checking the next groupSize - 1 cards. The TreeMap maintains sorted order, and frequency updates handle duplicates. The early divisibility check ensures feasibility, and the process continues until all cards are used or a group cannot be formed.

** Merge Triplets to Form Target Triplet**

Commented Code

class Solution {

*// Method to check if triplets can be merged to form the target triplet*

public boolean mergeTriplets(int[][] triplets, int[] target) {

*// Initialize array to track maximum values for each position*

int[] maxValues = new int[3];

*// Iterate over each triplet*

for (int[] triplet : triplets) {

*// Check if triplet can contribute to target (all values <= target)*

if (triplet[0] <= target[0] && triplet[1] <= target[1] && triplet[2] <= target[2]) {

*// Update max values for each position*

maxValues[0] = Math.max(maxValues[0], triplet[0]);

maxValues[1] = Math.max(maxValues[1], triplet[1]);

maxValues[2] = Math.max(maxValues[2], triplet[2]);

}

}

*// Return true if max values match target*

return maxValues[0] == target[0] && maxValues[1] == target[1] && maxValues[2] == target[2];

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the n triplets in the triplets array exactly once.
  + For each triplet, it performs O(1) operations: three comparisons to check if the triplet’s values are ≤ target values, and three Math.max operations to update maxValues.
  + The final comparison to check if maxValues equals target is O(1).
  + Therefore, the total time complexity is **O(n)**, where n is the number of triplets.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses a fixed-size array maxValues of length 3, which requires O(1) space.
  + The input arrays triplets and target are not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution determines if a target triplet can be formed by merging valid triplets from an array, where merging takes the maximum value for each position. It tracks the maximum values for each position in a triplet, considering only triplets where all values are ≤ the corresponding target values. If the resulting maximum values match the target triplet exactly, it returns true. The approach achieves **O(n)** time complexity for a single pass through the triplets and **O(1)** space complexity, efficiently checking if the target can be formed.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each triplet must be considered, and O(1) space minimizes memory usage.
  + **Clarity**: The greedy approach of tracking maximum values is intuitive and straightforward, making it interview-friendly.
  + **Correctness**: Correctly ensures that only valid triplets contribute to the target, and the final check verifies exact matching.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why only consider triplets where all values are ≤ target?
    - **Response**: Since merging takes the maximum value for each position, any triplet with a value greater than the target’s corresponding value would overshoot the target, making it impossible to achieve the exact target triplet.
  + **Interviewer might ask**: Why track maximum values?
    - **Response**: The merge operation is defined as taking the maximum for each position across selected triplets. We need to check if we can achieve exactly target[0], target[1], and target[2] using valid triplets.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: A brute-force approach could try all combinations of triplets to check if any subset produces the target via maximums, but this would take O(2^n) time. The greedy approach is far more efficient, as it processes each triplet once.
* **Edge Cases Handled**:
  + Empty triplets array: Returns false if no triplet can provide target values (handled by maxValues remaining 0).
  + Single triplet: Checks if it’s valid and matches the target (e.g., triplets = [[2,1,3]], target = [2,1,3] returns true).
  + No valid triplets: Returns false if all triplets have at least one value > target (e.g., triplets = [[4,1,1]], target = [2,1,1]).
  + Target not achievable: Returns false if any target value cannot be reached (e.g., triplets = [[1,1,1]], target = [2,1,1]).
  + **Note**: Add validation for null arrays or invalid inputs for robustness.
* **Assumptions**:
  + triplets and target are non-null, with triplets containing at least one triplet and target having exactly three elements.
  + All values in triplets and target are positive integers.
  + If these assumptions don’t hold (e.g., null inputs), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Brute Force**: Try all possible subsets of triplets and compute the maximum for each position to check if it matches the target. Time: O(2^n), Space: O(1). Highly inefficient.
  + **Sorting-Based**: Sort triplets by each position and check if the target can be formed, but this still requires checking combinations, leading to high complexity. Time: O(n log n + complex combination logic), Space: O(1). Less practical.
  + The greedy approach is preferred for its O(n) time and simplicity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null triplets, null target, or invalid lengths (e.g., target.length != 3).
  + **Clarity**: Add comments explaining the merge logic (e.g., “Only consider triplets where all values ≤ target to avoid overshooting”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single triplets, and unachievable targets to show thoroughness.
  + **Efficiency**: The approach is already optimal, but note that iterating directly over triplets is clear and efficient.
* **Note on Greedy Logic**: The algorithm greedily collects the maximum possible values for each position from valid triplets (those with all values ≤ target). By ensuring these maximums match the target exactly, it confirms that the target triplet can be formed via merging. The early filtering of invalid triplets (with values > target) ensures correctness.

** Partition Labels**

Commented Code

**Commented Code**

public class Solution {

*// Method to partition string into segments with unique characters*

public List<Integer> partitionLabels(String s) {

*// Map to store the last index of each character*

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

*// Record last occurrence of each character*

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

lastIndex.put(s.charAt(i), i);

}

List<Integer> res = new ArrayList<>(); *// List to store partition sizes*

int size = 0; *// Current partition size*

int end = 0; *// End index of current partition*

*// Iterate through the string*

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

size++; *// Increment size of current partition*

*// Update end to the farthest last occurrence of any character in partition*

end = Math.max(end, lastIndex.get(s.charAt(i)));

*// If current index reaches the end, complete the partition*

if (i == end) {

res.add(size); *// Add partition size to result*

size = 0; *// Reset size for next partition*

}

}

return res; *// Return list of partition sizes*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + Building the lastIndex map requires iterating through the string s once, where n is the length of the string. Each put operation in the HashMap is O(1), so this step is O(n).
  + The main loop iterates through the string again, performing O(1) operations per iteration: incrementing size, accessing lastIndex (get is O(1)), updating end with Math.max, and adding to res when i == end.
  + Total time complexity is O(n) for the first loop + O(n) for the second loop, resulting in **O(n)**.

**Space Complexity**

* **Space Complexity: O(m)**
  + The lastIndex map stores at most m entries, where m is the number of unique characters in s. Since s contains only lowercase letters (per typical problem constraints), m ≤ 26, making it effectively O(1). However, for general strings, it’s O(m).
  + The res list stores partition sizes, with at most n elements in the worst case (e.g., all single-character partitions), but this is part of the output and not counted as auxiliary space.
  + Variables size and end use O(1) space.
  + The input string s is not counted as extra space.
  + Therefore, the auxiliary space complexity is **O(m)**, where m is the number of unique characters (bounded by 26 for lowercase letters).

**Concise Summary of the Approach**

The solution partitions a string into the smallest possible segments where each character appears only within one segment. It first builds a map of each character’s last occurrence. Then, it iterates through the string, tracking the current partition’s size and the farthest end index (end) based on the last occurrence of characters seen. When the current index equals end, a partition is complete, and its size is added to the result. The approach achieves **O(n)** time complexity for two passes through the string and **O(m)** space complexity (effectively O(1) for lowercase letters), efficiently partitioning the string.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as the string must be scanned at least once, and O(m) space is minimal (O(1) for lowercase letters).
  + **Clarity**: The greedy approach of tracking partition boundaries using last indices is intuitive and interview-friendly.
  + **Correctness**: Ensures each partition contains all occurrences of its characters by extending the partition to the farthest last occurrence.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why store the last index of each character?
    - **Response**: The last index ensures that all occurrences of a character are included in the same partition. By tracking the farthest last index (end), we define the smallest possible partition containing all necessary characters.
  + **Interviewer might ask**: Why complete a partition when i == end?
    - **Response**: When the current index i equals end, all characters in the current partition have been fully accounted for (no later occurrences), so we can close the partition and start a new one.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: A two-pointer approach could track start and end of partitions without a map, but it requires multiple passes or complex logic, leading to O(n²) worst-case time. The map-based approach is more efficient and clearer.
* **Edge Cases Handled**:
  + Empty string: Returns empty list (no iterations occur).
  + Single character: Returns [1] (e.g., s = "a").
  + All unique characters: Returns list of 1s (e.g., s = "abc" returns [1,1,1]).
  + Repeated characters: Correctly groups all occurrences (e.g., s = "ababa" returns [5]).
  + **Note**: Add validation for null or invalid strings for robustness.
* **Assumptions**:
  + s is non-null and contains only lowercase letters (m ≤ 26).
  + The string can be empty, returning an empty list.
  + If these assumptions don’t hold (e.g., uppercase letters), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Two-Pointer Without Map**: Iterate with a start pointer and find the end of each partition by checking if all characters between start and end appear only within that range. Time: O(n²) due to repeated scans, Space: O(1). Inefficient.
  + **Union-Find**: Treat characters with overlapping ranges as connected components and merge them into partitions. Time: O(n α(n)), Space: O(m). Complex and unnecessary.
  + **Interval Merging**: Treat character ranges as intervals and merge overlapping ones. Time: O(n log n) for sorting intervals, Space: O(m). Overly complex for this problem.
  + The map-based greedy approach is preferred for its O(n) time and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null or empty strings, e.g., “If s is null, return empty list.”
  + **Clarity**: Add comments explaining partition logic (e.g., “Extend partition to include last occurrence of each character”).
  + **Edge Case Discussion**: In an interview, mention handling of single characters, all-unique strings, and repeated characters to show thoroughness.
  + **Efficiency**: For lowercase letters, note that m ≤ 26 makes the HashMap effectively O(1) space, enhancing the approach’s appeal.
* **Note on Greedy Logic**: The algorithm greedily forms the smallest possible partitions by extending each partition to include the last occurrence of any character within it. The lastIndex map ensures all occurrences are captured, and completing a partition when i == end guarantees minimal segments while satisfying the constraint that each character appears in only one partition.

** Valid Parenthesis String**

Commented Code

class Solution {

*// Method to check if a string with parentheses and '\*' is valid*

public boolean checkValidString(String s) {

int minOpen = 0; *// Minimum possible open parentheses*

int maxOpen = 0; *// Maximum possible open parentheses*

*// Iterate through each character in the string*

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

if (c == '(') {

*// Treat '(' as an open parenthesis*

minOpen++;

maxOpen++;

} else if (c == ')') {

*// Treat ')' as a closing parenthesis*

minOpen--;

maxOpen--;

} else {

*// '\*' can be treated as '(', ')', or empty string*

minOpen--; *// Treat '\*' as ')'*

maxOpen++; *// Treat '\*' as '('*

}

*// If maxOpen < 0, too many closing parentheses*

if (maxOpen < 0) {

return false;

}

*// Ensure minOpen doesn't go negative (no unmatched closing)*

minOpen = Math.max(minOpen, 0);

}

*// Valid if minOpen is 0 (all open parentheses can be matched)*

return minOpen == 0;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The method iterates through each character in the string s exactly once, where n is the length of the string.
  + Each iteration performs O(1) operations: checking the character, updating minOpen and maxOpen, and performing comparisons.
  + Converting the string to a char array (via toCharArray) is O(n), but this is done once.
  + Total time complexity is **O(n)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses two integer variables (minOpen and maxOpen), which require O(1) space.
  + The input string s is not counted as extra space, and no additional data structures are used.
  + The toCharArray method creates a temporary array, but this is a standard iteration mechanism and considered part of input processing.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution checks if a string containing parentheses and '*' (wildcards) is a valid parenthesis string using a greedy approach. It tracks the minimum (minOpen) and maximum (maxOpen) possible counts of unmatched open parentheses. For each '(', both counts increment; for ')', both decrement; for '*', minOpen decrements (as ')') and maxOpen increments (as '('). If maxOpen becomes negative, the string is invalid (too many ')'). minOpen is capped at 0 to avoid invalid negative counts. The string is valid if minOpen is 0 at the end, ensuring all parentheses can be matched. The approach achieves **O(n)** time complexity and **O(1)** space complexity, efficiently validating the string.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time and O(1) space are optimal, as the string must be scanned at least once, and no extra storage is needed.
  + **Clarity**: The two-variable approach (minOpen, maxOpen) is intuitive, tracking the range of possible open parentheses, making it interview-friendly.
  + **Correctness**: Correctly handles the flexibility of '\*' and ensures all parentheses can be matched.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why track both minOpen and maxOpen?
    - **Response**: minOpen represents the least number of unmatched open parentheses (treating '*' as ')'), and maxOpen represents the most (treating '*' as '('). This range ensures all possible configurations are valid if minOpen reaches 0 and maxOpen stays non-negative.
  + **Interviewer might ask**: Why set minOpen to 0 if negative?
    - **Response**: A negative minOpen would imply unmatched closing parentheses, which '\*' can avoid by being treated as empty. Capping at 0 reflects this flexibility while ensuring validity.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: A stack-based approach could track possible open parentheses, or a recursive approach could explore all '\*' possibilities, but both are less efficient (O(n) space or exponential time). The greedy range-based approach is optimal.
* **Edge Cases Handled**:
  + Empty string: Returns true (minOpen = 0 initially).
  + Only '*' characters: Valid (e.g., "*\*\*" has minOpen = 0, maxOpen ≥ 0).
  + Unmatched '(': Valid if '*' can compensate (e.g., "(*" returns true).
  + Unmatched ')': Invalid if maxOpen < 0 (e.g., ")" returns false).
  + Balanced parentheses: Returns true (e.g., "()" or "(\*())").
  + **Note**: Add validation for null or invalid characters if required.
* **Assumptions**:
  + The input string s contains only '(', ')', and '\*'.
  + The string is non-null (though empty is valid).
  + If these assumptions don’t hold (e.g., other characters), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Stack-Based**: Use two stacks to track indices of '(' and '\*', matching them with ')'. Time: O(n), Space: O(n). More complex and uses more space.
  + **Recursive Backtracking**: Explore all possibilities of '*' as '(', ')', or empty. Time: O(3^n) due to three choices per '*', Space: O(n). Infeasible for large strings.
  + **Dynamic Programming**: Use DP to track valid states for prefixes. Time: O(n²), Space: O(n²). Less efficient than the greedy approach.
  + The greedy range-based approach is preferred for its O(n) time and O(1) space.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null strings or invalid characters (e.g., letters).
  + **Clarity**: Add comments explaining the role of minOpen and maxOpen (e.g., “Track range of unmatched open parentheses to account for '\*' flexibility”).
  + **Edge Case Discussion**: In an interview, mention handling of empty strings, all '\*' cases, and unbalanced parentheses to show thoroughness.
  + **Efficiency**: The approach is already optimal, but note that iterating directly on s.charAt(i) could avoid toCharArray overhead, though it’s negligible.
* **Note on Greedy Logic**: The algorithm maintains a range of possible unmatched open parentheses (minOpen to maxOpen) to account for the flexibility of '*'. By ensuring maxOpen never goes negative (no excess closing) and minOpen reaches 0 (all opens can be matched), it validates the string efficiently without exploring all '*' possibilities.

16. Intervals (6 problems)

* Insert Interval

Commented Code  
public class Solution {

*// Method to insert a new interval into a list of non-overlapping intervals*

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

List<int[]> res = new ArrayList<>(); *// List to store resulting intervals*

*// Iterate through each interval in the input*

for (int[] interval : intervals) {

*// Case 1: No overlap, current interval ends before newInterval starts*

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

res.add(interval); *// Add current interval as is*

}

*// Case 2: No overlap, current interval starts after newInterval ends*

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

res.add(newInterval); *// Add merged newInterval*

res.add(interval); *// Add current interval*

newInterval = null; *// Mark newInterval as inserted*

}

*// Case 3: Overlap, merge intervals*

else {

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

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

}

}

*// If newInterval hasn't been inserted, add it*

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

*// Convert List<int[]> to int[][]*

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

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the intervals array once, where n is the number of intervals.
  + Each iteration performs O(1) operations:
    - Checking conditions (interval[1] < newInterval[0], interval[0] > newInterval[1]).
    - Adding to res (O(1) amortized for ArrayList).
    - Updating newInterval with Math.min and Math.max (O(1)).
  + The final check and addition of newInterval (if not null) is O(1).
  + Converting the ArrayList to an array (res.toArray) is O(n) in Java, as it copies the list elements.
  + The dominant term is the loop over n intervals, so the total time complexity is **O(n)**.
  + This is optimal, as each interval must be examined to check for overlaps.

**Space Complexity**

* **Space Complexity: O(n)**
  + The output list res stores up to n + 1 intervals (all input intervals plus possibly newInterval), requiring O(n) space. This is part of the output and not counted as auxiliary space in some analyses.
  + The input intervals and newInterval are not counted as extra space.
  + No additional data structures scale with input size beyond the output list.
  + Therefore, the auxiliary space complexity is **O(1)** (excluding output), and total space including output is **O(n)**.

**Concise Summary of the Approach**

The solution inserts a new interval into a sorted list of non-overlapping intervals, merging overlapping intervals. It iterates through the input intervals, adding non-overlapping intervals that end before newInterval starts directly to the result. For overlapping intervals, it merges them by updating newInterval’s start and end. Non-overlapping intervals that start after newInterval ends trigger adding newInterval followed by the current interval. Finally, it adds newInterval if not yet inserted. The approach achieves **O(n)** time complexity for a single pass and **O(1)** auxiliary space (excluding output), efficiently producing a sorted, non-overlapping interval list.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n) time is optimal, as each interval must be checked for overlap, and O(1) auxiliary space is ideal for minimal memory usage.
  + **Clarity**: The approach processes intervals in a single pass, handling three clear cases (before, overlap, after), making it interview-friendly for demonstrating interval logic.
  + **Correctness**: Correctly merges overlapping intervals and maintains the sorted, non-overlapping property of the result.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why set newInterval = null after adding it?
    - **Response**: Setting newInterval = null marks it as inserted, ensuring it’s added only once (in the "after" case or at the end). This prevents duplicate insertions and simplifies the final check.
  + **Interviewer might ask**: Why merge by updating newInterval?
    - **Response**: Merging updates newInterval’s start to the minimum and end to the maximum of overlapping intervals, consolidating all overlaps into a single interval that’s added later, maintaining the non-overlapping property.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: An alternative is to add all intervals (including newInterval) to a list, sort by start time, and merge overlapping intervals in a second pass. This takes O(n log n) time due to sorting and O(n) space. The current approach is more efficient with O(n) time and a single pass.
* **Edge Cases Handled**:
  + Empty intervals: Adds newInterval (e.g., [], [2,3] → [[2,3]]).
  + newInterval before/after all intervals: Adds correctly (e.g., [[2,3]], [1,1] → [[1,1],[2,3]]).
  + Overlapping intervals: Merges correctly (e.g., [[1,3],[6,9]], [2,5] → [[1,5],[6,9]]).
  + newInterval spanning multiple intervals: Merges all overlaps (e.g., [[1,2],[3,5],[6,7]], [2,6] → [[1,7]]).
  + **Note**: Add validation for null inputs or invalid intervals for robustness.
* **Assumptions**:
  + intervals is a non-null array of valid intervals [start, end] where start <= end.
  + newInterval is a non-null array of length 2 with valid start <= end.
  + Intervals in intervals are sorted by start time and non-overlapping.
  + Result must be sorted and non-overlapping.
  + If these assumptions don’t hold (e.g., unsorted intervals), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Sort and Merge**: Add newInterval to the list, sort all intervals by start time, and merge overlapping intervals. Time: O(n log n), Space: O(n). Less efficient due to sorting.
  + **Two-Pointer**: Use two pointers to collect intervals before and after newInterval, merging overlaps in between. Time: O(n), Space: O(1) auxiliary. Similar but may require more complex logic.
  + **Brute Force**: Check each interval for overlap and merge manually, reconstructing the list. Time: O(n), Space: O(n). Similar but less streamlined.
  + The current approach is preferred for its O(n) time and clear case handling.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null intervals, null newInterval, or invalid intervals (e.g., start > end), e.g., “If inputs are null or invalid, throw an exception.”
  + **Clarity**: Add comments explaining cases (e.g., “Handle non-overlapping before, overlapping, and non-overlapping after cases”).
  + **Edge Case Discussion**: In an interview, mention handling of empty lists, non-overlapping cases, and full merges to show thoroughness.
  + **Efficiency**: The approach is optimal, but note that in-place modification of newInterval simplifies the merge process.
* **Note on Interval Logic**: The algorithm processes intervals in order, handling three cases: (1) current interval ends before newInterval starts (add as is), (2) current interval starts after newInterval ends (add newInterval then current interval), and (3) overlap (merge by updating newInterval). The final check ensures newInterval is added if it wasn’t inserted during the loop, maintaining the sorted, non-overlapping property.
* Merge Intervals

Commented Code

class Solution {

*// Method to merge overlapping intervals in a list of intervals*

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

*// Sort intervals by start time*

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

List<int[]> ans = new ArrayList<>(); *// List to store merged intervals*

*// Iterate through sorted intervals*

for (int[] interval : intervals) {

*// If list is empty or no overlap with last interval, add current interval*

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

ans.add(interval);

} else {

*// Overlap: merge by updating end time of last interval*

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

}

}

*// Convert List<int[]> to int[][] and return*

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

}

}

**Time Complexity**

* **Time Complexity: O(n log n)**
  + Sorting the intervals array by start time using Arrays.sort with a comparator takes O(n log n), where n is the number of intervals.
  + The subsequent loop iterates through each of the n intervals exactly once, performing O(1) operations per interval:
    - Checking ans.isEmpty() or comparing end and start times (ans.get(ans.size() - 1)[1] < interval[0]).
    - Adding to ans (O(1) amortized for ArrayList).
    - Updating the end time with Math.max (O(1)).
  + Converting the ArrayList to an array (ans.toArray) takes O(n).
  + The dominant term is the sorting step, so the total time complexity is **O(n log n)**.

**Space Complexity**

* **Space Complexity: O(n)**
  + The output list ans stores up to n merged intervals, requiring O(n) space. This is part of the output and not counted as auxiliary space in some analyses.
  + The sorting operation in Java’s Arrays.sort (Timsort) uses O(log n) space for the recursion stack in the worst case.
  + No additional data structures scale with input size beyond the output list.
  + The input intervals array is modified in place during sorting and not counted as extra space.
  + Therefore, the auxiliary space complexity is **O(log n)** due to sorting, and total space including output is **O(n)**.

**Concise Summary of the Approach**

The solution merges overlapping intervals in a list by first sorting intervals by start time. It then iterates through the sorted intervals, adding non-overlapping intervals directly to the result list and merging overlapping intervals by updating the end time of the last interval in the list to the maximum of the current and previous end times. The result is converted to an array. The approach achieves **O(n log n)** time complexity due to sorting and **O(n)** space complexity for the output, efficiently producing a sorted, non-overlapping interval list.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log n) time is near-optimal due to the sorting requirement, and O(n) space is necessary for the output.
  + **Clarity**: Sorting followed by a single-pass merge is intuitive and demonstrates interval handling, making it interview-friendly.
  + **Correctness**: Correctly merges all overlapping intervals while preserving non-overlapping ones, maintaining the sorted order.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why sort by start time?
    - **Response**: Sorting by start time ensures intervals are processed in order, making it easier to detect overlaps by comparing the current interval’s start with the previous interval’s end. This guarantees all overlaps are handled sequentially.
  + **Interviewer might ask**: Why use Math.max for the end time?
    - **Response**: When intervals overlap (current start ≤ previous end), they form a single merged interval. The end time is the maximum of the two end times to include all covered points.
  + **Interviewer might ask**: Can you solve it without sorting?
    - **Response**: Without sorting, we’d need to check each interval against all others for overlaps, taking O(n²) time in the worst case. Sorting reduces this to O(n log n) by allowing a linear pass for merging, which is more efficient for large n.
* **Edge Cases Handled**:
  + Empty array: Returns empty array ([]).
  + Single interval: Returns the interval (e.g., [[1,3]] → [[1,3]]).
  + No overlaps: Returns sorted intervals (e.g., [[1,2],[3,4]] → [[1,2],[3,4]]).
  + All overlapping: Merges into one (e.g., [[1,4],[2,3]] → [[1,4]]).
  + Multiple merges: Handles correctly (e.g., [[1,3],[2,6],[8,10],[15,18]] → [[1,6],[8,10],[15,18]]).
  + **Note**: Add validation for null or invalid intervals for robustness.
* **Assumptions**:
  + intervals is non-null and contains valid intervals [start, end] where start <= end.
  + Intervals can be in any order and may overlap.
  + Result must be sorted by start time and non-overlapping.
  + If these assumptions don’t hold (e.g., null input), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Brute Force**: Check each interval against all others to find overlaps, merging as needed. Time: O(n²), Space: O(n). Inefficient for large n.
  + **Interval Graph**: Build a graph where overlapping intervals are connected, then find connected components to merge. Time: O(n²), Space: O(n). Overly complex.
  + **Sweep Line**: Use a sweep line to process start and end points, merging intervals. Time: O(n log n), Space: O(n). More complex and similar in efficiency.
  + The current approach is preferred for its simplicity and O(n log n) time.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null intervals or invalid intervals (e.g., start > end), e.g., “If intervals is null or contains invalid intervals, throw an exception.”
  + **Clarity**: Add comments explaining merge logic (e.g., “Merge overlapping intervals by updating end time to maximum”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single intervals, and multiple merges to show thoroughness.
  + **Efficiency**: The approach is near-optimal, but note that sorting is necessary unless additional constraints (e.g., pre-sorted input) are provided.
* **Note on Merge Logic**: After sorting by start time, the algorithm processes intervals sequentially. If the current interval’s start time is greater than the last merged interval’s end time, it’s non-overlapping and added directly. Otherwise, it overlaps, so the last interval’s end time is updated to the maximum. This ensures all overlaps are consolidated into minimal non-overlapping intervals.
* Non Overlapping Intervals

Commented Code

class Solution {

*// Method to find the minimum number of intervals to remove to make the rest non-overlapping*

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

*// Handle empty input*

if (intervals.length == 0) {

return 0;

}

*// Sort intervals by start time*

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

int prev = 0; *// Index of the last non-removed interval*

int count = 0; *// Count of intervals to remove*

*// Iterate through sorted intervals starting from second interval*

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

*// Check for overlap (previous interval's end > current interval's start)*

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

*// If current interval ends earlier, keep it and remove previous*

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

prev = i;

}

count++; *// Increment removal count*

} else {

*// No overlap, update prev to current interval*

prev = i;

}

}

return count; *// Return number of intervals to remove*

}

}

**Time Complexity**

* **Time Complexity: O(n log n)**
  + Sorting the intervals array by start time using Arrays.sort with a comparator takes O(n log n), where n is the number of intervals.
  + The subsequent loop iterates through n - 1 intervals (from i = 1 to n - 1), performing O(1) operations per iteration:
    - Comparing intervals[prev][1] and intervals[i][0] for overlap.
    - Comparing intervals[prev][1] and intervals[i][1] for the greedy choice.
    - Updating prev and incrementing count.
  + The loop takes O(n) time.
  + The initial empty check is O(1).
  + The dominant term is the sorting step, so the total time complexity is **O(n log n)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm modifies the input intervals array in-place during sorting (no extra space beyond input).
  + Variables prev, count, and i use O(1) space.
  + Java’s Arrays.sort (Timsort) uses O(log n) space for the recursion stack in the worst case, but this is typically considered part of the sorting process and not counted as auxiliary space in some analyses.
  + No additional data structures are used.
  + Therefore, the auxiliary space complexity is **O(1)** (excluding sorting’s O(log n) stack), and total space is **O(1)** since the input is modified in-place.

**Concise Summary of the Approach**

The solution finds the minimum number of intervals to remove to make the remaining intervals non-overlapping. It sorts intervals by start time and iterates through them, keeping track of the last non-removed interval (prev). For each interval, if it overlaps with the previous one (end of prev > start of current), it removes the interval with the later end time (greedy choice) and increments a counter; otherwise, it updates prev. The approach achieves **O(n log n)** time complexity due to sorting and **O(1)** auxiliary space, efficiently computing the minimum removals.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log n) time is near-optimal due to sorting, and O(1) auxiliary space is ideal for in-place processing.
  + **Clarity**: The greedy approach after sorting is intuitive and demonstrates interval scheduling logic, making it interview-friendly.
  + **Correctness**: The greedy choice of keeping the interval with the earlier end time ensures the maximum number of non-overlapping intervals, minimizing removals.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why sort by start time?
    - **Response**: Sorting by start time allows processing intervals in order, making it easier to detect overlaps by comparing the previous interval’s end with the current interval’s start. This ensures all overlaps are handled sequentially.
  + **Interviewer might ask**: Why keep the interval with the earlier end time?
    - **Response**: When two intervals overlap, removing the one with the later end time minimizes conflicts with future intervals, as it frees up more space for subsequent non-overlapping intervals. This is a greedy strategy to maximize the number of kept intervals.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: Sorting by end time and selecting non-overlapping intervals greedily (similar to interval scheduling) also works, counting kept intervals and subtracting from n to get removals. Time: O(n log n), Space: O(1). The current approach is equivalent but directly counts removals, which is intuitive.
* **Edge Cases Handled**:
  + Empty array: Returns 0 ([] → 0).
  + Single interval: Returns 0 ([[1,2]] → 0).
  + No overlaps: Returns 0 (e.g., [[1,2],[3,4]] → 0).
  + All overlapping: Removes all but one (e.g., [[1,3],[1,4],[1,5]] → 2, keeping [[1,3]]).
  + Same start times: Handles correctly by comparing end times (e.g., [[1,2],[1,3]] → 1).
  + **Note**: Add validation for null or invalid intervals for robustness.
* **Assumptions**:
  + intervals is non-null and contains valid intervals [start, end] where start <= end.
  + Intervals can be in any order and may overlap.
  + If these assumptions don’t hold (e.g., null input), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Sort by End Time**: Sort intervals by end time and select non-overlapping intervals greedily, counting kept intervals and computing removals as n - kept. Time: O(n log n), Space: O(1). Equivalent but focuses on keeping intervals.
  + **Brute Force**: Check each interval against all others, removing overlaps iteratively. Time: O(n²), Space: O(1). Inefficient for large n.
  + **Interval Graph**: Build a graph where overlapping intervals are connected, find the minimum vertex cover to remove. Time: O(n²), Space: O(n). Overly complex.
  + The current approach is preferred for its simplicity and direct counting of removals.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null intervals or invalid intervals (e.g., start > end), e.g., “If intervals is null or contains invalid intervals, throw an exception.”
  + **Clarity**: Add comments explaining greedy choice (e.g., “Remove interval with later end time to minimize conflicts”).
  + **Edge Case Discussion**: In an interview, mention handling of empty arrays, single intervals, and same start times to show thoroughness.
  + **Efficiency**: The approach is near-optimal, but note that sorting by end time is an alternative with similar performance.
* **Note on Greedy Logic**: After sorting by start time, the algorithm compares each interval with the last kept interval. If they overlap (previous end > current start), it removes the interval with the later end time, as it’s more likely to conflict with future intervals. Non-overlapping intervals update the prev pointer. This greedy strategy minimizes removals by maximizing the number of non-overlapping intervals kept.
* Meeting Rooms

Commented Code

class Solution {

*// Method to determine if all meetings can be attended without conflicts*

public boolean canAttendMeetings(List<Interval> intervals) {

*// Sort intervals by start time*

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

*// Check for any overlapping meetings*

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

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

return false; *// Overlap detected, cannot attend all meetings*

}

}

return true; *// No overlaps, all meetings can be attended*

}

}

**Time Complexity**

* **Time Complexity: O(n log n)**
  + Sorting the intervals list by start time using List.sort with a comparator takes O(n log n), where n is the number of intervals.
  + The subsequent loop iterates through n - 1 intervals, performing O(1) operations per iteration: accessing intervals.get(i).end and intervals.get(i + 1).start, and comparing them.
  + The loop takes O(n) time.
  + The dominant term is the sorting step, so the total time complexity is **O(n log n)**.
  + This is optimal, as sorting is necessary to efficiently check for overlaps in a single pass.

**Space Complexity**

* **Space Complexity: O(1)**
  + The sorting operation modifies the input intervals list in-place and uses O(log n) space for the recursion stack in Java’s List.sort (Timsort), but this is typically not counted as auxiliary space in some analyses.
  + The loop uses only a single integer variable (i), requiring O(1) space.
  + The input intervals list is not counted as extra space.
  + No additional data structures are used.
  + Therefore, the auxiliary space complexity is **O(1)** (excluding sorting’s O(log n) stack).

**Concise Summary of the Approach**

The solution determines if a person can attend all meetings by checking for any overlapping intervals. It sorts the intervals by start time and iterates through them, checking if the end time of one interval exceeds the start time of the next. If any overlap is found, it returns false; otherwise, true. The approach achieves **O(n log n)** time complexity due to sorting and **O(1)** auxiliary space, efficiently detecting conflicts in a sorted list of intervals.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log n) time is optimal due to the need to sort intervals to check overlaps efficiently, and O(1) auxiliary space is ideal.
  + **Clarity**: Sorting followed by a single pass to check overlaps is straightforward and demonstrates interval handling, making it interview-friendly.
  + **Correctness**: Correctly identifies overlaps by leveraging sorted order, ensuring no conflicts are missed.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why sort by start time?
    - **Response**: Sorting by start time ensures intervals are processed in chronological order, allowing a single pass to check for overlaps by comparing each interval’s end time with the next interval’s start time. Without sorting, we’d need O(n²) time to check all pairs.
  + **Interviewer might ask**: Why check end > start for overlap?
    - **Response**: Two intervals overlap if one starts before the other ends. After sorting by start time, if intervals[i].end > intervals[i+1].start, the current meeting extends past the next meeting’s start, causing a conflict.
  + **Interviewer might ask**: Can you solve it without sorting?
    - **Response**: Without sorting, we’d need to compare each interval with all others, taking O(n²) time in the worst case. Sorting reduces this to O(n log n) by enabling a linear scan for overlaps, which is more efficient for large n.
* **Edge Cases Handled**:
  + Empty list: Returns true (no meetings, no conflicts).
  + Single interval: Returns true (no overlaps possible).
  + Adjacent intervals: Handles non-overlapping case (e.g., [[1,2],[2,3]] → true).
  + Overlapping intervals: Detects conflict (e.g., [[1,3],[2,4]] → false).
  + Same start times: Correctly checks end times (e.g., [[1,3],[1,4]] → false if ends overlap).
  + **Note**: Add validation for null or invalid intervals for robustness.
* **Assumptions**:
  + intervals is a non-null List<Interval> with valid intervals (start <= end).
  + Interval class has start and end fields (integers).
  + If these assumptions don’t hold (e.g., null list or invalid intervals), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Brute Force**: Check each interval against all others for overlaps. Time: O(n²), Space: O(1). Inefficient for large n.
  + **Sort by End Time**: Sort by end time and check for conflicts, but start-time sorting is more natural for chronological checks. Time: O(n log n), Space: O(1). Similar efficiency but less intuitive.
  + **Sweep Line**: Process start and end points in sorted order, tracking active meetings. Time: O(n log n), Space: O(n). More complex and unnecessary for this problem.
  + The current approach is preferred for its simplicity and efficiency.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null intervals, empty list (handled), or invalid intervals (e.g., start > end), e.g., “If intervals is null or contains invalid intervals, throw an exception.”
  + **Clarity**: Add comments explaining overlap condition (e.g., “Check if current interval’s end exceeds next interval’s start for overlap”).
  + **Edge Case Discussion**: In an interview, mention handling of empty lists, single intervals, adjacent intervals, and same start times to show thoroughness.
  + **Efficiency**: The approach is optimal, but note that sorting is necessary unless the input is guaranteed sorted.
* **Note on Overlap Logic**: After sorting by start time, the algorithm checks for overlaps by comparing each interval’s end time with the next interval’s start time. If intervals[i].end > intervals[i+1].start, the meetings conflict, as one starts before the other ends. This linear scan after sorting efficiently detects any overlap, returning false immediately or true if no conflicts are found.
* Meeting Rooms II

Commented Code

*/\*\**

*\* 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 {

*// Method to find the minimum number of meeting rooms required*

public int minMeetingRooms(List<Interval> intervals) {

int n = intervals.size();

*// Arrays to store start and end times*

int[] start = new int[n];

int[] end = new int[n];

*// Populate start and end arrays*

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

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

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

}

*// Sort start and end times separately*

Arrays.sort(start);

Arrays.sort(end);

int res = 0, count = 0, s = 0, e = 0; *// res: max rooms, count: current rooms, s/e: start/end pointers*

*// Process events in chronological order*

while (s < n) {

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

s++; *// Start a new meeting*

count++; *// Increment room count*

} else {

e++; *// End a meeting*

count--; *// Decrement room count*

}

res = Math.max(res, count); *// Update max rooms needed*

}

return res; *// Return minimum rooms required*

}

}

**Time Complexity**

* **Time Complexity: O(n log n)**
  + Extracting start and end times into arrays takes O(n), where n is the number of intervals, with O(1) operations per interval (intervals.get(i).start/end).
  + Sorting the start and end arrays using Arrays.sort (Timsort) takes O(n log n) each, for a total of O(n log n).
  + The while loop processes n start and n end events, incrementing either s or e each iteration, for a total of O(n) iterations. Each iteration performs O(1) operations: comparison, increment/decrement, and updating res with Math.max.
  + The dominant term is the sorting step, so the total time complexity is **O(n log n)**.
  + This is optimal, as sorting is necessary to process events chronologically.

**Space Complexity**

* **Space Complexity: O(n)**
  + The start and end arrays each require O(n) space to store n integers.
  + Variables n, res, count, s, and e use O(1) space.
  + The input intervals list is not counted as extra space.
  + Java’s Arrays.sort uses O(log n) space for the recursion stack, but this is typically not counted as auxiliary space in some analyses.
  + Therefore, the total space complexity is **O(n)** due to the start and end arrays, and auxiliary space is **O(n)** (excluding sorting’s O(log n) stack).

**Concise Summary of the Approach**

The solution determines the minimum number of meeting rooms needed for a list of intervals. It extracts start and end times into separate arrays, sorts them, and processes events chronologically using two pointers (s for start, e for end). When a start time is less than the next end time, a meeting begins, incrementing the room count; otherwise, a meeting ends, decrementing the count. The maximum room count encountered is returned. The approach achieves **O(n log n)** time complexity due to sorting and **O(n)** space complexity for the arrays, efficiently computing the minimum rooms required.

**Interview Preparation Notes**

* **Strengths of the Approach**:
  + **Efficiency**: O(n log n) time is optimal due to sorting, and O(n) space is reasonable for storing start and end times.
  + **Clarity**: The sweep line approach (processing sorted start/end events) is elegant and demonstrates event-based processing, making it interview-friendly.
  + **Correctness**: Correctly tracks concurrent meetings by maintaining a running count, ensuring the maximum reflects the minimum rooms needed.
* **Potential Follow-Up Questions**:
  + **Interviewer might ask**: Why sort start and end times separately?
    - **Response**: Sorting start and end times allows processing all events (meeting starts and ends) in chronological order. When a start time is next, a room is needed; when an end time is next, a room is freed. This tracks the maximum number of concurrent meetings.
  + **Interviewer might ask**: Why use Math.max to update res?
    - **Response**: res tracks the maximum number of rooms needed at any point, which corresponds to the peak number of overlapping meetings (count). Updating res with Math.max ensures we capture the highest concurrency.
  + **Interviewer might ask**: Can you solve it with a different approach?
    - **Response**: An alternative is to sort intervals by start time and use a min-heap to track end times of active meetings. For each interval, remove ended meetings from the heap and add the new end time, tracking the heap’s maximum size. Time: O(n log n), Space: O(n). The sweep line approach is simpler and equally efficient.
* **Edge Cases Handled**:
  + Empty list: Returns 0 (no meetings, no rooms needed).
  + Single interval: Returns 1 (e.g., [[1,3]] → 1).
  + Non-overlapping intervals: Returns 1 (e.g., [[1,2],[3,4]] → 1).
  + All overlapping: Returns n (e.g., [[1,5],[1,5],[1,5]] → 3).
  + Adjacent intervals: Counts correctly (e.g., [[1,2],[2,3]] → 1).
  + **Note**: Add validation for null or invalid intervals for robustness.
* **Assumptions**:
  + intervals is a non-null List<Interval> with valid intervals (start <= end).
  + Interval class has start and end fields (integers).
  + If these assumptions don’t hold (e.g., null list), clarify with the interviewer.
* **Alternative Approaches** (Description Only):
  + **Min-Heap**: Sort intervals by start time, use a min-heap to track end times of active meetings. Remove ended meetings and add new end times, tracking max heap size. Time: O(n log n), Space: O(n). More complex but viable.
  + **Brute Force**: Check each interval against others to count maximum overlaps at any point. Time: O(n²), Space: O(1). Inefficient for large n.
  + **Interval Graph**: Build a graph of overlapping intervals and find the maximum clique. Time: O(n²), Space: O(n). Overly complex.
  + The current approach is preferred for its simplicity and clarity.
* **Suggestions for Improvement**:
  + **Input Validation**: Add checks for null intervals, empty list (handled), or invalid intervals (e.g., start > end), e.g., “If intervals is null or contains invalid intervals, throw an exception.”
  + **Clarity**: Add comments explaining sweep line logic (e.g., “Process start/end events chronologically to track concurrent meetings”).
  + **Edge Case Discussion**: In an interview, mention handling of empty lists, single intervals, adjacent intervals, and all overlapping cases to show thoroughness.
  + **Efficiency**: The approach is optimal, but note that the min-heap approach is an alternative with similar complexity.
* **Note on Sweep Line Logic**: The algorithm treats meeting starts and ends as events, sorting them separately to process in chronological order. When a start event occurs (start[s] < end[e]), a room is allocated (count++). When an end event occurs, a room is freed (count--). The maximum count reflects the minimum rooms needed to accommodate all meetings without conflicts.

17. Math & Geometry (8 problems)

* Rotate Image

Commented Code

class Solution {

*// Method to rotate an n x n matrix 90 degrees clockwise in-place*

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

int n = matrix.length; *// Size of the matrix*

*// Iterate over layers (up to half the matrix size)*

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

*// Iterate over elements in each layer (up to half the row length)*

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

*// Store top-left element of the current group*

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

*// Rotate four elements in a group:*

*// Bottom-left -> Top-left*

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

*// Bottom-right -> Bottom-left*

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

*// Top-right -> Bottom-right*

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

*// Top-left (temp) -> Top-right*

matrix[i][j] = temp;

}

}

}

}

**Time Complexity**

* **Time Complexity: O(n²)**
  + The input is an n x n matrix, and the algorithm iterates over approximately n/2 rows (i < (n + 1) / 2) and n/2 columns (j < n / 2) in the nested loops.
  + This covers roughly n²/4 elements (one quadrant of the matrix), as each iteration processes a group of four elements (one from each quadrant) to perform the 90-degree clockwise rotation.
  + Each iteration performs O(1) operations: one temporary storage and four assignments.
  + Therefore, the total time complexity is **O(n²)**, as it processes a constant fraction of the n² elements in the matrix.
  + This is optimal, as all elements must be visited to achieve the rotation.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm performs the rotation in-place, using only a single integer variable (temp) for swapping, which is O(1).
  + The variables n, i, and j also use O(1) space.
  + The input matrix matrix is modified in place and not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution rotates an n x n matrix 90 degrees clockwise in-place by processing groups of four elements in a layer-by-layer fashion. It iterates over the top-left quadrant (i < (n + 1) / 2, j < n / 2), swapping elements in a cycle: top-left → top-right, top-right → bottom-right, bottom-right → bottom-left, and bottom-left → top-left, using a temporary variable. The approach achieves **O(n²)** time complexity to process all elements and **O(1)** space complexity for in-place rotation, efficiently transforming the matrix.

* Spiral Matrix

Commented Code

public class Solution {

*// Method to traverse an m x n matrix in spiral order and return elements as a list*

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

List<Integer> res = new ArrayList<>(); *// List to store spiral order elements*

*// Directions for right, down, left, up*

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

*// Steps for horizontal (cols) and vertical (rows - 1) movements*

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

int r = 0, c = -1, d = 0; *// Start at (0, -1), direction index 0 (right)*

*// Continue while steps remain in current direction*

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

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

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

r += directions[d][0]; *// Update row*

c += directions[d][1]; *// Update column*

res.add(matrix[r][c]); *// Add element to result*

}

steps[d % 2]--; *// Decrease steps for horizontal (d=0,2) or vertical (d=1,3)*

d = (d + 1) % 4; *// Move to next direction (right -> down -> left -> up)*

}

return res; *// Return spiral order elements*

}

}

**Time Complexity**

* **Time Complexity: O(m \* n)**
  + The algorithm visits each element in the m x n matrix exactly once, adding it to the result list.
  + The outer while loop continues until all elements are processed, controlled by steps[d % 2], which tracks remaining horizontal (n) and vertical (m - 1) steps.
  + The inner for loop performs O(1) operations per element: updating r and c, adding to res, and array access.
  + The total number of elements processed is m \* n, so the time complexity is **O(m \* n)**.
  + This is optimal, as every element must be visited to produce the spiral order.

**Space Complexity**

* **Space Complexity: O(m \* n) for output, O(1) auxiliary**
  + The output list res stores all m \* n elements of the matrix, requiring O(m \* n) space. This is required by the problem and not counted as auxiliary space.
  + The algorithm uses:
    - A fixed-size directions array (4 x 2), O(1).
    - A steps array of size 2, O(1).
    - Integer variables r, c, d, and i, O(1).
  + No additional data structures scale with input size.
  + Therefore, the auxiliary space complexity is **O(1)**, excluding the output list.

**Concise Summary of the Approach**

The solution traverses an m x n matrix in spiral order (right, down, left, up) to return elements as a list. It uses a directions array to define movements and a steps array to track remaining horizontal and vertical steps. Starting at (0, -1), it iterates through each direction, moving the specified number of steps, adding elements to the result, and reducing steps after each direction change. The approach achieves **O(m \* n)** time complexity to visit all elements and **O(1)** auxiliary space, efficiently producing the spiral order.

* Set Matrix Zeroes

Commented Code

class Solution {

*// Method to set entire row and column to 0 for each 0 in the matrix, in-place*

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

Boolean firstCol = false; *// Flag to track if first column needs to be zeroed*

int r = matrix.length; *// Number of rows*

int c = matrix[0].length; *// Number of columns*

*// First pass: Mark rows and columns to be zeroed using first row and column*

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

*// Check if first column has a 0*

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

firstCol = true;

}

*// Check other columns, mark first row and column if 0 is found*

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

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 0 based on marks in first row and column*

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

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

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

matrix[i][j] = 0; *// Set element to 0 if row or column is marked*

}

}

}

*// Handle first row separately if matrix[0][0] is 0*

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

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

matrix[0][j] = 0; *// Zero out first row*

}

}

*// Handle first column if any element in it was 0*

if (firstCol) {

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

matrix[i][0] = 0; *// Zero out first column*

}

}

}

}

**Time Complexity**

* **Time Complexity: O(m \* n)**
  + **First pass**: Iterates over all elements in the m x n matrix, with O(1) operations per element (checking for 0 and marking first row/column). This takes O(m \* n).
  + **Second pass**: Iterates over elements from i = 1 to r-1 and j = 1 to c-1, a subset of m \* n, with O(1) operations per element (checking marks and setting to 0). This is O(m \* n).
  + **First row handling**: Iterates over c columns if matrix[0][0] == 0, taking O(n).
  + **First column handling**: Iterates over m rows if firstCol is true, taking O(m).
  + The dominant term is the matrix traversal, so the total time complexity is **O(m \* n)**, where m is the number of rows and n is the number of columns.
  + This is optimal, as all elements must be visited to check for zeros and update the matrix.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses the first row and first column of the matrix as markers, avoiding additional data structures.
  + A single boolean variable (firstCol) and integer variables (r, c, i, j) require O(1) space.
  + The input matrix matrix is modified in place and not counted as extra space.
  + Therefore, the space complexity is **O(1)**, which is optimal for an in-place solution.

**Concise Summary of the Approach**

The solution sets entire rows and columns to zero for each zero in an m x n matrix, in-place. It uses the first row and column as markers to indicate which rows and columns need zeroing, with a firstCol flag to track zeros in the first column. In the first pass, it marks the first cell of each row and column containing a zero. In the second pass, it zeros out elements based on these markers. Finally, it handles the first row and column separately. The approach achieves **O(m \* n)** time complexity to process all elements and **O(1)** space complexity, efficiently modifying the matrix in-place.

* Happy Number / Non-Cyclical Number

Commented Code

class Solution {

*// Method to determine if a number is happy*

public static boolean isHappy(int n) {

Set<Integer> seenNumbers = new HashSet<>(); *// Track numbers to detect cycles*

*// Continue until n becomes 1 (happy) or a cycle is detected*

while (n != 1 && !seenNumbers.contains(n)) {

seenNumbers.add(n); *// Add current number to seen set*

n = getSumOfSquares(n); *// Compute sum of squares of digits*

}

return n == 1; *// Return true if n is 1, false if cycle detected*

}

*// Helper method to compute sum of squares of digits*

private static int getSumOfSquares(int n) {

int sum = 0;

while (n > 0) {

int digit = n % 10; *// Extract last digit*

sum += digit \* digit; *// Add square of digit to sum*

n /= 10; *// Remove last digit*

}

return sum;

}

}

**Time Complexity**

* **Time Complexity: O(log n) for happy numbers, O(k \* log n) for non-happy numbers**
  + The getSumOfSquares method processes each digit of n. Since n has O(log n) digits (base 10), computing the sum of squares takes O(log n) time.
  + The main while loop in isHappy iterates until either:
    - n becomes 1 (happy number), which typically happens quickly (O(log n) iterations for most inputs, as numbers tend to reduce rapidly).
    - A cycle is detected (non-happy number), which depends on the cycle length in the sequence of sums. In the worst case, the number of distinct sums before a cycle is detected is bounded by a constant or logarithmic factor for 32-bit integers.
  + Each iteration of the while loop performs:
    - O(1) operations for HashSet operations (contains and add are O(1) on average).
    - O(log n) for getSumOfSquares.
  + For happy numbers, the total time is roughly O(log n \* log n) = O(log² n), but in practice, it’s closer to O(log n) due to rapid convergence.
  + For non-happy numbers, the number of iterations depends on the cycle length, which is typically small (e.g., bounded by a constant or logarithmic factor for 32-bit integers). Thus, the worst-case time is O(k \* log n), where k is the number of sums before a cycle.
  + Since k is effectively constant for practical inputs (due to 32-bit integer constraints), the time complexity is often approximated as **O(log n)**.

**Space Complexity**

* **Space Complexity: O(k)**
  + The HashSet (seenNumbers) stores each unique sum of squares encountered until either n becomes 1 or a cycle is detected.
  + The number of unique sums (k) is typically small (logarithmic or constant for 32-bit integers), as the sequence either converges to 1 or enters a cycle (e.g., common cycles like 4 → 16 → 37 → 58 → 89 → 145 → 42 → 20 → 4).
  + The getSumOfSquares method uses a few integer variables (sum, digit, n), requiring O(1) space.
  + The input n is not counted as extra space.
  + Therefore, the space complexity is **O(k)**, where k is the number of unique sums (often approximated as O(log n) or O(1) for practical purposes due to cycle length bounds).

**Concise Summary of the Approach**

The solution determines if a number is happy (i.e., the sum of squares of its digits eventually reaches 1) by iteratively computing the sum of squares of digits and tracking seen numbers in a HashSet to detect cycles. If the number reaches 1, it returns true; if a cycle is detected, it returns false. The helper method getSumOfSquares computes the sum of squared digits. The approach achieves **O(log n)** time complexity (approximated for practical inputs) and **O(k)** space complexity, where k is the number of unique sums, efficiently identifying happy numbers.

* Plus One

Commented Code

class Solution {

*// Method to increment a number represented as an array of digits by one*

public static int[] plusOne(int[] digits) {

*// Start from the last digit and move leftward*

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

if (digits[i] < 9) {

digits[i]++; *// Increment digit and return (no carry needed)*

return digits;

}

*// If digit is 9, set to 0 and continue to next digit (carry)*

digits[i] = 0;

}

*// If all digits were 9, create new array with extra digit*

int[] newDigits = new int[digits.length + 1];

newDigits[0] = 1; *// Set most significant digit to 1, rest are 0*

return newDigits;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the array from the last index (digits.length - 1) to 0, where n is the length of the array.
  + Each iteration performs O(1) operations: checking if digits[i] < 9, incrementing or setting to 0, and possibly returning.
  + In the worst case (e.g., digits = [9,9,9]), it processes all n digits and then creates a new array of size n + 1, which is O(n) for allocation and O(1) for setting newDigits[0] = 1 (other elements are 0 by default).
  + Therefore, the total time complexity is **O(n)**, dominated by the loop over the digits.

**Space Complexity**

* **Space Complexity: O(1) or O(n)**
  + In the common case, the algorithm modifies the input array digits in place (incrementing a digit or setting 9s to 0), using only a loop variable (i), which is O(1) auxiliary space.
  + In the worst case (e.g., all digits are 9), it creates a new array newDigits of size n + 1, requiring O(n) space.
  + Since the problem requires returning an array, the output space is not counted as auxiliary space. However, the worst-case creation of newDigits is considered extra space in some analyses.
  + Conservatively, the space complexity is **O(n)** due to the potential new array, but auxiliary space is **O(1)** for in-place modifications in most cases.

**Concise Summary of the Approach**

The solution increments a number represented as an array of digits by one. It iterates backward through the array, incrementing the first digit less than 9 and returning the array, or setting 9s to 0 to handle carry. If all digits are 9, it creates a new array with an extra digit, setting the most significant digit to 1. The approach achieves **O(n)** time complexity for a single pass through the digits and **O(1)** auxiliary space (or O(n) worst-case space for the new array), efficiently handling the increment operation.

* Pow(x, n)

Commented Code

class Solution {

*// Method to compute x raised to the power n*

public double myPow(double x, int n) {

*// Base case: any number raised to power 0 is 1*

if (n == 0) return 1;

*// Convert n to long to handle Integer.MIN\_VALUE edge case*

long N = n;

*// Handle negative exponent by using reciprocal of x*

if (N < 0) {

x = 1 / x;

N = -N;

}

double result = 1; *// Initialize result*

double currentProduct = x; *// Current power of x*

*// Exponentiation by squaring*

while (N > 0) {

*// If exponent is odd, multiply result by currentProduct*

if (N % 2 == 1) {

result \*= currentProduct;

}

*// Square the currentProduct*

currentProduct \*= currentProduct;

*// Divide exponent by 2*

N /= 2;

}

return result; *// Return final result*

}

}

**Time Complexity**

* **Time Complexity: O(log |n|)**
  + The algorithm uses exponentiation by squaring, which reduces the exponent n by half in each iteration of the while loop through integer division (N /= 2).
  + The number of iterations is proportional to the number of bits in |n|, which is O(log |n|), as each division by 2 corresponds to processing one bit of the exponent.
  + Each iteration performs O(1) operations: checking if N is odd (N % 2), multiplying result (if needed), squaring currentProduct, and dividing N.
  + Initial checks (n == 0, N < 0) and conversions are O(1).
  + Therefore, the total time complexity is **O(log |n|)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses a fixed number of variables: N (long), x (double), result (double), and currentProduct (double), all requiring O(1) space.
  + No additional data structures or recursion stack are used.
  + The input x and n are not counted as extra space.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution computes x raised to the power n using exponentiation by squaring. It handles the base case (n = 0), converts n to a long to manage Integer.MIN\_VALUE, and handles negative exponents by using the reciprocal of x. The algorithm iteratively squares the base and halves the exponent, multiplying the result by the current power when the exponent is odd. It achieves **O(log |n|)** time complexity by reducing the exponent logarithmically and **O(1)** space complexity, efficiently computing the power while handling edge cases like overflow.

* Multiply Strings

Commented Code

class Solution {

*// Method to multiply two numbers represented as strings*

public String multiply(String num1, String num2) {

*// If either number is "0", return "0"*

if (num1.equals("0") || num2.equals("0")) return "0";

*// Initialize result array to hold product (size = len1 + len2)*

int[] result = new int[num1.length() + num2.length()];

*// Iterate over digits of num1 and num2 from right to left*

for (int i = num1.length() - 1; i >= 0; i--) {

for (int j = num2.length() - 1; j >= 0; j--) {

*// Convert characters to integers*

int digit1 = num1.charAt(i) - '0';

int digit2 = num2.charAt(j) - '0';

int mul = digit1 \* digit2;

*// Calculate positions in result array*

int posLow = i + j + 1; *// Position for units digit*

int posHigh = i + j; *// Position for tens digit (carry)*

*// Add product to current position and handle carry*

int sum = mul + result[posLow];

result[posLow] = sum % 10; *// Store units digit*

result[posHigh] += sum / 10; *// Add carry to next position*

}

}

*// Build result string, skipping leading zeros*

StringBuilder product = new StringBuilder();

for (int num : result) {

if (!(product.length() == 0 && num == 0)) {

product.append(num);

}

}

*// Return "0" if result is empty, otherwise return the product string*

return product.length() == 0 ? "0" : product.toString();

}

}

**Time Complexity**

* **Time Complexity: O(m \* n)**
  + The algorithm uses two nested loops to process each digit of num1 (length m) and num2 (length n), performing O(1) operations per pair: character-to-integer conversion, multiplication, and updating the result array.
  + This results in m \* n iterations, each with O(1) work, so the nested loops take O(m \* n).
  + Building the result string involves iterating over the result array of size m + n, which is O(m + n).
  + The initial check for "0" is O(1) (string comparison is bounded by small input sizes).
  + Total time complexity is dominated by the nested loops: **O(m \* n)**, where m and n are the lengths of num1 and num2.

**Space Complexity**

* **Space Complexity: O(m + n)**
  + The result array has size m + n to store the digits of the product, requiring O(m + n) space.
  + The StringBuilder (product) stores up to m + n digits, also O(m + n) space. This is part of the output and not counted as auxiliary space in some analyses.
  + Temporary variables (digit1, digit2, mul, sum, posLow, posHigh) use O(1) space.
  + The input strings num1 and num2 are not counted as extra space.
  + Therefore, the auxiliary space complexity is **O(m + n)** for the result array, and total space including output is **O(m + n)**.

**Concise Summary of the Approach**

The solution multiplies two numbers represented as strings by simulating manual multiplication. It handles the base case of either number being "0" and uses an array of size m + n to store the product digits. It processes each pair of digits from num1 and num2 (right to left), computing their product and adding it to the appropriate positions with carry handling. The result array is converted to a string, skipping leading zeros. The approach achieves **O(m \* n)** time complexity for processing digit pairs and **O(m + n)** space complexity for the result array, efficiently computing the product.

* Detect Squares

Commented Code

class CountSquares {

List<int[]> coordinates; *// Stores all points as [x, y] arrays*

Map<String, Integer> counts; *// Maps point coordinates (x@y) to their frequency*

*// Constructor to initialize data structures*

public CountSquares() {

coordinates = new ArrayList<>();

counts = new HashMap<>();

}

*// Method to add a point to the collection*

public void add(int[] point) {

coordinates.add(point); *// Add point to list*

String key = point[0] + "@" + point[1]; *// Create key as "x@y"*

counts.put(key, counts.getOrDefault(key, 0) + 1); *// Increment frequency*

}

*// Method to count squares with given point as a vertex*

public int count(int[] point) {

int sum = 0, px = point[0], py = point[1]; *// Coordinates of query point*

for (int[] coordinate : coordinates) {

int x = coordinate[0], y = coordinate[1]; *// Coordinates of stored point*

*// Skip if points are on same x or y axis, or not equidistant diagonally*

if (px == x || py == y || (Math.abs(px - x) != Math.abs(py - y)))

continue;

*// Count squares by checking for other two vertices*

sum += counts.getOrDefault(x + "@" + py, 0) \* counts.getOrDefault(px + "@" + y, 0);

}

return sum; *// Return total number of squares*

}

}

**Time Complexity**

* **Constructor (CountSquares): O(1)**
  + Initializes an empty ArrayList and HashMap, both O(1).
* **add Method: O(1)**
  + Adding a point to the ArrayList (coordinates.add) is O(1) amortized.
  + Creating the key string (x + "@" + y) is O(1) for integer-to-string conversion (bounded by small integer sizes).
  + Updating the HashMap (counts.put and getOrDefault) is O(1) on average.
  + Total: **O(1)** per call.
* **count Method: O(n)**
  + Iterates through all n points in coordinates, where n is the number of points added.
  + For each point:
    - Accessing coordinates and performing comparisons (px == x, py == y, Math.abs) is O(1).
    - Checking Math.abs(px - x) != Math.abs(py - y) is O(1).
    - Two HashMap lookups (getOrDefault) are O(1) on average.
    - Multiplication and addition to sum are O(1).
  + Total per iteration: O(1), so the loop takes O(n).
  + Therefore, the time complexity is **O(n)**, where n is the number of points added.

**Space Complexity**

* **Space Complexity: O(n)**
  + The ArrayList (coordinates) stores n points, each an int[] of size 2, requiring O(n) space.
  + The HashMap (counts) stores at most n unique points, with keys as strings (x@y) and values as integers, requiring O(n) space. The string keys are small (bounded by integer sizes).
  + Temporary variables (sum, px, py, etc.) use O(1) space.
  + Therefore, the space complexity is **O(n)** for storing all points and their frequencies.

**Concise Summary of the Approach**

The CountSquares class maintains a list of points and a frequency map to count axis-aligned squares formed with a query point as one vertex. The add method stores a point and updates its frequency in a HashMap using a string key (x@y). The count method iterates through stored points, identifying potential opposite vertices of a square (diagonal points where |px - x| = |py - y|). For each valid diagonal point, it multiplies the frequencies of the other two vertices (at (x, py) and (px, y)) to count squares. The approach achieves **O(1)** time for add, **O(n)** time for count, and **O(n)** space, efficiently handling square counting.

18. Bit Manipulation (7 problems)

* Single Number

Commented Code

public class Solution {

*// Method to find the single number that appears once in the array*

public int singleNumber(int[] nums) {

int res = 0; *// Initialize result for XOR operation*

*// Iterate through each number in the array*

for (int num : nums) {

res ^= num; *// XOR each number with the result*

}

return res; *// Return the single number*

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates through the array nums exactly once, where n is the length of the array.
  + Each iteration performs a single XOR operation, which is O(1).
  + Therefore, the total time complexity is **O(n)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses only one integer variable (res) to store the running XOR result, requiring O(1) space.
  + The input array nums is not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution finds the single number in an array where every other number appears exactly twice, using the XOR operation. It iterates through the array, XORing each number with a running result (res). Since XOR of a number with itself is 0 and XOR with 0 is the number itself, all paired numbers cancel out, leaving the single number. The approach achieves **O(n)** time complexity for a single pass and **O(1)** space complexity, efficiently identifying the single number.

* Number of 1 Bits

Commented Code

class Solution {

*// Method to count the number of 1 bits in an integer (Hamming Weight)*

public int hammingWeight(int n) {

int res = 0; *// Counter for 1 bits*

*// Process until n becomes 0*

while (n != 0) {

n = n & (n - 1); *// Clear the least significant 1 bit*

res++; *// Increment count for each 1 bit*

}

return res; *// Return total number of 1 bits*

}

}

**Time Complexity**

* **Time Complexity: O(k)**
  + The algorithm iterates until n becomes 0, where k is the number of 1 bits in n.
  + Each iteration performs a bitwise AND operation (n & (n - 1)) and increments res, both O(1) operations.
  + Since n is a 32-bit integer (per problem constraints), k ≤ 32, making the worst-case time complexity O(32), which is effectively **O(1)** for a fixed-size input (32 bits).
  + For general analysis, it’s O(k), where k is the number of 1 bits, but practically **O(1)** due to the 32-bit constraint.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses only one integer variable (res) for counting, requiring O(1) space.
  + The input n is not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution counts the number of 1 bits (Hamming Weight) in a 32-bit integer using a bitwise operation. It repeatedly applies n & (n - 1) to clear the least significant 1 bit in n, incrementing a counter (res) each time, until n becomes 0. The approach achieves **O(1)** time complexity (since the number of 1 bits is at most 32) and **O(1)** space complexity, efficiently computing the Hamming Weight.

* Counting Bits

Commented Code

**e Complexity**

* **Time Complexity: O(n)**
  + The algorithm iterates from i = 1 to n, performing n iterations.
  + Each iteration involves:
    - A bitwise operation i & (i - 1), which is O(1).
    - An array lookup ans[i & (i - 1)], which is O(1).
    - An addition and assignment to ans[i], which is O(1).
  + Initializing the array ans of size n + 1 is O(1) in terms of computation (memory allocation is handled by the space complexity).
  + Therefore, the total time complexity is **O(n)**.

**Space Complexity**

* **Space Complexity: O(n)**
  + The algorithm uses an array ans of size n + 1 to store the number of 1 bits for each number from 0 to n, requiring O(n) space.
  + The input n and loop variable i use O(1) space.
  + The output array is required by the problem, but even excluding it, the auxiliary space for computation is O(1).
  + Therefore, the space complexity is **O(n)** for the output array.

**Concise Summary of the Approach**

The solution computes the number of 1 bits (Hamming Weight) for all integers from 0 to n using a dynamic programming approach with a bitwise trick. It initializes an array with ans[0] = 0 and, for each i from 1 to n, calculates the number of 1 bits as ans[i & (i - 1)] + 1, where i & (i - 1) clears the least significant 1 bit of i. The approach achieves **O(n)** time complexity for a single pass and **O(n)** space complexity for the output array, efficiently computing the 1-bit counts.

* Reverse Bits

Commented Code

class Solution {

*// Method to reverse the bits of a 32-bit integer*

public int reverseBits(int n) {

int reverse = 0; *// Initialize result for reversed bits*

*// Process all 32 bits*

for (int i = 0; i < 32; i++) {

reverse = reverse << 1; *// Left shift to make space for next bit*

reverse = reverse | (n & 1); *// Add least significant bit of n*

n = n >> 1; *// Right shift n to process next bit*

}

return reverse; *// Return the reversed bits*

}

}

**Time Complexity**

* **Time Complexity: O(1)**
  + The algorithm iterates exactly 32 times, as it processes each bit of the 32-bit integer n.
  + Each iteration performs O(1) operations: a left shift (reverse << 1), a bitwise AND (n & 1), a bitwise OR (reverse | ...), and a right shift (n >> 1).
  + Since the number of iterations is fixed at 32 (due to the 32-bit integer constraint), the total time complexity is **O(1)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses only one integer variable (reverse) to store the result, requiring O(1) space.
  + The loop variable i and input n also use O(1) space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution reverses the bits of a 32-bit integer by iterating 32 times. In each iteration, it left-shifts the result (reverse) to make space, adds the least significant bit of the input (n & 1) using OR, and right-shifts the input to process the next bit. The approach achieves **O(1)** time complexity for a fixed 32-bit input and **O(1)** space complexity, efficiently reversing the bits.

* Missing Number

Commented Code

class Solution {

*// Method to find the missing number in an array of 0 to n*

public int missingNumber(int[] nums) {

int n = nums.length; *// Length of the array*

*// Calculate expected sum of numbers from 0 to n*

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

*// Calculate actual sum of numbers in the array*

int actualSum = 0;

for (int num : nums) {

actualSum += num;

}

*// Return the missing number (difference between expected and actual sum)*

return expectedSum - actualSum;

}

}

**Time Complexity**

* **Time Complexity: O(n)**
  + Calculating n = nums.length is O(1).
  + Computing the expected sum n \* (n + 1) / 2 is O(1).
  + Iterating through the array to compute actualSum takes O(n), where n is the length of the array, with each iteration performing an O(1) addition.
  + The final subtraction (expectedSum - actualSum) is O(1).
  + Therefore, the total time complexity is **O(n)** due to the single pass through the array.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses three integer variables (n, expectedSum, actualSum), requiring O(1) space.
  + The input array nums is not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution finds the missing number in an array containing n distinct numbers from 0 to n. It calculates the expected sum of numbers from 0 to n using the formula n \* (n + 1) / 2, computes the actual sum of the array elements, and returns their difference as the missing number. The approach achieves **O(n)** time complexity for a single pass through the array and **O(1)** space complexity, efficiently identifying the missing number.

* Sum of Two Integers

Commented Code

public class Solution {

*// Method to calculate the sum of two integers without using + or -*

public int getSum(int a, int b) {

while (b != 0) { *// Continue until there is no carry*

int carry = (a & b) << 1; *// Calculate carry (bits that are 1 in both a and b, shifted left)*

a ^= b; *// Sum without carry (XOR simulates addition without carry)*

b = carry; *// Update b to carry for next iteration*

}

return a; *// Return final sum*

}

}

**Time Complexity**

* **Time Complexity: O(log n)**
  + The algorithm iterates until b (the carry) becomes 0.
  + Each iteration performs:
    - A bitwise AND (a & b) and left shift (<< 1) to compute the carry, both O(1).
    - An XOR (a ^ b) to compute the sum without carry, O(1).
    - An assignment to update b, O(1).
  + The number of iterations depends on the number of bits in the carry. In the worst case, the carry propagates through all 32 bits of a 32-bit integer (e.g., adding large numbers like 0x7FFFFFFF and 1). Each iteration shifts the carry left, reducing the number of significant bits logarithmically.
  + Therefore, the time complexity is **O(log n)**, where n is the maximum absolute value of the input integers (bounded by 32 bits in practice, but logarithmic for general analysis).

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses only one integer variable (carry) in each iteration, requiring O(1) space.
  + The input integers a and b are modified in place and not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution calculates the sum of two integers without using the + or - operators by simulating binary addition with bitwise operations. It uses XOR (^) to compute the sum without carry and AND (&) with a left shift (<< 1) to compute the carry. The carry is repeatedly added to the sum until no carry remains. The approach achieves **O(log n)** time complexity, where n is the maximum absolute value of the inputs, and **O(1)** space complexity, efficiently computing the sum.

* Reverse Integer

Commented Code

class Solution {

*// Method to reverse the digits of a 32-bit signed integer*

public int reverse(int x) {

int rev = 0; *// Initialize reversed number*

*// Process each digit of x*

while (x != 0) {

*// Extract the last digit*

int pop = x % 10;

x /= 10; *// Remove the last digit*

*// Check for overflow before updating rev*

if (rev > Integer.MAX\_VALUE / 10 || (rev == Integer.MAX\_VALUE / 10 && pop > 7)) {

return 0; *// Return 0 for overflow*

}

if (rev < Integer.MIN\_VALUE / 10 || (rev == Integer.MIN\_VALUE / 10 && pop < -8)) {

return 0; *// Return 0 for underflow*

}

*// Update reversed number by adding the digit*

rev = rev \* 10 + pop;

}

return rev; *// Return the reversed number*

}

}

**Time Complexity**

* **Time Complexity: O(log |x|)**
  + The algorithm processes each digit of the input number x. The number of digits is proportional to log |x| (base 10), as a number x has approximately log10(|x|) + 1 digits.
  + Each iteration performs O(1) operations: modulo (%), division (/), comparisons for overflow/underflow, and updating rev.
  + For a 32-bit integer, the maximum number of digits is 10 (e.g., 2^31 - 1 = 2147483647), so the complexity is bounded by a constant, but generally expressed as **O(log |x|)**.

**Space Complexity**

* **Space Complexity: O(1)**
  + The algorithm uses two integer variables (rev and pop), requiring O(1) space.
  + The input x is modified in place and not counted as extra space.
  + No additional data structures are used.
  + Therefore, the space complexity is **O(1)**.

**Concise Summary of the Approach**

The solution reverses the digits of a 32-bit signed integer by extracting the last digit using modulo (x % 10), removing it with division (x /= 10), and building the reversed number (rev \* 10 + pop). It checks for overflow/underflow before adding each digit, returning 0 if the result would exceed 32-bit integer limits ([-2^31, 2^31 - 1]). The approach achieves **O(log |x|)** time complexity for processing digits and **O(1)** space complexity, efficiently reversing the integer while handling overflow.

**SORTINGS**

🔹 1. **Bubble Sort**

📘 Notes

* Simple algorithm → repeatedly compare adjacent elements and swap if needed.
* Largest element “bubbles up” to the end in each pass.
* Not practical for large input but good for learning basics.

class BubbleSort {

public static void bubbleSort(int[] arr) {

int n = arr.length;

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

boolean swapped = false;

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break; // optimization

}

}

}

⏱ Time: Best O(n), Avg/Worst O(n²) | 🧮 Space: O(1) | ✅ Stable

🔹 2. **Selection Sort**

📘 Notes

* Repeatedly select the minimum element from unsorted part → place in front.
* Fewer swaps than bubble, but still O(n²).
* Not stable (swapping breaks order).

class SelectionSort {

public static void selectionSort(int[] arr) {

int n = arr.length;

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

int minIndex = i;

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

if (arr[j] < arr[minIndex]) minIndex = j;

}

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

}

⏱ Time: Always O(n²) | 🧮 Space: O(1) | ❌ Not stable

🔹 3. **Insertion Sort**

📘 Notes

* Works like sorting playing cards in hand.
* Insert each element in the correct position of the sorted left side.
* Efficient for small arrays or nearly sorted data.

class InsertionSort {

public static void insertionSort(int[] arr) {

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

}

⏱ Time: Best O(n), Worst O(n²) | 🧮 Space: O(1) | ✅ Stable

🔹 4. **Merge Sort**

📘 Notes

* Divide & Conquer → split array into halves → sort recursively → merge.
* Always O(n log n), very reliable.
* Requires extra space for merging.
* Stable → maintains order of equal elements.

class MergeSort {

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

private static void merge(int[] arr, int left, int mid, int right) {

int n1 = mid - left + 1, n2 = right - mid;

int[] L = new int[n1], R = new int[n2];

for (int i = 0; i < n1; i++) L[i] = arr[left + i];

for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

}

⏱ Time: Always O(n log n) | 🧮 Space: O(n) | ✅ Stable

🔹 5. **Quick Sort**

📘 Notes

* Divide & Conquer → choose pivot, partition elements, recursively sort.
* Average O(n log n), but worst-case O(n²) (bad pivot).
* In-place → low space usage.
* Not stable.

class QuickSort {

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high], i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;

}

}

int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;

return i + 1;

}

}

⏱ Time: Best/Avg O(n log n), Worst O(n²) | 🧮 Space: O(log n) | ❌ Not stable

**Templates**

class AlgoTemplates {

// ---------------- BFS ----------------

public void bfs(int start, Map<Integer, List<Integer>> graph) {

Queue<Integer> queue = new LinkedList<>();

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

queue.offer(start);

visited.add(start);

while (!queue.isEmpty()) {

int node = queue.poll();

// process node

for (int neighbor : graph.getOrDefault(node, new ArrayList<>())) {

if (!visited.contains(neighbor)) {

visited.add(neighbor);

queue.offer(neighbor);

}

}

}

}

// ---------------- DFS (Recursive) ----------------

public void dfsRecursive(int node, Map<Integer, List<Integer>> graph, Set<Integer> visited) {

if (visited.contains(node)) return;

visited.add(node);

// process node

for (int neighbor : graph.getOrDefault(node, new ArrayList<>())) {

dfsRecursive(neighbor, graph, visited);

}

}

// ---------------- DFS (Iterative) ----------------

public void dfsIterative(int start, Map<Integer, List<Integer>> graph) {

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

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

stack.push(start);

while (!stack.isEmpty()) {

int node = stack.pop();

if (visited.contains(node)) continue;

visited.add(node);

// process node

for (int neighbor : graph.getOrDefault(node, new ArrayList<>())) {

if (!visited.contains(neighbor)) {

stack.push(neighbor);

}

}

}

}

// ---------------- Greedy Template ----------------

public void greedyExample(int[] nums) {

Arrays.sort(nums); // often sort first

int result = 0;

for (int num : nums) {

// make the greedy choice

result += num;

}

// return or process result

}

// ---------------- DP (Memoization) ----------------

public int fibMemo(int n, Map<Integer, Integer> memo) {

if (n <= 1) return n;

if (memo.containsKey(n)) return memo.get(n);

int val = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);

memo.put(n, val);

return val;

}

// ---------------- DP (Tabulation) ----------------

public int fibTab(int n) {

if (n <= 1) return n;

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

dp[0] = 0; dp[1] = 1;

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

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

// ---------------- Knapsack (0/1) ----------------

public int knapsack(int[] weights, int[] values, int W) {

int n = weights.length;

int[][] dp = new int[n + 1][W + 1];

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

for (int w = 0; w <= W; w++) {

if (weights[i - 1] <= w) {

dp[i][w] = Math.max(dp[i - 1][w],

values[i - 1] + dp[i - 1][w - weights[i - 1]]);

} else {

dp[i][w] = dp[i - 1][w];

}

}

}

return dp[n][W];

}

// ---------------- Kadane’s Algorithm ----------------

public int kadane(int[] nums) {

int maxSoFar = nums[0], currMax = nums[0];

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

currMax = Math.max(nums[i], currMax + nums[i]);

maxSoFar = Math.max(maxSoFar, currMax);

}

return maxSoFar;

}

// ---------------- Topological Sort (Kahn’s Algorithm) ----------------

public List<Integer> topologicalSort(int n, Map<Integer, List<Integer>> graph) {

int[] indegree = new int[n];

for (int u : graph.keySet()) {

for (int v : graph.get(u)) {

indegree[v]++;

}

}

Queue<Integer> queue = new LinkedList<>();

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

if (indegree[i] == 0) queue.offer(i);

}

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

while (!queue.isEmpty()) {

int u = queue.poll();

result.add(u);

for (int v : graph.getOrDefault(u, new ArrayList<>())) {

indegree[v]--;

if (indegree[v] == 0) queue.offer(v);

}

}

return result; // empty if cycle exists

}

// ---------------- Dijkstra’s Algorithm ----------------

public int[] dijkstra(int n, Map<Integer, List<int[]>> graph, int src) {

int[] dist = new int[n];

Arrays.fill(dist, Integer.MAX\_VALUE);

dist[src] = 0;

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));

pq.offer(new int[]{src, 0});

while (!pq.isEmpty()) {

int[] curr = pq.poll();

int u = curr[0], d = curr[1];

if (d > dist[u]) continue;

for (int[] edge : graph.getOrDefault(u, new ArrayList<>())) {

int v = edge[0], w = edge[1];

if (dist[u] + w < dist[v]) {

dist[v] = dist[u] + w;

pq.offer(new int[]{v, dist[v]});

}

}

}

return dist;

}

// ---------------- Bellman-Ford Algorithm ----------------

public int[] bellmanFord(int n, int[][] edges, int src) {

int[] dist = new int[n];

Arrays.fill(dist, Integer.MAX\_VALUE);

dist[src] = 0;

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

for (int[] e : edges) {

int u = e[0], v = e[1], w = e[2];

if (dist[u] != Integer.MAX\_VALUE && dist[u] + w < dist[v]) {

dist[v] = dist[u] + w;

}

}

}

return dist; // if negative cycle: run one more iteration to detect

}

// ---------------- Floyd-Warshall Algorithm ----------------

public int[][] floydWarshall(int n, int[][] graph) {

int[][] dist = new int[n][n];

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

dist[i] = Arrays.copyOf(graph[i], n);

}

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

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

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

if (dist[i][k] != Integer.MAX\_VALUE && dist[k][j] != Integer.MAX\_VALUE) {

dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);

}

}

}

}

return dist;

}

// ---------------- Trie (Prefix Tree) ----------------

class TrieNode {

Map<Character, TrieNode> children = new HashMap<>();

boolean isWord;

}

class Trie {

private final TrieNode root = new TrieNode();

public void insert(String word) {

TrieNode node = root;

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

node.children.putIfAbsent(c, new TrieNode());

node = node.children.get(c);

}

node.isWord = true;

}

public boolean search(String word) {

TrieNode node = root;

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

if (!node.children.containsKey(c)) return false;

node = node.children.get(c);

}

return node.isWord;

}

public boolean startsWith(String prefix) {

TrieNode node = root;

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

if (!node.children.containsKey(c)) return false;

node = node.children.get(c);

}

return true;

}

}

}