**Category: Array and String Manipulation**

**Problem 3: Two Sum (LeetCode 1)**

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

**Solution**:

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

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

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

int complement = target - nums[i];

if (map.containsKey(complement)) {

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

}

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

}

throw new IllegalArgumentException("No two sum solution");

}

**Explanation**:

* Use a hashmap to store the number and its index.
* Check if the complement (target - nums[i]) exists in the map.

**Time Complexity**: O(n), where n is the length of the array.  
**Space Complexity**: O(n) (hashmap space).

**Problem 1: Valid Anagram (LeetCode 242)**

**Description**: Given two strings s and t, return true if t is an anagram of s, otherwise return false.

**Solution**:

public boolean isAnagram(String s, String t) {

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

int[] count = new int[26];

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

count[c - 'a']++;

}

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

count[c - 'a']--;

if (count[c - 'a'] < 0) return false;

}

return true;

}

**Explanation**:

* Use a frequency array of size 26 (for lowercase English letters).
* Increment counts for characters in s and decrement for characters in t.
* If any count goes negative, t is not an anagram of s.

**Time Complexity**: O(n), where n is the length of the strings.  
**Space Complexity**: O(1) (fixed-size array).

**Or**

public boolean isAnagram(String s, String t) {

// If lengths are different, they cannot be anagrams

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

return false;

}

// Convert both strings to character arrays, sort them, and compare

char[] sArray = s.toCharArray();

char[] tArray = t.toCharArray();

Arrays.sort(sArray);

Arrays.sort(tArray);

return Arrays.equals(sArray, tArray);

}

**Time Complexity:** O(n log n), where n is the length of the strings

**Space Complexity:** O(n) or O(1), depending on how you interpret the problem

**Problem 4: First Unique Character in a String (LeetCode 387)**

**Description**: Given a string s, find the first non-repeating character and return its index. If it doesn't exist, return -1.

**Solution**:

public int firstUniqChar(String s) {

int[] count = new int[26];

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

count[c - 'a']++;

}

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

if (count[s.charAt(i) - 'a'] == 1) {

return i;

}

}

return -1;

}

**Explanation**:

* Use a frequency array to count occurrences of each character.
* Iterate through the string to find the first character with a count of 1.

**Time Complexity**: O(n), where n is the length of the string.  
**Space Complexity**: O(1) (fixed-size array).

**Problem 8: String Compression (LeetCode 443)**

**Description**: Compress a string by replacing consecutive repeating characters with the character followed by the count.

**Solution**:

public int compress(char[] chars) {

int write = 0, read = 0;

while (read < chars.length) {

char currentChar = chars[read];

int count = 0;

while (read < chars.length && chars[read] == currentChar) {

count++;

read++;

}

chars[write++] = currentChar;

if (count > 1) {

for (char digit : String.valueOf(count).toCharArray()) {

chars[write++] = digit;

}

}

}

return write;

}

**Explanation**:

* Use two pointers: one for reading and one for writing.
* Append the count if it’s greater than 1.

**Time Complexity**: O(n), where n is the length of the array.  
**Space Complexity**: O(1) (in-place modification).

**Problem 10: Move Zeroes (LeetCode 283)**

**Description**: Move all 0's to the end of an array while maintaining the relative order of non-zero elements.

**Solution**:

public void moveZeroes(int[] nums) {

int index = 0;

for (int num : nums) {

if (num != 0) {

nums[index++] = num;

}

}

while (index < nums.length) {

nums[index++] = 0;

}

}

**Explanation**:

* Use two pointers: one for writing non-zero elements and another for filling zeros.

**Time Complexity**: O(n), where n is the length of the array.  
**Space Complexity**: O(1) (in-place modification).

**Problem 11: Best Time to Buy and Sell Stock (LeetCode 121)**

**Description**: Given an array prices where prices[i] is the price of a stock on the i-th day, find the maximum profit you can achieve by buying on one day and selling on a later day.

**Solution**:

java

Copy

public int maxProfit(int[] prices) {

int minPrice = Integer.MAX\_VALUE;

int maxProfit = 0;

for (int price : prices) {

if (price < minPrice) {

minPrice = price;

} else if (price - minPrice > maxProfit) {

maxProfit = price - minPrice;

}

}

return maxProfit;

}

**Explanation**:

* Track the minimum price encountered so far.
* Calculate the profit if selling at the current price and update the maximum profit.

**Time Complexity**: O(n), where n is the length of the array.  
**Space Complexity**: O(1).

**Problem 15: Shortest Distance to a Character (LeetCode 821)**

**Description**: Given a string s and a character c, return an array of integers representing the shortest distance from each character in s to c.

**Solution**:

public int[] shortestToChar(String s, char c) {

int n = s.length();

int[] result = new int[n];

int prev = -n;

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

if (s.charAt(i) == c) prev = i;

result[i] = i - prev;

}

prev = 2 \* n;

for (int i = n - 1; i >= 0; i--) {

if (s.charAt(i) == c) prev = i;

result[i] = Math.min(result[i], prev - i);

}

return result;

}

**Explanation**:

* Perform two passes: one from left to right and another from right to left.
* Track the position of the target character and calculate distances.

**Time Complexity**: O(n), where n is the length of the string.  
**Space Complexity**: O(n) (output array).

**Problem 47: Reverse String (LeetCode 344)**

**Description**: Reverse a string in-place.

**Solution**:

java

Copy

public void reverseString(char[] s) {

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

while (left < right) {

char temp = s[left];

s[left++] = s[right];

s[right--] = temp;

}

}

**Explanation**:

* Use two pointers to swap characters from the start and end.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 13: Reverse Integer (LeetCode 7)**

**Description**: Given a 32-bit signed integer, reverse its digits. If reversing causes overflow, return 0.

**Solution**:

java

Copy

public int reverse(int x) {

int reversed = 0;

while (x != 0) {

int digit = x % 10;

x /= 10;

if (reversed > Integer.MAX\_VALUE / 10 || (reversed == Integer.MAX\_VALUE / 10 && digit > 7)) return 0;

if (reversed < Integer.MIN\_VALUE / 10 || (reversed == Integer.MIN\_VALUE / 10 && digit < -8)) return 0;

reversed = reversed \* 10 + digit;

}

return reversed;

}

**Explanation**:

* Reverse the integer digit by digit.
* Handle overflow by checking bounds before updating the reversed number.

**Time Complexity**: O(log n), where n is the integer.  
**Space Complexity**: O(1).

**Problem 34: Palindrome Number (LeetCode 9)**

**Description**: Determine if an integer is a palindrome.

**Solution**:

java

Copy

public boolean isPalindrome(int x) {

if (x < 0 || (x % 10 == 0 && x != 0)) return false;

int reversed = 0;

while (x > reversed) {

reversed = reversed \* 10 + x % 10;

x /= 10;

}

return x == reversed || x == reversed / 10;

}

**Explanation**:

* Reverse the second half of the number and compare it with the first half.

**Time Complexity**: O(log n).  
**Space Complexity**: O(1).

**Problem 52: Implement strStr() (LeetCode 28)**

**Description**: Return the index of the first occurrence of needle in haystack, or -1 if not found.

**Solution**:

java

Copy

public int strStr(String haystack, String needle) {

if (needle.isEmpty()) return 0;

for (int i = 0; i <= haystack.length() - needle.length(); i++) {

if (haystack.substring(i, i + needle.length()).equals(needle)) {

return i;

}

}

return -1;

}

**Explanation**:

* Use a sliding window to check for the substring.

**Time Complexity**: O(n \* m), where n is the length of haystack and m is the length of needle.  
**Space Complexity**: O(1).

**Problem 88: Longest Substring Without Repeating Characters (LeetCode 3)**

**Description**: Find the length of the longest substring without repeating characters.

**Solution**:

java

Copy

public int lengthOfLongestSubstring(String s) {

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

int left = 0, maxLen = 0;

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

char c = s.charAt(right);

if (map.containsKey(c)) {

left = Math.max(left, map.get(c) + 1);

}

map.put(c, right);

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

}

return maxLen;

}

**Explanation**:

* Use a sliding window with two pointers to track the longest substring.
* Use a hashmap to store the last index of each character.

**Time Complexity**: O(n).  
**Space Complexity**: O(min(m, n)), where m is the character set size.

Problem 100: Subarray Sum Equals K (LeetCode 560)

**Description**: Find the total number of continuous subarrays whose sum equals k.

**Solution**:

java

Copy

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

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

map.put(0, 1);

int sum = 0, count = 0;

for (int num : nums) {

sum += num;

if (map.containsKey(sum - k)) {

count += map.get(sum - k);

}

map.put(sum, map.getOrDefault(sum, 0) + 1);

}

return count;

}

**Explanation**:

* Use a hashmap to store prefix sums and their frequencies.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem** 10: Minimum Size Subarray Sum (Leetcode 209)

**Description**: Given an array of positive integers nums and a positive integer target, return the minimal length of a contiguous subarray whose sum is greater than or equal to target. If there is no such subarray, return

Solution :

*public class Solution {*

*public int minSubArrayLen(int target, int[] nums) {*

*int left = 0; // Left pointer of the sliding window*

*int currentSum = 0; // Sum of the current window*

*int minLength = Integer.MAX\_VALUE; // Initialize with maximum possible value*

*// Iterate through the array using the right pointer*

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

*currentSum += nums[right]; // Expand the window by adding the current element*

*// Shrink the window from the left if the current sum is >= target*

*while (currentSum >= target) {*

*minLength = Math.min(minLength, right - left + 1); // Update the minimum length*

*currentSum -= nums[left]; // Remove the leftmost element from the window*

*left++; // Move the left pointer to the right*

*}*

*}*

*// If no valid subarray is found, return 0; otherwise, return the minimum length*

*return minLength == Integer.MAX\_VALUE ? 0 : minLength;*

*}*

*}*

**Time Complexity:** **O(n)**

* + Each element is visited at most twice (once by the right pointer and once by the left pointer), so the algorithm runs in linear time.

**Space Complexity:** **O(1)**

* + Only a constant amount of extra space is used for variables like left, right, currentSum, and minLength.

**Problem 31: Shortest Unsorted Continuous Subarray (LeetCode 581)**

**Description**: Find the shortest continuous subarray that, if sorted, makes the entire array sorted.

**Solution**:

java

Copy

public int findUnsortedSubarray(int[] nums) {

int n = nums.length;

int start = -1, end = -2;

int min = nums[n - 1], max = nums[0];

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

max = Math.max(max, nums[i]);

min = Math.min(min, nums[n - 1 - i]);

if (nums[i] < max) end = i;

if (nums[n - 1 - i] > min) start = n - 1 - i;

}

return end - start + 1;

}

**Explanation**:

* Traverse the array to find the start and end of the unsorted subarray.
* Track the minimum and maximum values to determine the boundaries.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 51: Rotate Array (LeetCode 189)**

**Description**: Rotate an array to the right by k steps.

**Solution**:

java

Copy

public void rotate(int[] nums, int k) {

k %= nums.length;

reverse(nums, 0, nums.length - 1);

reverse(nums, 0, k - 1);

reverse(nums, k, nums.length - 1);

}

private void reverse(int[] nums, int start, int end) {

while (start < end) {

int temp = nums[start];

nums[start++] = nums[end];

nums[end--] = temp;

}

}

**Explanation**:

* Reverse the entire array, then reverse the first k elements, and finally reverse the remaining elements.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem: Product of Array Except Self (Leetcode 238)**

**Description:**

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i]. You must write an algorithm that runs in **O(n)** time and **without using the division operation**.

Solution:

*public class Solution {*

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

*int n = nums.length;*

*int[] answer = new int[n];*

*// Step 1: Compute prefix products*

*answer[0] = 1; // There are no elements to the left of the first element*

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

*answer[i] = answer[i - 1] \* nums[i - 1];*

*}*

*// Step 2: Compute suffix products and multiply with prefix products*

*int suffixProduct = 1; // Initialize suffix product*

*for (int i = n - 1; i >= 0; i--) {*

*answer[i] = answer[i] \* suffixProduct; // Multiply prefix and suffix*

*suffixProduct = suffixProduct \* nums[i]; // Update suffix product*

*}*

*return answer;*

*}*

*}*

**Time Complexity:** **O(n)**

* + The algorithm makes two passes through the array: one for prefix products and one for suffix products. Each pass takes **O(n)** time.

**Space Complexity:** **O(1)** (excluding the output array)

* + The algorithm uses only a constant amount of extra space for variables like suffixProduct. The output array answer is not counted toward the space complexity as it is required by the problem.

**Problem 39: Find Pivot Index (LeetCode 724)**

**Description**: Find the pivot index where the sum of numbers to the left equals the sum to the right.

**Solution**:

java

Copy

public int pivotIndex(int[] nums) {

int totalSum = 0;

for (int num : nums) totalSum += num;

int leftSum = 0;

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

if (leftSum == totalSum - leftSum - nums[i]) return i;

leftSum += nums[i];

}

return -1;

}

**Explanation**:

* Calculate the total sum and track the left sum while traversing the array.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem: Remove Duplicates from Sorted Array II (Leetcode 80)**

Description: Given an integer array nums sorted in non-decreasing order, remove some duplicates in-place such that each unique element appears **at most twice**. The relative order of the elements should be kept the same. Return the number of elements after removing the duplicates.

Solution:

public class Solution {

public int removeDuplicates(int[] nums) {

int n = nums.length;

if (n <= 2) {

return n; // No need to process if the array has 2 or fewer elements

}

int k = 2; // Pointer to track the position of the next valid element

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

// If the current element is not a duplicate (more than twice), include it

if (nums[i] != nums[k - 2]) {

nums[k] = nums[i]; // Place the valid element at position k

k++; // Move the pointer forward

}

}

return k; // Return the number of valid elements

}

}

**Time Complexity:** **O(n)**

* + The algorithm makes a single pass through the array, so it runs in linear time.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space for variables like k and i.

**Problem: String to Integer (atoi) (Leetcode 8)**

Description: Implement the atoi function, which converts a string to a 32-bit signed integer. The algorithm should handle the following cases:

Solution:

public class Solution {

public int myAtoi(String s) {

int index = 0;

int n = s.length();

int sign = 1; // 1 for positive, -1 for negative

int result = 0;

// Step 1: Skip leading whitespace

while (index < n && s.charAt(index) == ' ') {

index++;

}

// Step 2: Handle optional sign

if (index < n && (s.charAt(index) == '+' || s.charAt(index) == '-')) {

sign = (s.charAt(index) == '-') ? -1 : 1;

index++;

}

// Step 3: Read numerical digits

while (index < n && Character.isDigit(s.charAt(index))) {

int digit = s.charAt(index) - '0';

// Step 4: Handle overflow/underflow

if (result > Integer.MAX\_VALUE / 10 ||

(result == Integer.MAX\_VALUE / 10 && digit > Integer.MAX\_VALUE % 10)) {

return (sign == 1) ? Integer.MAX\_VALUE : Integer.MIN\_VALUE;

}

result = result \* 10 + digit;

index++;

}

// Step 5: Return the result with the appropriate sign

return result \* sign;

}

}

**Time Complexity:** **O(n)**

* + The algorithm processes each character of the string at most once.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem: Integer to Roman**

**Description:** Given an integer, convert it to a Roman numeral. Roman numerals are represented by combinations of letters from the Latin alphabet

**Solution:**

public class Solution {

public String intToRoman(int num) {

// Arrays to store Roman numerals and their corresponding values

int[] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};

String[] symbols = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};

StringBuilder result = new StringBuilder();

// Iterate through the values and symbols

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

// Append the symbol while the current value is <= num

while (num >= values[i]) {

result.append(symbols[i]);

num -= values[i];

}

}

return result.toString();

}

}

**Time Complexity:** **O(1)**

* + The algorithm iterates through a fixed number of values (13 in this case), so the time complexity is constant.

**Space Complexity:** **O(1)**

* + The space used by the StringBuilder is proportional to the length of the Roman numeral, which is bounded by a small constant (the longest Roman numeral for numbers up to 3999 is MMMCMXCIX, which has 15 characters).

**Problem:** Next Permutation (Leetcode 31)

Description: Given an array of integers nums, find the **next lexicographically greater permutation** of its elements. If such an arrangement is not possible, rearrange the array into the **lowest possible order** (i.e., sorted in ascending order).

Solution:

public class Solution {

public void nextPermutation(int[] nums) {

int n = nums.length;

int i = n - 2;

// Step 1: Find the first decreasing element

while (i >= 0 && nums[i] >= nums[i + 1]) {

i--;

}

// Step 2: Find the element to swap with

if (i >= 0) {

int j = n - 1;

while (j >= 0 && nums[j] <= nums[i]) {

j--;

}

swap(nums, i, j);

}

// Step 3: Reverse the suffix

reverse(nums, i + 1, n - 1);

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

private void reverse(int[] nums, int start, int end) {

while (start < end) {

swap(nums, start, end);

start++;

end--;

}

}

}

**Time Complexity:** **O(n)**

* + The algorithm makes at most two passes through the array.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem**: Restore IP Addresses

Description: Given a string s containing only digits, return all possible valid IP addresses that can be formed by inserting dots into s. A valid IP address consists of exactly four integers (each between 0 and 255) separated by dots. Integers cannot have leading zeros unless they are zero itself.

Solution:

import java.util.ArrayList;

import java.util.List;

public class Solution {

public List<String> restoreIpAddresses(String s) {

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

backtrack(s, 0, new ArrayList<>(), result);

return result;

}

private void backtrack(String s, int start, List<String> current, List<String> result) {

// If we have 4 segments and used the entire string, add to result

if (current.size() == 4) {

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

result.add(String.join(".", current));

}

return;

}

// Try all possible segment lengths (1 to 3 digits)

for (int len = 1; len <= 3; len++) {

if (start + len > s.length()) {

break; // Out of bounds

}

String segment = s.substring(start, start + len);

if (isValidSegment(segment)) {

current.add(segment); // Add valid segment

backtrack(s, start + len, current, result); // Recurse

current.remove(current.size() - 1); // Backtrack

}

}

}

private boolean isValidSegment(String segment) {

if (segment.length() > 1 && segment.charAt(0) == '0') {

return false; // No leading zeros

}

int value = Integer.parseInt(segment);

return value >= 0 && value <= 255;

}

}

**Time Complexity:** **O(1)**

* + The number of possible IP addresses is limited (at most 34=8134=81 combinations), so the time complexity is constant.

**Space Complexity:** **O(1)**

* + The space used by the recursion stack and result list is bounded by a small constant.

**Problem**: **ZigZag Conversion (Leetcode 6)**

**Description**: Given a string s and an integer numRows, write the string in a zigzag pattern on numRows rows and then read it row by row to produce the converted string.

Solution:

public class Solution {

public String convert(String s, int numRows) {

if (numRows == 1) {

return s; // No zigzag pattern needed

}

StringBuilder[] rows = new StringBuilder[numRows];

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

rows[i] = new StringBuilder();

}

int currentRow = 0;

boolean goingDown = false;

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

rows[currentRow].append(c);

if (currentRow == 0 || currentRow == numRows - 1) {

goingDown = !goingDown; // Change direction

}

currentRow += goingDown ? 1 : -1;

}

StringBuilder result = new StringBuilder();

for (StringBuilder row : rows) {

result.append(row);

}

return result.toString();

}

}

**Time Complexity:** **O(n)**

* + Each character is visited once.

**Space Complexity:** **O(n)**

* + The space used by the StringBuilder array is proportional to the input string length.

**Problem**: **Letter Combinations of a Phone Number (Leetcode 17)**

**Description**: Given a string containing digits from 2-9, return all possible letter combinations that the number could represent, based on the mapping of digits to letters on a telephone keypad.

**Solution**:

import java.util.ArrayList;

import java.util.List;

public class Solution {

public List<String> letterCombinations(String digits) {

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

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

return result;

}

// Mapping of digits to letters

String[] digitToLetters = {

"", // 0

"", // 1

"abc", // 2

"def", // 3

"ghi", // 4

"jkl", // 5

"mno", // 6

"pqrs", // 7

"tuv", // 8

"wxyz" // 9

};

backtrack(digits, 0, new StringBuilder(), result, digitToLetters);

return result;

}

private void backtrack(String digits, int index, StringBuilder current, List<String> result, String[] digitToLetters) {

// If the current combination is complete, add it to the result

if (index == digits.length()) {

result.add(current.toString());

return;

}

// Get the letters corresponding to the current digit

String letters = digitToLetters[digits.charAt(index) - '0'];

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

current.append(c); // Add the current letter

backtrack(digits, index + 1, current, result, digitToLetters); // Recurse

current.deleteCharAt(current.length() - 1); // Backtrack

}

}

}

**Time Complexity:** **O(4^n)**

* + Each digit can map to up to 4 letters, and there are n digits, so the total number of combinations is 4n4*n*.

**Space Complexity:** **O(n)**

* + The recursion stack depth is proportional to the number of digits (n), and the space used by the result list is O(4n)*O*(4*n*).

**Problem**: Group Anagrams (leetcode 49)

**Description**: Given an array of strings strs, group the anagrams together. An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Solution**:

import java.util.\*;

public class Solution {

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

// Map to store groups of anagrams

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

for (String s : strs) {

// Sort the string to use as a key

char[] chars = s.toCharArray();

Arrays.sort(chars);

String sorted = new String(chars);

// Add the original string to the corresponding list

if (!map.containsKey(sorted)) {

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

}

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

}

// Return the grouped anagrams

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

}

}

**Time Complexity:** **O(n \* k log k)**

* + n is the number of strings, and k is the maximum length of a string.
  + Sorting each string takes O(klog⁡k)*O*(*k*log*k*), and this is done for all n strings.

**Space Complexity:** **O(n \* k)**

* + The map stores all n strings, each of length up to k.

**Problem 44: Count and Say (LeetCode 38)**

**Description**: Generate the nth term of the "count and say" sequence.

**Solution**:

java

Copy

public String countAndSay(int n) {

String result = "1";

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

StringBuilder sb = new StringBuilder();

int count = 1;

for (int j = 1; j < result.length(); j++) {

if (result.charAt(j) == result.charAt(j - 1)) {

count++;

} else {

sb.append(count).append(result.charAt(j - 1));

count = 1;

}

}

sb.append(count).append(result.charAt(result.length() - 1));

result = sb.toString();

}

return result;

}

**Explanation**:

* Build the sequence iteratively by counting and appending characters.

**Time Complexity**: O(n \* m), where m is the length of the sequence.  
**Space Complexity**: O(m).

**Problem**: Reverse Words in a String (Leetcode 151)

**Description**: Given an input string s, reverse the order of the words. A word is defined as a sequence of non-space characters. The input string may contain leading or trailing spaces or multiple spaces between words. The returned string should have the words reversed and separated by a single space.

**Solution**:

public class Solution {

public String reverseWords(String s) {

// Trim leading and trailing spaces

s = s.trim();

// Split into words (handling multiple spaces)

String[] words = s.split("\\s+");

// Reverse the array of words

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

while (left < right) {

String temp = words[left];

words[left] = words[right];

words[right] = temp;

left++;

right--;

}

// Join the words with a single space

return String.join(" ", words);

}

}

**Time Complexity:** **O(n)**

* + Trimming, splitting, reversing, and joining all take linear time relative to the input string length.

**Space Complexity:** **O(n)**

* + The space used by the words array and the final result string is proportional to the input string length.

**Problem 109: Encode and Decode Strings (LeetCode 271)**

**Description**: Encode a list of strings into a single string and decode it back.

**Solution**:

java

Copy

public String encode(List<String> strs) {

StringBuilder sb = new StringBuilder();

for (String s : strs) {

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

}

return sb.toString();

}

public List<String> decode(String s) {

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

int i = 0;

while (i < s.length()) {

int slash = s.indexOf('/', i);

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

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

i = slash + 1 + length;

}

return result;

}

**Explanation**:

* Encode by prefixing each string with its length and a delimiter.
* Decode by reading the length and extracting the string.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 79: Sort Characters By Frequency (LeetCode 451)**

**Description**: Sort characters in a string by their frequency.

**Solution**:

java

Copy

public String frequencySort(String s) {

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

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

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

}

PriorityQueue<Character> pq = new PriorityQueue<>((a, b) -> map.get(b) - map.get(a));

pq.addAll(map.keySet());

StringBuilder result = new StringBuilder();

while (!pq.isEmpty()) {

char c = pq.poll();

int count = map.get(c);

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

result.append(c);

}

}

return result.toString();

}

**Explanation**:

* Use a hashmap to count frequencies and a priority queue to sort characters by frequency.

**Time Complexity**: O(n log n).  
**Space Complexity**: O(n).

**Category:** Stack and Queue

**Problem 2: Valid Parentheses (LeetCode 20)**

**Description**: Given a string s containing just the characters (, ), {, }, [, and ], determine if the input string is valid.

**Solution**:

public boolean isValid(String s) {

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

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

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;

}

}

}

return stack.isEmpty();

}

**Explanation**:

* Use a stack to track opening brackets.
* For closing brackets, check if the top of the stack matches the corresponding opening bracket.

**Time Complexity**: O(n), where n is the length of the string.  
**Space Complexity**: O(n) (stack space).

**Problem 5: Min Stack (LeetCode 155)**

**Description**: Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

**Solution**:

java

Copy

class MinStack {

private Stack<Integer> stack;

private Stack<Integer> minStack;

public MinStack() {

stack = new Stack<>();

minStack = new Stack<>();

}

public void push(int val) {

stack.push(val);

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

minStack.push(val);

}

}

public void pop() {

if (stack.pop().equals(minStack.peek())) {

minStack.pop();

}

}

public int top() {

return stack.peek();

}

public int getMin() {

return minStack.peek();

}

}

**Explanation**:

* Use two stacks: one for storing values and another for tracking the minimum value.

**Time Complexity**: O(1) for all operations.  
**Space Complexity**: O(n) (stack space).

**Problem 6: Moving Average from Data Stream (LeetCode 346)**

**Description**: Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

**Solution**:

java

Copy

class MovingAverage {

private Queue<Integer> queue;

private int size;

private double sum;

public MovingAverage(int size) {

this.queue = new LinkedList<>();

this.size = size;

this.sum = 0;

}

public double next(int val) {

if (queue.size() == size) {

sum -= queue.poll();

}

queue.offer(val);

sum += val;

return sum / queue.size();

}

}

**Explanation**:

* Use a queue to maintain the sliding window.
* Update the sum by removing the oldest element and adding the new one.

**Time Complexity**: O(1) for each next call.  
**Space Complexity**: O(size) (queue space).

**Problem 23: Next Greater Element I (LeetCode 496)**

**Description**: Given two arrays nums1 and nums2, find the next greater element for each element in nums1 in nums2.

**Solution**:

java

Copy

public int[] nextGreaterElement(int[] nums1, int[] nums2) {

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

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

for (int num : nums2) {

while (!stack.isEmpty() && stack.peek() < num) {

map.put(stack.pop(), num);

}

stack.push(num);

}

int[] result = new int[nums1.length];

for (int i = 0; i < nums1.length; i++) {

result[i] = map.getOrDefault(nums1[i], -1);

}

return result;

}

**Explanation**:

* Use a stack to find the next greater element for each element in nums2.
* Use a hashmap to store the results.

**Time Complexity**: O(n + m), where n is the length of nums1 and m is the length of nums2.  
**Space Complexity**: O(m).

**Problem** 97: Daily Temperatures (LeetCode 739)

**Description**: Given an array of daily temperatures, return an array of how many days you have to wait for a warmer temperature.

**Solution**:

java

Copy

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

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

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

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

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

int index = stack.pop();

result[index] = i - index;

}

stack.push(i);

}

return result;

}

**Explanation**:

* Use a stack to keep track of indices of temperatures.
* Pop from the stack when a warmer temperature is found.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 78: Decode String (LeetCode 394)**

**Description**: Decode a string encoded with a pattern like k[encoded\_string].

**Solution**:

java

Copy

public String decodeString(String s) {

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

Stack<String> stringStack = new Stack<>();

String currentString = "";

int k = 0;

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

if (Character.isDigit(c)) {

k = k \* 10 + (c - '0');

} else if (c == '[') {

countStack.push(k);

stringStack.push(currentString);

currentString = "";

k = 0;

} else if (c == ']') {

StringBuilder decodedString = new StringBuilder(stringStack.pop());

int count = countStack.pop();

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

decodedString.append(currentString);

}

currentString = decodedString.toString();

} else {

currentString += c;

}

}

return currentString;

}

**Explanation**:

* Use stacks to track counts and strings.
* Build the decoded string by repeating the substring.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 58: Remove Invalid Parentheses (LeetCode 301)**

**Description**: Remove the minimum number of invalid parentheses to make the input string valid.

**Solution**:

java

Copy

public List<String> removeInvalidParentheses(String s) {

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

remove(s, result, 0, 0, '(', ')');

return result;

}

private void remove(String s, List<String> result, int lastI, int lastJ, char open, char close) {

int balance = 0;

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

if (s.charAt(i) == open) balance++;

if (s.charAt(i) == close) balance--;

if (balance >= 0) continue;

for (int j = lastJ; j <= i; j++) {

if (s.charAt(j) == close && (j == lastJ || s.charAt(j - 1) != close)) {

remove(s.substring(0, j) + s.substring(j + 1)), result, i, j, open, close);

}

}

return;

}

String reversed = new StringBuilder(s).reverse().toString();

if (open == '(') {

remove(reversed, result, 0, 0, ')', '(');

} else {

result.add(reversed);

}

}

**Explanation**:

* Use recursion to remove invalid parentheses and backtrack to find all valid combinations.

**Time Complexity**: O(2^n).  
**Space Complexity**: O(n).

**Problem**: Valid Parenthesis String (leetcode 678)

**Description**: Given a string s containing only three types of characters: '(', ')', and '\*', determine if the string is valid. The '\*' can be treated as

**Solution**:

public class Solution {

public boolean checkValidString(String s) {

int minOpen = 0; // Minimum number of open parentheses

int maxOpen = 0; // Maximum number of open parentheses

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

if (c == '(') {

minOpen++;

maxOpen++;

} else if (c == ')') {

minOpen--;

maxOpen--;

} else if (c == '\*') {

minOpen--; // Treat '\*' as ')'

maxOpen++; // Treat '\*' as '('

}

// If maxOpen < 0, the string is invalid

if (maxOpen < 0) {

return false;

}

// Ensure minOpen is not negative

minOpen = Math.max(minOpen, 0);

}

// Check if all parentheses are balanced

return minOpen == 0;

}

}

**Time Complexity:** **O(n)**

* + The algorithm processes each character in the string once.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem 125: Flatten Nested List Iterator (LeetCode 341)**

**Description**: Implement an iterator to flatten a nested list.

**Solution**:

java

Copy

public class NestedIterator implements Iterator<Integer> {

private Stack<NestedInteger> stack = new Stack<>();

public NestedIterator(List<NestedInteger> nestedList) {

for (int i = nestedList.size() - 1; i >= 0; i--) {

stack.push(nestedList.get(i));

}

}

@Override

public Integer next() {

return stack.pop().getInteger();

}

@Override

public boolean hasNext() {

while (!stack.isEmpty()) {

NestedInteger current = stack.peek();

if (current.isInteger()) return true;

stack.pop();

List<NestedInteger> list = current.getList();

for (int i = list.size() - 1; i >= 0; i--) {

stack.push(list.get(i));

}

}

return false;

}

}

**Explanation**:

* Use a stack to flatten the nested list iteratively.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 70: Maximum Frequency Stack (LeetCode 895)**

**Description**: Design a stack-like data structure to push and pop elements based on their frequency.

**Solution**:

java

Copy

class FreqStack {

private Map<Integer, Integer> frequency;

private Map<Integer, Stack<Integer>> group;

private int maxFrequency;

public FreqStack() {

frequency = new HashMap<>();

group = new HashMap<>();

maxFrequency = 0;

}

public void push(int x) {

int f = frequency.getOrDefault(x, 0) + 1;

frequency.put(x, f);

if (f > maxFrequency) maxFrequency = f;

group.computeIfAbsent(f, z -> new Stack<>()).push(x);

}

public int pop() {

int x = group.get(maxFrequency).pop();

frequency.put(x, frequency.get(x) - 1);

if (group.get(maxFrequency).isEmpty()) maxFrequency--;

return x;

}

}

**Explanation**:

* Use two hashmaps to track frequencies and group elements by frequency.

**Time Complexity**: O(1) for both push and pop.  
**Space Complexity**: O(n).

**Category: Linked List**

**Problem 21: Reverse Linked List (LeetCode 206)**

**Description**: Reverse a singly linked list.

**Solution**:

java

Copy

public ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode current = head;

while (current != null) {

ListNode next = current.next;

current.next = prev;

prev = current;

current = next;

}

return prev;

}

**Explanation**:

* Use three pointers: prev, current, and next.
* Reverse the direction of each node's next pointer.

**Time Complexity**: O(n), where n is the number of nodes.  
**Space Complexity**: O(1).

**Problem 12: Remove Linked List Elements (LeetCode 203)**

**Description**: Given the head of a linked list and an integer val, remove all nodes with value val.

**Solution**:

java

Copy

public ListNode removeElements(ListNode head, int val) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode current = dummy;

while (current.next != null) {

if (current.next.val == val) {

current.next = current.next.next;

} else {

current = current.next;

}

}

return dummy.next;

}

**Explanation**:

* Use a dummy node to handle edge cases (e.g., head node being removed).
* Traverse the list and skip nodes with the target value.

**Time Complexity**: O(n), where n is the number of nodes.  
**Space Complexity**: O(1).

**Problem 18: Merge Two Sorted Lists (LeetCode 21)**

**Description**: Merge two sorted linked lists into one sorted linked list.

**Solution**:

java

Copy

public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(0);

ListNode current = dummy;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

current.next = l1;

l1 = l1.next;

} else {

current.next = l2;

l2 = l2.next;

}

current = current.next;

}

current.next = (l1 != null) ? l1 : l2;

return dummy.next;

}

**Explanation**:

* Use a dummy node to simplify the merging process.
* Append the remaining nodes from either list.

**Time Complexity**: O(m + n), where m and n are the lengths of the lists.  
**Space Complexity**: O(1).

**Problem 38: Palindrome Linked List (LeetCode 234)**

**Description**: Determine if a linked list is a palindrome.

**Solution**:

java

Copy

public boolean isPalindrome(ListNode head) {

ListNode slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

ListNode reversed = reverse(slow);

while (reversed != null) {

if (head.val != reversed.val) return false;

head = head.next;

reversed = reversed.next;

}

return true;

}

private ListNode reverse(ListNode head) {

ListNode prev = null;

while (head != null) {

ListNode next = head.next;

head.next = prev;

prev = head;

head = next;

}

return prev;

}

Or   
public boolean isPalindrome(ListNode head) {

        // Step 1: Convert linked list to a string or array

        StringBuilder sb = new StringBuilder();

        ListNode current = head;

        while (current != null) {

            sb.append(current.val);

            current = current.next;

        }

        // Step 2: Check if it's a palindrome

        String str = sb.toString();

        int left = 0;

        int right = str.length() - 1;

        while (left < right) {

            if (str.charAt(left) != str.charAt(right)) {

                return false;

            }

            left++;

            right--;

        }

        return true;

    }

**Explanation**:

* Use fast and slow pointers to find the middle of the list.
* Reverse the second half and compare it with the first half.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 50: Intersection of Two Linked Lists (LeetCode 160)**

**Description**: Find the node where two linked lists intersect.

**Solution**:

java

Copy

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

ListNode a = headA, b = headB;

while (a != b) {

a = (a == null) ? headB : a.next;

b = (b == null) ? headA : b.next;

}

return a;

}

**Explanation**:

* Use two pointers to traverse both lists until they meet.

**Time Complexity**: O(n + m).  
**Space Complexity**: O(1).

**Problem 53: Linked List Cycle (LeetCode 141)**

**Description**: Determine if a linked list has a cycle.

**Solution**:

java

Copy

public boolean hasCycle(ListNode head) {

ListNode slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

if (slow == fast) return true;

}

return false;

}

**Explanation**:

* Use two pointers (slow and fast) to detect a cycle.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem: Remove Nth Node From End of List (Leetcode 19)**

**Description:** Given the head of a linked list, remove the nth node from the end of the list and return the modified list.

**Solution:**

public class Solution {

public ListNode removeNthFromEnd(ListNode head, int n) {

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

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode fast = dummy;

ListNode slow = dummy;

// Move fast pointer n steps ahead

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

fast = fast.next;

}

// Move both pointers until fast reaches the end

while (fast != null) {

fast = fast.next;

slow = slow.next;

}

// Remove the nth node from the end

slow.next = slow.next.next;

return dummy.next;

}

}

**Time Complexity: O(n)**

* + **The algorithm makes a single pass through the list.**

**Space Complexity: O(1)**

* + **The algorithm uses only a constant amount of extra space.**

**Problem**: **Odd Even Linked List**

Description: Given the head of a singly linked list, group all nodes with odd indices together followed by nodes with even indices. The first node is considered odd, and the second node is even, and so on.

Solution:

public class Solution {

public ListNode oddEvenList(ListNode head) {

if (head == null || head.next == null) {

return head; // No need to process if the list is empty or has only one node

}

ListNode odd = head; // Pointer for odd-indexed nodes

ListNode even = head.next; // Pointer for even-indexed nodes

ListNode evenHead = even; // Save the head of the even-indexed list

// Traverse the list and separate odd and even nodes

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

odd.next = even.next; // Link odd to the next odd node

odd = odd.next; // Move odd pointer forward

even.next = odd.next; // Link even to the next even node

even = even.next; // Move even pointer forward

}

// Connect the end of the odd list to the head of the even list

odd.next = evenHead;

return head;

}

}

**Time Complexity:** **O(n)**

* + The algorithm makes a single pass through the list.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem 75: Copy List with Random Pointer (LeetCode 138)**

**Description**: Deep copy a linked list with a random pointer.

**Solution**:

java

Copy

public Node copyRandomList(Node head) {

if (head == null) return null;

Map<Node, Node> map = new HashMap<>();

Node current = head;

while (current != null) {

map.put(current, new Node(current.val));

current = current.next;

}

current = head;

while (current != null) {

map.get(current).next = map.get(current.next);

map.get(current).random = map.get(current.random);

current = current.next;

}

return map.get(head);

}

**Explanation**:

* Use a hashmap to map original nodes to their copies.
* Reconstruct the copied list using the map.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 89: Add Two Numbers II (LeetCode 445)**

**Description**: Add two numbers represented as linked lists (most significant digit first).

**Solution**:

java

Copy

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

Stack<Integer> stack1 = new Stack<>();

Stack<Integer> stack2 = new Stack<>();

while (l1 != null) {

stack1.push(l1.val);

l1 = l1.next;

}

while (l2 != null) {

stack2.push(l2.val);

l2 = l2.next;

}

ListNode result = null;

int carry = 0;

while (!stack1.isEmpty() || !stack2.isEmpty() || carry != 0) {

int sum = carry;

if (!stack1.isEmpty()) sum += stack1.pop();

if (!stack2.isEmpty()) sum += stack2.pop();

ListNode node = new ListNode(sum % 10);

node.next = result;

result = node;

carry = sum / 10;

}

return result;

}

**Explanation**:

* Use stacks to reverse the linked lists.
* Add the numbers digit by digit and construct the result linked list.

**Time Complexity**: O(n + m).  
**Space Complexity**: O(n + m).

**Problem**: **Insertion Sort List**

**Description**: Given the head of a singly linked list, sort the list using insertion sort and return the sorted list.

**Solution**:

public class Solution {

public ListNode insertionSortList(ListNode head) {

if (head == null || head.next == null) {

return head; // No need to sort if the list is empty or has only one node

}

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

ListNode curr = head; // Pointer to traverse the original list

while (curr != null) {

ListNode prev = dummy; // Pointer to find the insertion position in the sorted list

ListNode next = curr.next; // Save the next node to process

// Find the correct position to insert the current node

while (prev.next != null && prev.next.val < curr.val) {

prev = prev.next;

}

// Insert the current node into the sorted list

curr.next = prev.next;

prev.next = curr;

// Move to the next node in the original list

curr = next;

}

return dummy.next;

}

}

**Time Complexity:** **O(n^2)**

* + In the worst case, each insertion requires traversing the entire sorted list, leading to O(n2)*O*(*n*2) time.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem 80: Flatten a Multilevel Doubly Linked List (LeetCode 430)**

**Description**: Flatten a multilevel doubly linked list.

**Solution**:

java

Copy

public Node flatten(Node head) {

if (head == null) return null;

Node current = head;

while (current != null) {

if (current.child != null) {

Node next = current.next;

Node child = flatten(current.child);

current.next = child;

child.prev = current;

current.child = null;

while (current.next != null) {

current = current.next;

}

current.next = next;

if (next != null) next.prev = current;

}

current = current.next;

}

return head;

}

**Explanation**:

* Recursively flatten the child list and merge it with the main list.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 126: Convert Sorted List to Binary Search Tree (LeetCode 109)**

**Description**: Convert a sorted linked list to a height-balanced BST.

**Solution**:

java

Copy

public TreeNode sortedListToBST(ListNode head) {

if (head == null) return null;

return buildTree(head, null);

}

private TreeNode buildTree(ListNode head, ListNode tail) {

if (head == tail) return null;

ListNode slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

TreeNode root = new TreeNode(slow.val);

root.left = buildTree(head, slow);

root.right = buildTree(slow.next, tail);

return root;

}

**Explanation**:

* Use the slow and fast pointer technique to find the middle of the list.
* Recursively build the left and right subtrees.

**Time Complexity**: O(n log n).  
**Space Complexity**: O(log n).

**Problem : Swap Nodes in Pairs (Leetcode 24)**

**Description**: Given a linked list, swap every two adjacent nodes and return the modified list. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed).

Solution:

public class Solution {

public ListNode swapPairs(ListNode head) {

// Create a dummy node to simplify edge cases

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy; // Pointer to the node before the current pair

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

// Nodes to be swapped

ListNode first = prev.next;

ListNode second = prev.next.next;

// Swap the nodes

prev.next = second;

first.next = second.next;

second.next = first;

// Move prev to the next pair

prev = first;

}

return dummy.next;

}

}

**Time Complexity:** **O(n)**

* + The algorithm processes each node once.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem 121: Reverse Linked List II (LeetCode 92)**

**Description**: Reverse a linked list from position m to n.

**Solution**:

java

Copy

public ListNode reverseBetween(ListNode head, int m, int n) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy;

for (int i = 1; i < m; i++) prev = prev.next;

ListNode start = prev.next, then = start.next;

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

start.next = then.next;

then.next = prev.next;

prev.next = then;

then = start.next;

}

return dummy.next;

}

**Explanation**:

* Use three pointers to reverse the sublist in-place.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Category:** Binary Tree and Binary Search Tree

**Problem 25: Invert Binary Tree (LeetCode 226)**

**Description**: Invert a binary tree.

**Solution**:

java

Copy

public TreeNode invertTree(TreeNode root) {

if (root == null) return null;

TreeNode left = invertTree(root.left);

TreeNode right = invertTree(root.right);

root.left = right;

root.right = left;

return root;

}

**Explanation**:

* Recursively swap the left and right subtrees.

**Time Complexity**: O(n), where n is the number of nodes.  
**Space Complexity**: O(h), where h is the height of the tree.

**Problem 32: Diameter of Binary Tree (LeetCode 543)**

**Description**: Find the diameter of a binary tree (the longest path between any two nodes).

**Solution**:

java

Copy

int maxDiameter = 0;

public int diameterOfBinaryTree(TreeNode root) {

maxDepth(root);

return maxDiameter;

}

private int maxDepth(TreeNode node) {

if (node == null) return 0;

int left = maxDepth(node.left);

int right = maxDepth(node.right);

maxDiameter = Math.max(maxDiameter, left + right);

return Math.max(left, right) + 1;

}

**Explanation**:

* Use a helper function to calculate the depth of each node.
* Update the maximum diameter during traversal.

**Time Complexity**: O(n).  
**Space Complexity**: O(h), where h is the height of the tree.

**Problem 35: Lowest Common Ancestor of a Binary Search Tree (LeetCode 235)**

**Description**: Find the lowest common ancestor of two nodes in a BST.

**Solution**:

java

Copy

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

while (root != null) {

if (root.val > p.val && root.val > q.val) root = root.left;

else if (root.val < p.val && root.val < q.val) root = root.right;

else return root;

}

return null;

}

**Explanation**:

* Traverse the BST based on the values of p and q.

**Time Complexity**: O(h), where h is the height of the tree.  
**Space Complexity**: O(1).

**Problem 84: Validate Binary Search Tree (LeetCode 98)**

**Description**: Check if a binary tree is a valid BST.

**Solution**:

java

Copy

public boolean isValidBST(TreeNode root) {

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

}

private boolean isValidBST(TreeNode node, long min, long max) {

if (node == null) return true;

if (node.val <= min || node.val >= max) return false;

return isValidBST(node.left, min, node.val) && isValidBST(node.right, node.val, max);

}

**Explanation**:

* Use recursion to check if each node's value is within the valid range.

**Time Complexity**: O(n).  
**Space Complexity**: O(h).

**Problem 19: Closest Binary Search Tree Value (LeetCode 270)**

**Description**: Given a binary search tree and a target value, find the value in the BST that is closest to the target.

**Solution**:

java

Copy

public int closestValue(TreeNode root, double target) {

int closest = root.val;

while (root != null) {

if (Math.abs(root.val - target) < Math.abs(closest - target)) {

closest = root.val;

}

root = (target < root.val) ? root.left : root.right;

}

return closest;

}

**Explanation**:

* Traverse the BST and update the closest value based on the absolute difference.

**Time Complexity**: O(h), where h is the height of the tree.  
**Space Complexity**: O(1).

**Problem 20: Subtree of Another Tree (LeetCode 572)**

**Description**: Given two binary trees s and t, check if t is a subtree of s.

**Solution**:

java

Copy

public boolean isSubtree(TreeNode s, TreeNode t) {

if (s == null) return false;

if (isSameTree(s, t)) return true;

return isSubtree(s.left, t) || isSubtree(s.right, t);

}

private boolean isSameTree(TreeNode s, TreeNode t) {

if (s == null && t == null) return true;

if (s == null || t == null) return false;

return (s.val == t.val) && isSameTree(s.left, t.left) && isSameTree(s.right, t.right);

}

**Explanation**:

* Recursively check if t matches any subtree of s.
* Use a helper function to compare two trees.

**Time Complexity**: O(m \* n), where m and n are the sizes of the trees.  
**Space Complexity**: O(h), where h is the height of s.

**Problem**: Binary Tree Level Order Traversal (Leetcode 102)

Description: Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Solution:

import java.util.\*;

public class Solution {

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

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

if (root == null) {

return result; // Return empty list if the tree is empty

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root); // Add the root node to the queue

while (!queue.isEmpty()) {

int levelSize = queue.size(); // Number of nodes at the current level

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

// Process all nodes at the current level

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

TreeNode node = queue.poll();

currentLevel.add(node.val); // Add the node's value to the current level

// Add the node's children to the queue for the next level

if (node.left != null) {

queue.offer(node.left);

}

if (node.right != null) {

queue.offer(node.right);

}

}

// Add the current level to the result

result.add(currentLevel);

}

return result;

}

}

**Time Complexity:** **O(n)**

* + Each node is processed once.

**Space Complexity:** **O(n)**

* + The queue can hold up to n*n* nodes in the worst case (when the tree is a complete binary tree).

**Problem 107: Binary Tree Right Side View (LeetCode 199)**

**Description**: Return the nodes visible from the right side of a binary tree.

**Solution**:

java

Copy

public List<Integer> rightSideView(TreeNode root) {

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

if (root == null) return result;

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()) {

int size = queue.size();

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

TreeNode node = queue.poll();

if (i == size - 1) result.add(node.val);

if (node.left != null) queue.offer(node.left);

if (node.right != null) queue.offer(node.right);

}

}

return result;

}

**Explanation**:

* Use BFS to traverse the tree level by level.
* Add the last node of each level to the result.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 92: Flatten Binary Tree to Linked List (LeetCode 114)**

**Description**: Flatten a binary tree to a linked list in-place.

**Solution**:

java

Copy

public void flatten(TreeNode root) {

if (root == null) return;

flatten(root.left);

flatten(root.right);

TreeNode right = root.right;

root.right = root.left;

root.left = null;

while (root.right != null) {

root = root.right;

}

root.right = right;

}

**Explanation**:

* Use recursion to flatten the left and right subtrees.
* Attach the flattened left subtree to the right of the root.

**Time Complexity**: O(n).  
**Space Complexity**: O(h).

**Problem 91: Construct Binary Tree from Preorder and Inorder Traversal (LeetCode 105)**

**Description**: Construct a binary tree from preorder and inorder traversal arrays.

**Solution**:

java

Copy

public TreeNode buildTree(int[] preorder, int[] inorder) {

return buildTree(preorder, inorder, 0, 0, inorder.length - 1);

}

private TreeNode buildTree(int[] preorder, int[] inorder, int preStart, int inStart, int inEnd) {

if (preStart >= preorder.length || inStart > inEnd) return null;

TreeNode root = new TreeNode(preorder[preStart]);

int inIndex = 0;

for (int i = inStart; i <= inEnd; i++) {

if (inorder[i] == root.val) {

inIndex = i;

break;

}

}

root.left = buildTree(preorder, inorder, preStart + 1, inStart, inIndex - 1);

root.right = buildTree(preorder, inorder, preStart + inIndex - inStart + 1, inIndex + 1, inEnd);

return root;

}

**Explanation**:

* Use recursion to construct the tree by finding the root in the inorder array.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem: Construct Binary Tree from Inorder and Postorder Traversal (Leetcode 106)**

**Description:** Given two integer arrays inorder and postorder representing the inorder and postorder traversals of a binary tree, construct and return the binary tree.

Solution:

import java.util.\*;

public class Solution {

public TreeNode buildTree(int[] inorder, int[] postorder) {

// Map to store the indices of inorder elements for quick lookup

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

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

inorderMap.put(inorder[i], i);

}

// Start the recursive construction

return buildTreeHelper(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1, inorderMap);

}

private TreeNode buildTreeHelper(int[] inorder, int inStart, int inEnd, int[] postorder, int postStart, int postEnd, Map<Integer, Integer> inorderMap) {

// Base case: no elements to process

if (inStart > inEnd || postStart > postEnd) {

return null;

}

// The last element in postorder is the root

int rootVal = postorder[postEnd];

TreeNode root = new TreeNode(rootVal);

// Find the index of the root in inorder

int rootIndex = inorderMap.get(rootVal);

// Calculate the number of elements in the left subtree

int leftSubtreeSize = rootIndex - inStart;

// Recursively build the left and right subtrees

root.left = buildTreeHelper(inorder, inStart, rootIndex - 1, postorder, postStart, postStart + leftSubtreeSize - 1, inorderMap);

root.right = buildTreeHelper(inorder, rootIndex + 1, inEnd, postorder, postStart + leftSubtreeSize, postEnd - 1, inorderMap);

return root;

}

}

**Time Complexity:** **O(n)**

* + Each element is processed once, and the inorder map allows constant-time lookups.

**Space Complexity:** **O(n)**

* + The space is used for the recursion stack and the inorder map.

**Problem 119: Construct Binary Tree from Preorder and Postorder Traversal (LeetCode 889)**

**Description**: Construct a binary tree from its preorder and postorder traversal.

**Solution**:

java

Copy

public TreeNode constructFromPrePost(int[] pre, int[] post) {

return buildTree(pre, post, 0, pre.length - 1, 0, post.length - 1);

}

private TreeNode buildTree(int[] pre, int[] post, int preStart, int preEnd, int postStart, int postEnd) {

if (preStart > preEnd) return null;

TreeNode root = new TreeNode(pre[preStart]);

if (preStart == preEnd) return root;

int leftRootVal = pre[preStart + 1];

int postIndex = postStart;

while (post[postIndex] != leftRootVal) postIndex++;

int leftSize = postIndex - postStart + 1;

root.left = buildTree(pre, post, preStart + 1, preStart + leftSize, postStart, postIndex);

root.right = buildTree(pre, post, preStart + leftSize + 1, preEnd, postIndex + 1, postEnd - 1);

return root;

}

**Explanation**:

* Use recursion to construct the tree by finding the left subtree in the postorder array.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 113: Binary Search Tree Iterator (LeetCode 173)**

**Description**: Implement an iterator for a binary search tree.

**Solution**:

java

Copy

class BSTIterator {

private Stack<TreeNode> stack = new Stack<>();

public BSTIterator(TreeNode root) {

pushAll(root);

}

public int next() {

TreeNode node = stack.pop();

pushAll(node.right);

return node.val;

}

public boolean hasNext() {

return !stack.isEmpty();

}

private void pushAll(TreeNode node) {

while (node != null) {

stack.push(node);

node = node.left;

}

}

}

**Explanation**:

* Use a stack to simulate inorder traversal.

**Time Complexity**: O(1) for next() and hasNext().  
**Space Complexity**: O(h).

**Problem**: Convert Binary Search Tree to Sorted Doubly Linked List (Leetcode 426)

**Description**: Given a Binary Search Tree (BST), convert it to a sorted circular doubly linked list in-place. The left and right pointers in the tree nodes should be used as the previous and next pointers in the doubly linked list.

Solution:

class Node {

public int val;

public Node left;

public Node right;

public Node(int val) {

this.val = val;

}

}

public class Solution {

private Node prev = null; // Pointer to the previous node in the linked list

private Node head = null; // Pointer to the head of the linked list

public Node treeToDoublyList(Node root) {

if (root == null) {

return null; // Return null if the tree is empty

}

// Perform in-order traversal to build the doubly linked list

inorderTraversal(root);

// Connect the head and tail to make the list circular

head.left = prev;

prev.right = head;

return head;

}

private void inorderTraversal(Node curr) {

if (curr == null) {

return;

}

// Traverse the left subtree

inorderTraversal(curr.left);

// Process the current node

if (prev == null) {

// If prev is null, this is the first node (head)

head = curr;

} else {

// Link the previous node and the current node

prev.right = curr;

curr.left = prev;

}

// Update prev to the current node

prev = curr;

// Traverse the right subtree

inorderTraversal(curr.right);

}

}

**Time Complexity:** **O(n)**

* + Each node is processed once during the in-order traversal.

**Space Complexity:** **O(h)**

* + The recursion stack uses space proportional to the height of the tree (h). In the worst case (skewed tree), h = n.

**Problem**: Find Bottom Left Tree Value Leetcode 513)

**Description**: Given the root of a binary tree, return the value of the leftmost node in the last row of the tree.

**Solution**:

import java.util.\*;

public class Solution {

public int findBottomLeftValue(TreeNode root) {

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root); // Add the root node to the queue

int leftmostValue = 0;

while (!queue.isEmpty()) {

int levelSize = queue.size(); // Number of nodes at the current level

leftmostValue = queue.peek().val; // Track the leftmost node at the current level

// Process all nodes at the current level

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

TreeNode node = queue.poll();

// Add the left child first to ensure leftmost node is processed first

if (node.left != null) {

queue.offer(node.left);

}

if (node.right != null) {

queue.offer(node.right);

}

}

}

return leftmostValue;

}

}

**Time Complexity:** **O(n)**

* + Each node is processed once.

**Space Complexity:** **O(n)**

* + The queue can hold up to n*n* nodes in the worst case (when the tree is a complete binary tree).

**Problem 93: Maximum Width of Binary Tree (LeetCode 662)**

**Description**: Find the maximum width of a binary tree.

**Solution**:

java

Copy

public int widthOfBinaryTree(TreeNode root) {

if (root == null) return 0;

Queue<Pair<TreeNode, Integer>> queue = new LinkedList<>();

queue.offer(new Pair<>(root, 0));

int maxWidth = 0;

while (!queue.isEmpty()) {

int size = queue.size();

int left = queue.peek().getValue(), right = left;

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

Pair<TreeNode, Integer> pair = queue.poll();

TreeNode node = pair.getKey();

right = pair.getValue();

if (node.left != null) queue.offer(new Pair<>(node.left, 2 \* right));

if (node.right != null) queue.offer(new Pair<>(node.right, 2 \* right + 1));

}

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

}

return maxWidth;

}

**Explanation**:

* Use BFS to assign indices to nodes and calculate the width at each level.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem**: Sum Root to Leaf Numbers (Leetcode 129)

**Description**: Given a binary tree where each node contains a digit from 0 to 9, return the total sum of all root-to-leaf numbers. A root-to-leaf number is formed by concatenating the digits along the path from the root to a leaf node.

**Solution**:

public class Solution {

public int sumNumbers(TreeNode root) {

return dfs(root, 0);

}

private int dfs(TreeNode node, int currentSum) {

if (node == null) {

return 0; // Base case: return 0 for null nodes

}

// Update the current sum by appending the current node's value

currentSum = currentSum \* 10 + node.val;

// If the node is a leaf, return the current sum

if (node.left == null && node.right == null) {

return currentSum;

}

// Recursively compute the sum for the left and right subtrees

return dfs(node.left, currentSum) + dfs(node.right, currentSum);

}

}

**Time Complexity:** **O(n)**

* + Each node is visited once.

**Space Complexity:** **O(h)**

* + The recursion stack uses space proportional to the height of the tree (h). In the worst case (skewed tree), h = n.

**Problem 43: Symmetric Tree (LeetCode 101)**

**Description**: Check if a binary tree is symmetric.

**Solution**:

java

Copy

public boolean isSymmetric(TreeNode root) {

return isMirror(root, root);

}

private boolean isMirror(TreeNode t1, TreeNode t2) {

if (t1 == null && t2 == null) return true;

if (t1 == null || t2 == null) return false;

return (t1.val == t2.val) && isMirror(t1.left, t2.right) && isMirror(t1.right, t2.left);

}

**Explanation**:

* Use a helper function to compare the left and right subtrees.

**Time Complexity**: O(n).  
**Space Complexity**: O(h), where h is the height of the tree.

**Problem 49: Same Tree (LeetCode 100)**

**Description**: Check if two binary trees are the same.

**Solution**:

java

Copy

public boolean isSameTree(TreeNode p, TreeNode q) {

if (p == null && q == null) return true;

if (p == null || q == null) return false;

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

}

**Explanation**:

* Recursively compare the values and structure of the trees.

**Time Complexity**: O(n).  
**Space Complexity**: O(h), where h is the height of the tree.

**Problem**: Binary Tree Vertical Order Traversal Leetcode 314)

**Description**: Given the root of a binary tree, return the vertical order traversal of its nodes' values. (i.e., from top to bottom, column by column). If two nodes are in the same row and column, the order should be from left to right.

Solution:

import java.util.\*;

public class Solution {

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

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

if (root == null) {

return result; // Return empty list if the tree is empty

}

// Map to store nodes grouped by column index

Map<Integer, List<Integer>> columnMap = new TreeMap<>();

// Queue for level-order traversal

Queue<TreeNode> nodeQueue = new LinkedList<>();

Queue<Integer> columnQueue = new LinkedList<>();

// Start with the root node at column 0

nodeQueue.offer(root);

columnQueue.offer(0);

while (!nodeQueue.isEmpty()) {

TreeNode node = nodeQueue.poll();

int column = columnQueue.poll();

// Add the node's value to the corresponding column in the map

columnMap.computeIfAbsent(column, k -> new ArrayList<>()).add(node.val);

// Add left child to the queue with column - 1

if (node.left != null) {

nodeQueue.offer(node.left);

columnQueue.offer(column - 1);

}

// Add right child to the queue with column + 1

if (node.right != null) {

nodeQueue.offer(node.right);

columnQueue.offer(column + 1);

}

}

// Extract the results in sorted order of columns

for (List<Integer> columnNodes : columnMap.values()) {

result.add(columnNodes);

}

return result;

}

}

**Time Complexity:** **O(n log n)**

* + The TreeMap sorts columns in O(nlog⁡n)*O*(*n*log*n*) time.
  + Each node is processed once during BFS.

**Space Complexity:** **O(n)**

* + The queue and map store up to n*n* nodes.

**Category: Dynamic Programming**

**Problem 29: Climbing Stairs (LeetCode 70)**

**Description**: Given n steps, find the number of distinct ways to climb to the top (you can take 1 or 2 steps at a time).

**Solution**:

java

Copy

public int climbStairs(int n) {

if (n == 1) return 1;

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

dp[1] = 1;

dp[2] = 2;

for (int i = 3; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

**Explanation**:

* Use dynamic programming to store the number of ways to reach each step.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 40: Best Time to Buy and Sell Stock II (LeetCode 122)**

**Description**: Maximize profit by buying and selling stocks multiple times.

**Solution**:

java

Copy

public int maxProfit(int[] prices) {

int profit = 0;

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

if (prices[i] > prices[i - 1]) {

profit += prices[i] - prices[i - 1];

}

}

return profit;

}

**Explanation**:

* Add the profit from every increasing pair of prices.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 74: Best Time to Buy and Sell Stock III (LeetCode 123)**

**Description**: Find the maximum profit with at most two transactions.

**Solution**:

java

Copy

public int maxProfit(int[] prices) {

int buy1 = Integer.MAX\_VALUE, buy2 = Integer.MAX\_VALUE;

int profit1 = 0, profit2 = 0;

for (int price : prices) {

buy1 = Math.min(buy1, price);

profit1 = Math.max(profit1, price - buy1);

buy2 = Math.min(buy2, price - profit1);

profit2 = Math.max(profit2, price - buy2);

}

return profit2;

}

**Explanation**:

* Track the minimum cost for the first and second buy.
* Track the maximum profit for the first and second transaction.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 27: Maximum Subarray (LeetCode 53)**

**Description**: Find the contiguous subarray with the largest sum.

**Solution**:

java

Copy

public int maxSubArray(int[] nums) {

int maxSum = nums[0];

int currentSum = nums[0];

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

currentSum = Math.max(nums[i], currentSum + nums[i]);

maxSum = Math.max(maxSum, currentSum);

}

return maxSum;

}

**Explanation**:

* Use Kadane's algorithm to track the maximum subarray sum.

**Time Complexity**: O(n), where n is the length of the array.  
**Space Complexity**: O(1).

**Problem 71: Longest Increasing Path in a Matrix (LeetCode 329)**

**Description**: Find the length of the longest increasing path in a matrix.

**Solution**:

java

Copy

public int longestIncreasingPath(int[][] matrix) {

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

int m = matrix.length, n = matrix[0].length;

int[][] cache = new int[m][n];

int max = 1;

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

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

max = Math.max(max, dfs(matrix, i, j, cache));

}

}

return max;

}

private int dfs(int[][] matrix, int i, int j, int[][] cache) {

if (cache[i][j] != 0) return cache[i][j];

int max = 1;

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

for (int[] dir : dirs) {

int x = i + dir[0], y = j + dir[1];

if (x >= 0 && x < matrix.length && y >= 0 && y < matrix[0].length && matrix[x][y] > matrix[i][j]) {

max = Math.max(max, 1 + dfs(matrix, x, y, cache));

}

}

cache[i][j] = max;

return max;

}

**Explanation**:

* Use DFS with memoization to find the longest increasing path.

**Time Complexity**: O(m \* n).  
**Space Complexity**: O(m \* n).

**Problem 115: Coin Change (LeetCode 322)**

**Description**: Find the minimum number of coins needed to make up a given amount.

**Solution**:

java

Copy

public int coinChange(int[] coins, int amount) {

int[] dp = new int[amount + 1];

Arrays.fill(dp, amount + 1);

dp[0] = 0;

for (int coin : coins) {

for (int i = coin; i <= amount; i++) {

dp[i] = Math.min(dp[i], dp[i - coin] + 1);

}

}

return dp[amount] > amount ? -1 : dp[amount];

}

**Explanation**:

* Use dynamic programming to compute the minimum number of coins for each amount.

**Time Complexity**: O(n \* m), where n is the amount and m is the number of coins.  
**Space Complexity**: O(n).

Problem 98: Word Break (LeetCode 139)

**Description**: Determine if a string can be segmented into space-separated words from a dictionary.

**Solution**:

java

Copy

public boolean wordBreak(String s, List<String> wordDict) {

Set<String> set = new HashSet<>(wordDict);

boolean[] dp = new boolean[s.length() + 1];

dp[0] = true;

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

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

if (dp[j] && set.contains(s.substring(j, i))) {

dp[i] = true;

break;

}

}

}

return dp[s.length()];

}

**Explanation**:

* Use dynamic programming to track if substrings can be segmented.

**Time Complexity**: O(n^2).  
**Space Complexity**: O(n).

**Problem 60: Word Break II (LeetCode 140)**

**Description**: Given a string and a dictionary, return all possible sentences formed by breaking the string into dictionary words.

**Solution**:

java

Copy

public List<String> wordBreak(String s, List<String> wordDict) {

return dfs(s, wordDict, new HashMap<>());

}

private List<String> dfs(String s, List<String> wordDict, Map<String, List<String>> memo) {

if (memo.containsKey(s)) return memo.get(s);

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

if (s.isEmpty()) {

result.add("");

return result;

}

for (String word : wordDict) {

if (s.startsWith(word)) {

List<String> subResults = dfs(s.substring(word.length()), wordDict, memo);

for (String subResult : subResults) {

result.add(word + (subResult.isEmpty() ? "" : " ") + subResult);

}

}

}

memo.put(s, result);

return result;

}

**Explanation**:

* Use DFS with memoization to generate all valid sentences.

**Time Complexity**: O(n^2).  
**Space Complexity**: O(n^2).

**Problem 65: Edit Distance (LeetCode 72)**

**Description**: Given two words, find the minimum number of operations (insert, delete, replace) required to convert one word to the other.

**Solution**:

java

Copy

public int minDistance(String word1, String word2) {

int m = word1.length(), n = word2.length();

int[][] dp = new int[m + 1][n + 1];

for (int i = 0; i <= m; i++) dp[i][0] = i;

for (int j = 0; j <= n; j++) dp[0][j] = j;

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

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

if (word1.charAt(i - 1) == word2.charAt(j - 1)) {

dp[i][j] = dp[i - 1][j - 1];

} else {

dp[i][j] = 1 + Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1]));

}

}

}

return dp[m][n];

}

**Explanation**:

* Use dynamic programming to compute the minimum edit distance.

**Time Complexity**: O(m \* n).  
**Space Complexity**: O(m \* n).

**Problem 124: Unique Paths II (LeetCode 63)**

**Description**: Find the number of unique paths in a grid with obstacles.

**Solution**:

java

Copy

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

int m = obstacleGrid.length, n = obstacleGrid[0].length;

int[][] dp = new int[m][n];

dp[0][0] = obstacleGrid[0][0] == 1 ? 0 : 1;

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

dp[i][0] = (obstacleGrid[i][0] == 1) ? 0 : dp[i - 1][0];

}

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

dp[0][j] = (obstacleGrid[0][j] == 1) ? 0 : dp[0][j - 1];

}

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

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

dp[i][j] = (obstacleGrid[i][j] == 1) ? 0 : dp[i - 1][j] + dp[i][j - 1];

}

}

return dp[m - 1][n - 1];

}

**Explanation**:

* Use dynamic programming to count paths, avoiding obstacles.

**Time Complexity**: O(m \* n).  
**Space Complexity**: O(m \* n).

**Problem**: Minimum Path Sum Leetcode 64)

**Description**: Given a m x n grid filled with non-negative numbers, find a path from the top-left corner to the bottom-right corner that minimizes the sum of the numbers along the path. You can only move either down or right at any point in time.

**Solution**:

public class Solution {

public int minPathSum(int[][] grid) {

int m = grid.length; // Number of rows

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

// DP table to store the minimum path sum to reach each cell

int[][] dp = new int[m][n];

// Initialize the starting cell

dp[0][0] = grid[0][0];

// Fill the first row

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

dp[0][j] = dp[0][j - 1] + grid[0][j];

}

// Fill the first column

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

dp[i][0] = dp[i - 1][0] + grid[i][0];

}

// Fill the rest of the DP table

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

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

dp[i][j] = grid[i][j] + Math.min(dp[i - 1][j], dp[i][j - 1]);

}

}

// Return the minimum path sum to the bottom-right corner

return dp[m - 1][n - 1];

}

}

**Time Complexity:** **O(m \* n)**

* + Each cell in the grid is processed once.

**Space Complexity:** **O(m \* n)**

* + The DP table uses O(m×n)*O*(*m*×*n*) space. This can be optimized to O(n)*O*(*n*) by using a 1D DP array.

**Problem: Decode Ways (Leetcode 91)**

**Description** : Given a string s containing only digits, determine the number of ways to decode it into letters, where 'A' maps to 1, 'B' to 2, ..., and 'Z' to 26.

**Solution:**

public class Solution {

public int numDecodings(String s) {

int n = s.length();

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

dp[0] = 1; // Empty string has 1 way to decode

dp[1] = s.charAt(0) == '0' ? 0 : 1; // First character

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

int oneDigit = Integer.parseInt(s.substring(i - 1, i)); // Single digit

int twoDigits = Integer.parseInt(s.substring(i - 2, i)); // Two digits

if (oneDigit >= 1 && oneDigit <= 9) {

dp[i] += dp[i - 1];

}

if (twoDigits >= 10 && twoDigits <= 26) {

dp[i] += dp[i - 2];

}

}

return dp[n];

}

}

**Time Complexity: O(n)**

* + The algorithm processes each character of the string exactly once.
  + The dynamic programming approach ensures that each step is computed in constant time.

**Space Complexity: O(n)**

* + The DP array uses space proportional to the length of the input string.
  + This can be optimized to **O(1)** by using two variables instead of an array to store the previous two states.

**Problem**: **Longest Palindromic Substring (Leetcode 5)**

**Description:** Given a string s, return the longest palindromic substring in s. A palindrome is a string that reads the same backward as forward.

**Solution**:

public class Solution {

public String longestPalindrome(String s) {

if (s == null || s.length() < 1) {

return "";

}

int start = 0, end = 0; // Indices of the longest palindrome

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

// Find longest odd-length palindrome with center at i

int len1 = expandAroundCenter(s, i, i);

// Find longest even-length palindrome with center at i and i+1

int len2 = expandAroundCenter(s, i, i + 1);

// Get the maximum length

int len = Math.max(len1, len2);

// Update the indices of the longest palindrome

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private int expandAroundCenter(String s, int left, int right) {

while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {

left--;

right++;

}

// Return the length of the palindrome

return right - left - 1;

}

}

**Time Complexity:** **O(n^2)**

* + For each character, the algorithm expands around the center, which takes O(n)*O*(*n*) in the worst case. Since this is done for all n*n* characters, the total time complexity is O(n2)*O*(*n*2).

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem 56: Trapping Rain Water (LeetCode 42)**

**Description**: Compute how much water can be trapped after raining.

**Solution**:

java

Copy

public int trap(int[] height) {

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

int leftMax = 0, rightMax = 0;

int result = 0;

while (left < right) {

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

if (height[left] >= leftMax) leftMax = height[left];

else result += leftMax - height[left];

left++;

} else {

if (height[right] >= rightMax) rightMax = height[right];

else result += rightMax - height[right];

right--;

}

}

return result;

}

**Explanation**:

* Use two pointers to track the maximum heights from the left and right.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Category: Graph and BFS/DFS**

**Problem 81: Number of Islands (LeetCode 200)**

**Description**: Count the number of islands in a 2D grid.

**Solution**:

java

Copy

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

int count = 0;

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

for (int j = 0; j < grid[0].length; j++) {

if (grid[i][j] == '1') {

dfs(grid, i, j);

count++;

}

}

}

return count;

}

private void dfs(char[][] grid, int i, int j) {

if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] != '1') return;

grid[i][j] = '0';

dfs(grid, i + 1, j);

dfs(grid, i - 1, j);

dfs(grid, i, j + 1);

dfs(grid, i, j - 1);

}

**Explanation**:

* Use DFS to mark all connected land cells as visited.

**Time Complexity**: O(m \* n).  
**Space Complexity**: O(m \* n).

**Problem 33: Flood Fill (LeetCode 733)**

**Description**: Perform a flood fill on an image starting from a given pixel.

**Solution**:

java

Copy

public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {

if (image[sr][sc] == newColor) return image;

dfs(image, sr, sc, image[sr][sc], newColor);

return image;

}

private void dfs(int[][] image, int r, int c, int oldColor, int newColor) {

if (r < 0 || r >= image.length || c < 0 || c >= image[0].length || image[r][c] != oldColor) return;

image[r][c] = newColor;

dfs(image, r + 1, c, oldColor, newColor);

dfs(image, r - 1, c, oldColor, newColor);

dfs(image, r, c + 1, oldColor, newColor);

dfs(image, r, c - 1, oldColor, newColor);

}

**Explanation**:

* Use DFS to traverse and fill connected pixels with the same color.

**Time Complexity**: O(n \* m), where n and m are the dimensions of the image.  
**Space Complexity**: O(n \* m) (recursion stack).

Problem 96: Word Search (LeetCode 79)

**Description**: Given a 2D board and a word, determine if the word exists in the grid.

**Solution**:

java

Copy

public boolean exist(char[][] board, String word) {

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

for (int j = 0; j < board[0].length; j++) {

if (dfs(board, word, i, j, 0)) return true;

}

}

return false;

}

private boolean dfs(char[][] board, String word, int i, int j, int index) {

if (index == word.length()) return true;

if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || board[i][j] != word.charAt(index)) return false;

char temp = board[i][j];

board[i][j] = '#';

boolean found = dfs(board, word, i + 1, j, index + 1) ||

dfs(board, word, i - 1, j, index + 1) ||

dfs(board, word, i, j + 1, index + 1) ||

dfs(board, word, i, j - 1, index + 1);

board[i][j] = temp;

return found;

}

**Explanation**:

* Use DFS to search for the word in all four directions.
* Mark visited cells to avoid reuse.

**Time Complexity**: O(m \* n \* 4^k), where m and n are the board dimensions, and k is the word length.  
**Space Complexity**: O(k) (recursion stack).

**Problem**: Word Ladder (Leetcode 127)

**Description**: Given two words, beginWord and endWord, and a list of words wordList, find the length of the shortest transformation sequence from beginWord to endWord such that

**Solution**:

import java.util.\*;

public class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

// Convert wordList to a set for O(1) lookups

Set<String> wordSet = new HashSet<>(wordList);

if (!wordSet.contains(endWord)) {

return 0; // endWord is not in wordList

}

// Queue for BFS

Queue<String> queue = new LinkedList<>();

queue.offer(beginWord);

// Set to track visited words

Set<String> visited = new HashSet<>();

visited.add(beginWord);

// Level counter

int level = 1;

while (!queue.isEmpty()) {

int size = queue.size();

// Process all nodes at the current level

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

String currentWord = queue.poll();

// Check if the current word can be transformed into endWord

if (currentWord.equals(endWord)) {

return level;

}

// Generate all possible transformations of the current word

for (int j = 0; j < currentWord.length(); j++) {

char[] wordArray = currentWord.toCharArray();

// Try changing the current character to every possible letter

for (char c = 'a'; c <= 'z'; c++) {

wordArray[j] = c;

String transformedWord = new String(wordArray);

// If the transformed word is in wordList and not visited, add it to the queue

if (wordSet.contains(transformedWord) && !visited.contains(transformedWord)) {

queue.offer(transformedWord);

visited.add(transformedWord);

}

}

}

}

// Increment the level after processing all nodes at the current level

level++;

}

// If endWord is not reachable, return 0

return 0;

}

}

**Time Complexity:** **O(M^2 \* N)**

* + M is the length of each word.
  + N is the number of words in wordList.
  + For each word, we generate M transformations, and each transformation takes O(M) time to construct.

**Space Complexity:** **O(M \* N)**

* + The queue and visited set can store up to N words, each of length M.

**Problem 122: Friend Circles (LeetCode 547)**

**Description**: Find the number of friend circles in a matrix of friendships.

**Solution**:

java

Copy

public int findCircleNum(int[][] M) {

int n = M.length, count = 0;

boolean[] visited = new boolean[n];

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

if (!visited[i]) {

dfs(M, visited, i);

count++;

}

}

return count;

}

private void dfs(int[][] M, boolean[] visited, int i) {

for (int j = 0; j < M.length; j++) {

if (M[i][j] == 1 && !visited[j]) {

visited[j] = true;

dfs(M, visited, j);

}

}

}

**Explanation**:

* Use DFS to traverse friend circles and mark visited nodes.

**Time Complexity**: O(n^2).  
**Space Complexity**: O(n).

**Problem 82: All Paths From Source to Target (LeetCode 797)**

**Description**: Find all paths from the source node to the target node in a directed acyclic graph.

**Solution**:

java

Copy

public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

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

dfs(graph, 0, new ArrayList<>(), result);

return result;

}

private void dfs(int[][] graph, int node, List<Integer> path, List<List<Integer>> result) {

path.add(node);

if (node == graph.length - 1) {

result.add(new ArrayList<>(path));

} else {

for (int neighbor : graph[node]) {

dfs(graph, neighbor, path, result);

}

}

path.remove(path.size() - 1);

}

**Explanation**:

* Use DFS to explore all paths from the source to the target.

**Time Complexity**: O(2^n).  
**Space Complexity**: O(n).

**Problem**: Course Schedule II (Leetcode 210)

**Description**: Given the total number of courses numCourses and a list of prerequisites represented as pairs [a, b] (where b must be taken before a), return the ordering of courses you should take to finish all courses. If it is impossible to finish all courses, return an empty array.

**Solution**:

import java.util.\*;

public class Solution {

public int[] findOrder(int numCourses, int[][] prerequisites) {

// Step 1: Build the graph and compute in-degrees

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

int[] inDegree = new int[numCourses];

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

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

}

for (int[] edge : prerequisites) {

int course = edge[0];

int prerequisite = edge[1];

graph.get(prerequisite).add(course); // Add edge from prerequisite to course

inDegree[course]++; // Increment in-degree of the course

}

// Step 2: Initialize the queue with courses having in-degree 0

Queue<Integer> queue = new LinkedList<>();

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

if (inDegree[i] == 0) {

queue.offer(i);

}

}

// Step 3: Perform BFS (Kahn's algorithm)

int[] result = new int[numCourses];

int index = 0;

while (!queue.isEmpty()) {

int course = queue.poll();

result[index++] = course; // Add the course to the result

// Reduce in-degree of neighbors and add them to the queue if in-degree becomes 0

for (int neighbor : graph.get(course)) {

inDegree[neighbor]--;

if (inDegree[neighbor] == 0) {

queue.offer(neighbor);

}

}

}

// Step 4: Check if all courses were processed

if (index == numCourses) {

return result;

} else {

return new int[0]; // Impossible to finish all courses (cycle detected)

}

}

}

**Time Complexity:** **O(V + E)**

* + V is the number of courses (vertices).
  + E is the number of prerequisites (edges).
  + Building the graph takes O(V+E)*O*(*V*+*E*), and BFS takes O(V+E)*O*(*V*+*E*).

**Space Complexity:** **O(V + E)**

* + The graph uses O(V+E)*O*(*V*+*E*) space.
  + The queue and in-degree array use O(V)*O*(*V*) space.

**Problem 59: Alien Dictionary (LeetCode 269)**

**Description**: Given a sorted dictionary of an alien language, return the order of its letters.

**Solution**:

public String alienOrder(String[] words) {

Map<Character, Set<Character>> graph = new HashMap<>();

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

for (String word : words) {

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

inDegree.put(c, 0);

}

}

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

String word1 = words[i - 1], word2 = words[i];

int len = Math.min(word1.length(), word2.length());

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

char c1 = word1.charAt(j), c2 = word2.charAt(j);

if (c1 != c2) {

if (!graph.containsKey(c1)) graph.put(c1, new HashSet<>());

if (!graph.get(c1).contains(c2)) {

graph.get(c1).add(c2);

inDegree.put(c2, inDegree.get(c2) + 1);

}

break;

}

if (j == len - 1 && word1.length() > word2.length()) return "";

}

}

Queue<Character> queue = new LinkedList<>();

for (char c : inDegree.keySet()) {

if (inDegree.get(c) == 0) queue.add(c);

}

StringBuilder result = new StringBuilder();

while (!queue.isEmpty()) {

char c = queue.poll();

result.append(c);

if (graph.containsKey(c)) {

for (char neighbor : graph.get(c)) {

inDegree.put(neighbor, inDegree.get(neighbor) - 1);

if (inDegree.get(neighbor) == 0) queue.add(neighbor);

}

}

}

return result.length() == inDegree.size() ? result.toString() : "";

}

**Explanation**:

* Build a graph and perform topological sorting to determine the order of letters.

**Time Complexity**: O(n), where n is the total number of characters.  
**Space Complexity**: O(n).

**Problem**: Reconstruct Itinerary Leetcode 332)

**Description**: Given a list of airline tickets represented as pairs [from, to], reconstruct the itinerary in order. All tickets belong to a traveler who departs from "JFK". The itinerary must:

1. Use all tickets exactly once.
2. Return the smallest lexical order if multiple valid itineraries exist.

**Solution**:

import java.util.\*;

public class Solution {

public List<String> findItinerary(List<List<String>> tickets) {

// Step 1: Build the graph

Map<String, PriorityQueue<String>> graph = new HashMap<>();

for (List<String> ticket : tickets) {

String from = ticket.get(0);

String to = ticket.get(1);

graph.computeIfAbsent(from, k -> new PriorityQueue<>()).add(to);

}

// Step 2: Perform Hierholzer's algorithm

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

dfs("JFK", graph, result);

// Step 3: Reverse the result to get the correct order

Collections.reverse(result);

return result;

}

private void dfs(String airport, Map<String, PriorityQueue<String>> graph, List<String> result) {

PriorityQueue<String> neighbors = graph.get(airport);

// Visit all neighbors in lexical order

while (neighbors != null && !neighbors.isEmpty()) {

String nextAirport = neighbors.poll();

dfs(nextAirport, graph, result);

}

// Add the current airport to the result after visiting all neighbors

result.add(airport);

}

}

**Time Complexity:** **O(E log E)**

* + Building the graph takes O(E)*O*(*E*), where E*E* is the number of tickets.
  + Sorting the neighbors using a priority queue takes O(Elog⁡E)*O*(*E*log*E*).
  + Traversing the graph takes O(E)*O*(*E*).

**Space Complexity:** **O(E)**

* + The graph uses O(E)*O*(*E*) space.
  + The recursion stack and result list use O(E)*O*(*E*) space.

**Category: Heap and Priority Queue**

**Problem**: Kth Largest Element in an Array (Leetcode 215)

**Description**: Given an integer array nums and an integer k, return the kth largest element in the array

**Solution**:

import java.util.PriorityQueue;

public class Solution {

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

// Min-heap to store the k largest elements

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

for (int num : nums) {

minHeap.offer(num); // Add the current element to the heap

if (minHeap.size() > k) {

minHeap.poll(); // Remove the smallest element if heap size exceeds k

}

}

// The root of the heap is the kth largest element

return minHeap.peek();

}

}

**Time Complexity:** **O(n log k)**

* + Inserting and removing elements from the heap takes O(log⁡k)*O*(log*k*).
  + This is done for all n*n* elements, resulting in O(nlog⁡k)*O*(*n*log*k*).

**Space Complexity:** **O(k)**

* + The heap stores at most k elements.

**Problem** 104: Top K Frequent Elements (LeetCode 347)

**Description**: Return the k most frequent elements.

**Solution**:

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

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

for (int num : nums) {

map.put(num, map.getOrDefault(num, 0) + 1);

}

PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> map.get(b) - map.get(a));

pq.addAll(map.keySet());

int[] result = new int[k];

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

result[i] = pq.poll();

}

return result;

}

**Explanation**:

* Use a hashmap to count frequencies and a priority queue to select the top k elements.

**Time Complexity**: O(n log k).  
**Space Complexity**: O(n).

**Problem 83: Top K Frequent Words (LeetCode 692)**

**Description**: Return the k most frequent words.

**Solution**:

public List<String> topKFrequent(String[] words, int k) {

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

for (String word : words) {

map.put(word, map.getOrDefault(word, 0) + 1);

}

PriorityQueue<String> pq = new PriorityQueue<>((a, b) -> map.get(a).equals(map.get(b)) ? b.compareTo(a) : map.get(a) - map.get(b));

for (String word : map.keySet()) {

pq.offer(word);

if (pq.size() > k) pq.poll();

}

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

while (!pq.isEmpty()) {

result.add(0, pq.poll());

}

return result;

}

**Explanation**:

* Use a hashmap to count frequencies and a priority queue to select the top k words.

**Time Complexity**: O(n log k).  
**Space Complexity**: O(n).

**Problem 66: Find Median from Data Stream (LeetCode 295)**

**Description**: Design a data structure to find the median of a stream of numbers.

**Solution**:

class MedianFinder {

private PriorityQueue<Integer> minHeap;

private PriorityQueue<Integer> maxHeap;

public MedianFinder() {

minHeap = new PriorityQueue<>();

maxHeap = new PriorityQueue<>((a, b) -> b - a);

}

public void addNum(int num) {

maxHeap.offer(num);

minHeap.offer(maxHeap.poll());

if (maxHeap.size() < minHeap.size()) {

maxHeap.offer(minHeap.poll());

}

}

public double findMedian() {

if (maxHeap.size() == minHeap.size()) {

return (maxHeap.peek() + minHeap.peek()) / 2.0;

} else {

return maxHeap.peek();

}

}

}

**Explanation**:

* Use two heaps: a max-heap for the lower half and a min-heap for the upper half.

**Time Complexity**: O(log n) for addNum, O(1) for findMedian.  
**Space Complexity**: O(n).

**Problem 64: Merge k Sorted Lists (LeetCode 23)**

**Description**: Merge k sorted linked lists into one sorted linked list.

**Solution**:

public ListNode mergeKLists(ListNode[] lists) {

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

PriorityQueue<ListNode> pq = new PriorityQueue<>((a, b) -> a.val - b.val);

for (ListNode node : lists) {

if (node != null) pq.offer(node);

}

ListNode dummy = new ListNode(0);

ListNode current = dummy;

while (!pq.isEmpty()) {

ListNode node = pq.poll();

current.next = node;

current = current.next;

if (node.next != null) pq.offer(node.next);

}

return dummy.next;

}

**Explanation**:

* Use a min-heap to efficiently merge the lists.

**Time Complexity**: O(n log k), where n is the total number of nodes and k is the number of lists.  
**Space Complexity**: O(k).

**Problem 62: Sliding Window Maximum (LeetCode 239)**

**Description**: Given an array and a sliding window size k, return the maximum value in each window.

**Solution**:

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

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

int[] result = new int[nums.length - k + 1];

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

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

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

deque.pollFirst();

}

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

deque.pollLast();

}

deque.offerLast(i);

if (i >= k - 1) {

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

}

}

return result;

}

**Explanation**:

* Use a deque to maintain indices of potential maximum values in the current window.
* Remove indices that are out of the window and smaller elements from the deque.

**Time Complexity**: O(n).  
**Space Complexity**: O(k).

**Problem 116: Kth Smallest Element in a BST (LeetCode 230)**

**Description**: Find the kth smallest element in a binary search tree.

**Solution**:

public int kthSmallest(TreeNode root, int k) {

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

while (true) {

while (root != null) {

stack.push(root);

root = root.left;

}

root = stack.pop();

if (--k == 0) return root.val;

root = root.right;

}

}

**Explanation**:

* Use inorder traversal to find the kth smallest element.

**Time Complexity**: O(h + k).  
**Space Complexity**: O(h).

**Category: Hash Table and Counting**

**Problem 22: Subdomain Visit Count (LeetCode 811)**

**Description**: Given a list of count-paired domain strings, return the count of visits for each subdomain.

**Solution**:

public List<String> subdomainVisits(String[] cpdomains) {

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

for (String domain : cpdomains) {

String[] parts = domain.split(" ");

int count = Integer.parseInt(parts[0]);

String[] subdomains = parts[1].split("\\.");

String current = "";

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

current = subdomains[i] + (current.isEmpty() ? "" : ".") + current;

map.put(current, map.getOrDefault(current, 0) + count);

}

}

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

for (Map.Entry<String, Integer> entry : map.entrySet()) {

result.add(entry.getValue() + " " + entry.getKey());

}

return result;

}

**Explanation**:

* Use a hashmap to count visits for each subdomain.
* Split each domain into subdomains and update the counts.

**Time Complexity**: O(n \* m), where n is the number of domains and m is the maximum number of subdomains.  
**Space Complexity**: O(n \* m).

**Problem 36: Intersection of Two Arrays II (LeetCode 350)**

**Description**: Find the intersection of two arrays, including duplicates.

**Solution**:

public int[] intersect(int[] nums1, int[] nums2) {

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

for (int num : nums1) {

map.put(num, map.getOrDefault(num, 0) + 1);

}

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

for (int num : nums2) {

if (map.containsKey(num) && map.get(num) > 0) {

result.add(num);

map.put(num, map.get(num) - 1);

}

}

int[] output = new int[result.size()];

for (int i = 0; i < result.size(); i++) {

output[i] = result.get(i);

}

return output;

}

**Explanation**:

* Use a hashmap to count occurrences of each number in nums1.
* Traverse nums2 and add common numbers to the result.

**Time Complexity**: O(n + m), where n and m are the lengths of the arrays.  
**Space Complexity**: O(min(n, m)).

**Problem 85: Insert Delete GetRandom O(1) (LeetCode 380)**

**Description**: Design a data structure that supports insert, remove, and getRandom in average O(1) time.

**Solution**:

class RandomizedSet {

private List<Integer> list;

private Map<Integer, Integer> map;

private Random random;

public RandomizedSet() {

list = new ArrayList<>();

map = new HashMap<>();

random = new Random();

}

public boolean insert(int val) {

if (map.containsKey(val)) return false;

map.put(val, list.size());

list.add(val);

return true;

}

public boolean remove(int val) {

if (!map.containsKey(val)) return false;

int index = map.get(val);

int lastElement = list.get(list.size() - 1);

list.set(index, lastElement);

map.put(lastElement, index);

list.remove(list.size() - 1);

map.remove(val);

return true;

}

public int getRandom() {

return list.get(random.nextInt(list.size()));

}

}

**Explanation**:

* Use a list to store elements and a hashmap to store their indices for O(1) access.
* For remove, swap the element with the last element in the list and update the map.

**Time Complexity**: O(1) for all operations.  
**Space Complexity**: O(n).

**Problem 123: Design Hit Counter (LeetCode 362)**

**Description**: Design a hit counter that counts hits in the last 5 minutes.

**Solution**:

class HitCounter {

private Queue<Integer> queue;

public HitCounter() {

queue = new LinkedList<>();

}

public void hit(int timestamp) {

queue.offer(timestamp);

}

public int getHits(int timestamp) {

while (!queue.isEmpty() && timestamp - queue.peek() >= 300) {

queue.poll();

}

return queue.size();

}

}

**Explanation**:

* Use a queue to store timestamps of hits.
* Remove outdated hits when counting.

**Time Complexity**: O(1) for hit, O(n) for getHits.  
**Space Complexity**: O(n).

**Category: Matrix and Grid**

**Problem 108: Spiral Matrix (LeetCode 54)**

**Description**: Return all elements of a matrix in spiral order.

**Solution**:

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

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

if (matrix == null || matrix.length == 0) return result;

int top = 0, bottom = matrix.length - 1, left = 0, right = matrix[0].length - 1;

while (top <= bottom && left <= right) {

for (int i = left; i <= right; i++) result.add(matrix[top][i]);

top++;

for (int i = top; i <= bottom; i++) result.add(matrix[i][right]);

right--;

if (top <= bottom) {

for (int i = right; i >= left; i--) result.add(matrix[bottom][i]);

bottom--;

}

if (left <= right) {

for (int i = bottom; i >= top; i--) result.add(matrix[i][left]);

left++;

}

}

return result;

}

**Explanation**:

* Use four boundaries to traverse the matrix in spiral order.

**Time Complexity**: O(m \* n).  
**Space Complexity**: O(1).

**Problem**: Set Matrix Zeroes (LeetCode 73)

**Description**: Given an m x n matrix, if an element is 0, set its entire row and column to 0. Do this in-place.

**Solution**:

public class Solution {

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

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

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

// Flags to check if the first row or first column should be set to 0

boolean firstRowZero = false;

boolean firstColZero = false;

// Step 1: Check if the first row or first column needs to be zeroed

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

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

firstRowZero = true;

break;

}

}

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

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

firstColZero = true;

break;

}

}

// Step 2: Use the first row and first column as markers

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

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

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

matrix[i][0] = 0; // Mark the row

matrix[0][j] = 0; // Mark the column

}

}

}

// Step 3: Set rows and columns to 0 based on markers

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

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

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

matrix[i][j] = 0;

}

}

}

// Step 4: Handle the first row and first column

if (firstRowZero) {

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

matrix[0][j] = 0;

}

}

if (firstColZero) {

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

matrix[i][0] = 0;

}

}

}

}

**Time Complexity:** **O(m \* n)**

* + Each element in the matrix is processed a constant number of times.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem**: Rotate Image (LeetCode 48)

Description: Given an n x n 2D matrix representing an image, rotate the image by 90 degrees clockwise. You must perform the rotation in-place.

Solution:

public class Solution {

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

int n = matrix.length;

// Step 1: Transpose the matrix

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

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

int temp = matrix[i][j];

matrix[i][j] = matrix[j][i];

matrix[j][i] = temp;

}

}

// Step 2: Reverse each row

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

int left = 0, right = n - 1;

while (left < right) {

int temp = matrix[i][left];

matrix[i][left] = matrix[i][right];

matrix[i][right] = temp;

left++;

right--;

}

}

}

}

**Time Complexity:** **O(n^2)**

* + Each element is processed once during the transpose and once during the row reversal.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem**: Search a 2D Matrix II (LeetCode 240)

**Description**: Given an m x n matrix where each row and column is sorted in ascending order, write an efficient algorithm to search for a target value.

**Solution**:

public class Solution {

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

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return false; // Edge case: empty matrix

}

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

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

// Start from the top-right corner

int row = 0, col = n - 1;

while (row < m && col >= 0) {

if (matrix[row][col] == target) {

return true; // Target found

} else if (matrix[row][col] > target) {

col--; // Move left (eliminate the current column)

} else {

row++; // Move down (eliminate the current row)

}

}

return false; // Target not found

}

}

**Time Complexity:** **O(m + n)**

* + In the worst case, the algorithm traverses one row and one column.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem** 101: Minesweeper (LeetCode 529)

**Description**: Simulate the Minesweeper game.

**Solution**:

java

Copy

public char[][] updateBoard(char[][] board, int[] click) {

int x = click[0], y = click[1];

if (board[x][y] == 'M') {

board[x][y] = 'X';

return board;

}

dfs(board, x, y);

return board;

}

private void dfs(char[][] board, int x, int y) {

if (x < 0 || x >= board.length || y < 0 || y >= board[0].length || board[x][y] != 'E') return;

int mines = countMines(board, x, y);

if (mines > 0) {

board[x][y] = (char) (mines + '0');

return;

}

board[x][y] = 'B';

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

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

if (i == 0 && j == 0) continue;

dfs(board, x + i, y + j);

}

}

}

private int countMines(char[][] board, int x, int y) {

int count = 0;

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

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

if (i == 0 && j == 0) continue;

int nx = x + i, ny = y + j;

if (nx >= 0 && nx < board.length && ny >= 0 && ny < board[0].length && board[nx][ny] == 'M') {

count++;

}

}

}

return count;

}

**Explanation**:

* Use DFS to reveal cells and count adjacent mines.

**Time Complexity**: O(m \* n).  
**Space Complexity**: O(m \* n).

**Problem** 103: Sparse Matrix Multiplication (LeetCode 311)

**Description**: Multiply two sparse matrices.

**Solution**:

public int[][] multiply(int[][] A, int[][] B) {

int m = A.length, n = A[0].length, p = B[0].length;

int[][] result = new int[m][p];

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

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

if (A[i][k] != 0) {

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

if (B[k][j] != 0) {

result[i][j] += A[i][k] \* B[k][j];

}

}

}

}

}

return result;

}

**Explanation**:

* Optimize multiplication by skipping zero elements.

**Time Complexity**: O(m \* n \* p).  
**Space Complexity**: O(m \* p).

**Category: Sorting and Searching**

**Problem 17: Merge Sorted Array (LeetCode 88)**

**Description**: Merge two sorted arrays nums1 and nums2 into nums1 in non-decreasing order.

**Solution**:

public void merge(int[] nums1, int m, int[] nums2, int n) {

int i = m - 1, j = n - 1, k = m + n - 1;

while (i >= 0 && j >= 0) {

if (nums1[i] > nums2[j]) {

nums1[k--] = nums1[i--];

} else {

nums1[k--] = nums2[j--];

}

}

while (j >= 0) {

nums1[k--] = nums2[j--];

}

}

**Explanation**:

* Use three pointers to merge the arrays from the end.
* Fill nums1 in reverse order to avoid overwriting.

**Time Complexity**: O(m + n).  
**Space Complexity**: O(1).

**Problem 86: Merge Intervals (LeetCode 56)**

**Description**: Merge overlapping intervals.

**Solution**:

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

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

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

int[] current = intervals[0];

for (int[] interval : intervals) {

if (interval[0] <= current[1]) {

current[1] = Math.max(current[1], interval[1]);

} else {

result.add(current);

current = interval;

}

}

result.add(current);

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

}

**Explanation**:

* Sort intervals by start time.
* Merge overlapping intervals by updating the end time.

**Time Complexity**: O(n log n).  
**Space Complexity**: O(n).

**Problem 41: Search Insert Position (LeetCode 35)**

**Description**: Find the index where a target should be inserted in a sorted array.

**Solution**:

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

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

while (left <= right) {

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

if (nums[mid] == target) return mid;

if (nums[mid] < target) left = mid + 1;

else right = mid - 1;

}

return left;

}

**Explanation**:

* Use binary search to find the insertion position.

**Time Complexity**: O(log n).  
**Space Complexity**: O(1).

**Problem 118: Find Minimum in Rotated Sorted Array (LeetCode 153)**

**Description**: Find the minimum element in a rotated sorted array.

**Solution**:

public int findMin(int[] nums) {

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

while (left < right) {

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

if (nums[mid] > nums[right]) left = mid + 1;

else right = mid;

}

return nums[left];

}

**Explanation**:

* Use binary search to find the pivot point (minimum element).

**Time Complexity**: O(log n).  
**Space Complexity**: O(1).

**Problem 106: Search in Rotated Sorted Array (LeetCode 33)**

**Description**: Search for a target value in a rotated sorted array.

**Solution**:

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

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

while (left <= right) {

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

if (nums[mid] == target) return mid;

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

if (nums[left] <= target && target < nums[mid]) right = mid - 1;

else left = mid + 1;

} else {

if (nums[mid] < target && target <= nums[right]) left = mid + 1;

else right = mid - 1;

}

}

return -1;

}

**Explanation**:

* Use binary search to find the target in the rotated array.
* Check which part of the array is sorted and adjust the search range accordingly.

**Time Complexity**: O(log n).  
**Space Complexity**: O(1).

**Problem 45: First Bad Version (LeetCode 278)**

**Description**: Find the first bad version in a series of versions.

**Solution**:

public int firstBadVersion(int n) {

int left = 1, right = n;

while (left < right) {

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

if (isBadVersion(mid)) right = mid;

else left = mid + 1;

}

return left;

}

**Explanation**:

* Use binary search to find the first bad version.

**Time Complexity**: O(log n).  
**Space Complexity**: O(1).

**Problem**:Find Peak Element (Leetcode 162)

**Description**: Given an integer array nums, find a peak element and return its index. A peak element is an element that is strictly greater than its neighbors. You may imagine that nums[-1] = nums[n] = -∞.

**Solution**:

public class Solution {

public int findPeakElement(int[] nums) {

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

while (left < right) {

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

// Compare the middle element with its right neighbor

if (nums[mid] < nums[mid + 1]) {

left = mid + 1; // Peak is on the right side

} else {

right = mid; // Peak is on the left side

}

}

// When left == right, we have found a peak

return left;

}

}

**Time Complexity:** **O(log n)**

* + The binary search reduces the search space by half in each iteration.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem 26: Peak Index in a Mountain Array (LeetCode 852)**

**Description**: Given a mountain array, return the index of the peak element.

**Solution**:

public int peakIndexInMountainArray(int[] arr) {

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

while (left < right) {

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

if (arr[mid] < arr[mid + 1]) {

left = mid + 1;

} else {

right = mid;

}

}

return left;

}

**Explanation**:

* Use binary search to find the peak element.

**Time Complexity**: O(log n), where n is the length of the array.  
**Space Complexity**: O(1).

**Problem 67: Median of Two Sorted Arrays (LeetCode 4)**

**Description**: Find the median of two sorted arrays.

**Solution**:

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

if (nums1.length > nums2.length) {

return findMedianSortedArrays(nums2, nums1);

}

int x = nums1.length, y = nums2.length;

int low = 0, high = x;

while (low <= high) {

int partitionX = (low + high) / 2;

int partitionY = (x + y + 1) / 2 - partitionX;

int maxX = (partitionX == 0) ? Integer.MIN\_VALUE : nums1[partitionX - 1];

int minX = (partitionX == x) ? Integer.MAX\_VALUE : nums1[partitionX];

int maxY = (partitionY == 0) ? Integer.MIN\_VALUE : nums2[partitionY - 1];

int minY = (partitionY == y) ? Integer.MAX\_VALUE : nums2[partitionY];

if (maxX <= minY && maxY <= minX) {

if ((x + y) % 2 == 0) {

return (Math.max(maxX, maxY) + Math.min(minX, minY)) / 2.0;

} else {

return Math.max(maxX, maxY);

}

} else if (maxX > minY) {

high = partitionX - 1;

} else {

low = partitionX + 1;

}

}

throw new IllegalArgumentException();

}

**Explanation**:

* Use binary search to partition the arrays and find the median.

**Time Complexity**: O(log(min(m, n))).  
**Space Complexity**: O(1).

**Problem**:Find K Closest Elements (Leetcode 658)

**Description**: Given a sorted integer array arr, two integers k and x, return the k closest integers to x in the array. The result should also be sorted in ascending order.

**Solution**:

import java.util.\*;

public class Solution {

public List<Integer> findClosestElements(int[] arr, int k, int x) {

int left = 0, right = arr.length - k;

// Binary search to find the starting position of the k closest elements

while (left < right) {

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

// Compare the distances of mid and mid + k from x

if (x - arr[mid] > arr[mid + k] - x) {

left = mid + 1; // Move the window to the right

} else {

right = mid; // Move the window to the left

}

}

// Extract the k closest elements

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

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

result.add(arr[i]);

}

return result;

}

}

**Time Complexity:** **O(log n + k)**

* + Binary search takes O(log⁡n)*O*(log*n*), and extracting the k elements takes O(k)*O*(*k*).

**Space Complexity:** **O(k)**

* + The result list stores k elements.

**Category: Bit Manipulation**

**Problem 42: Number of 1 Bits (LeetCode 191)**

**Description**: Count the number of 1 bits in an unsigned integer.

**Solution**:

public int hammingWeight(int n) {

int count = 0;

while (n != 0) {

count += n & 1;

n >>>= 1;

}

return count;

}

**Explanation**:

* Use bitwise operations to count the 1 bits.

**Time Complexity**: O(1) (32 bits).  
**Space Complexity**: O(1).

**Problem 37: Add Digits (LeetCode 258)**

**Description**: Repeatedly add the digits of a number until the result is a single digit.

**Solution**:

public int addDigits(int num) {

return num == 0 ? 0 : 1 + (num - 1) % 9;

}

**Explanation**:

* Use the mathematical property of digital roots.

**Time Complexity**: O(1).  
**Space Complexity**: O(1).

**Category: Recursion and Backtracking**

**Problem**: Generate Parentheses (Leetcode 22)

**Description**: Given an integer n, generate all combinations of well-formed parentheses with n pairs of parentheses.

**Solution**:

import java.util.\*;

public class Solution {

public List<String> generateParenthesis(int n) {

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

backtrack(result, "", 0, 0, n);

return result;

}

private void backtrack(List<String> result, String current, int open, int close, int max) {

// Base case: if the current string is valid, add it to the result

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

result.add(current);

return;

}

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

if (open < max) {

backtrack(result, current + "(", open + 1, close, max);

}

// Add a closing parenthesis if it doesn't exceed the number of opening parentheses

if (close < open) {

backtrack(result, current + ")", open, close + 1, max);

}

}

}

**Time Complexity:** **O(4^n / sqrt(n))**

* + The number of valid combinations is the nth Catalan number, which grows as O(4n/n)*O*(4*n*/*n*​).

**Space Complexity:** **O(4^n / sqrt(n))**

* + The space is used to store the result, which has O(4n/n)*O*(4*n*/*n*​) combinations.

**Problem**: Permutations (Leetcode 46)

**Description**: Given an array nums of distinct integers, return all possible permutations of the array.

**Solution**:

import java.util.\*;

public class Solution {

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

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

backtrack(nums, 0, result);

return result;

}

private void backtrack(int[] nums, int start, List<List<Integer>> result) {

// Base case: if start reaches the end of the array, add the current permutation to the result

if (start == nums.length) {

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

for (int num : nums) {

permutation.add(num);

}

result.add(permutation);

return;

}

// Swap elements to create different permutations

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

swap(nums, start, i); // Swap the current element with the start element

backtrack(nums, start + 1, result); // Recurse for the next position

swap(nums, start, i); // Backtrack (undo the swap)

}

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

**Time Complexity:** **O(n!)**

* + There are n!*n*! permutations for an array of size n.

**Space Complexity:** **O(n!)**

* + The space is used to store the result, which contains n!*n*! permutations.

**Problem**: Permutations II (Leetcode 47)

**Description**: Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations.

**Solution**:

import java.util.\*;

public class Solution {

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

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

Arrays.sort(nums); // Sort the array to group duplicates

backtrack(nums, new boolean[nums.length], new ArrayList<>(), result);

return result;

}

private void backtrack(int[] nums, boolean[] used, List<Integer> current, List<List<Integer>> result) {

// Base case: if the current permutation is complete, add it to the result

if (current.size() == nums.length) {

result.add(new ArrayList<>(current));

return;

}

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

// Skip if the element is already used or if it's a duplicate

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

continue;

}

// Include the current element in the permutation

used[i] = true;

current.add(nums[i]);

// Recurse for the next position

backtrack(nums, used, current, result);

// Backtrack (remove the current element and mark it as unused)

used[i] = false;

current.remove(current.size() - 1);

}

}

}

**Time Complexity:** **O(n!)**

* + There are n!*n*! permutations in the worst case, but pruning reduces the number of recursive calls.

**Space Complexity:** **O(n!)**

* + The space is used to store the result, which contains n!*n*! permutations in the worst case.

**Problem**: Subsets (Leetcode 78)

**Description**: Given an integer array nums of unique elements, return all possible subsets (the power set). The solution set must not contain duplicate subsets.

**Solution**:

import java.util.\*;

public class Solution {

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

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

backtrack(nums, 0, new ArrayList<>(), result);

return result;

}

private void backtrack(int[] nums, int start, List<Integer> current, List<List<Integer>> result) {

// Add the current subset to the result

result.add(new ArrayList<>(current));

// Iterate through the array and generate subsets

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

current.add(nums[i]); // Include the current element

backtrack(nums, i + 1, current, result); // Recurse for the next element

current.remove(current.size() - 1); // Backtrack (exclude the current element)

}

}

}

**Time Complexity:** **O(n \* 2^n)**

* + There are 2n2*n* subsets, and each subset takes O(n)*O*(*n*) time to construct.

**Space Complexity:** **O(n \* 2^n)**

* + The space is used to store the result, which contains 2n2*n* subsets, each of size up to n*n*.

**Problem**: Subsets II (Leetcode 90)

**Description**: Given an integer array nums that may contain duplicates, return all possible subsets (the power set). The solution set must not contain duplicate subsets.

**Solution**:

import java.util.\*;

public class Solution {

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

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

Arrays.sort(nums); // Sort the array to group duplicates

backtrack(nums, 0, new ArrayList<>(), result);

return result;

}

private void backtrack(int[] nums, int start, List<Integer> current, List<List<Integer>> result) {

// Add the current subset to the result

result.add(new ArrayList<>(current));

// Iterate through the array and generate subsets

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

// Skip duplicates to avoid generating duplicate subsets

if (i > start && nums[i] == nums[i - 1]) {

continue;

}

current.add(nums[i]); // Include the current element

backtrack(nums, i + 1, current, result); // Recurse for the next element

current.remove(current.size() - 1); // Backtrack (exclude the current element)

}

}

}

**Time Complexity:** **O(n \* 2^n)**

* + There are 2n2*n* subsets, and each subset takes O(n)*O*(*n*) time to construct.
  + Sorting the array takes O(nlog⁡n)*O*(*n*log*n*), but this is dominated by O(n∗2n)*O*(*n*∗2*n*).

**Space Complexity:** **O(n \* 2^n)**

* + The space is used to store the result, which contains 2n2*n* subsets, each of size up to n*n*.

**Problem 28: Letter Case Permutation (LeetCode 784)**

**Description**: Given a string s, return all possible permutations by changing the case of each letter.

**Solution**:

public List<String> letterCasePermutation(String s) {

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

backtrack(s.toCharArray(), 0, result);

return result;

}

private void backtrack(char[] chars, int index, List<String> result) {

if (index == chars.length) {

result.add(new String(chars));

return;

}

if (Character.isLetter(chars[index])) {

chars[index] = Character.toLowerCase(chars[index]);

backtrack(chars, index + 1, result);

chars[index] = Character.toUpperCase(chars[index]);

}

backtrack(chars, index + 1, result);

}

**Explanation**:

* Use backtracking to generate all permutations by toggling the case of each letter.

**Time Complexity**: O(2^n \* n), where n is the length of the string.  
**Space Complexity**: O(2^n \* n).

**Category: Greedy Algorithms**

**Problem**: Jump Game (Leetcode 55)

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

**Solution**:

public class Solution {

public boolean canJump(int[] nums) {

int maxReach = 0; // Farthest position that can be reached

int n = nums.length;

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

// If the current index is beyond the farthest reachable position, return false

if (i > maxReach) {

return false;

}

// Update the farthest reachable position

maxReach = Math.max(maxReach, i + nums[i]);

// If the farthest reachable position is beyond or at the last index, return true

if (maxReach >= n - 1) {

return true;

}

}

return false;

}

}

**Time Complexity:** **O(n)**

* + The algorithm iterates through the array once.

**Space Complexity:** **O(1)**

* + The algorithm uses only a constant amount of extra space.

**Problem 127: Non-overlapping Intervals (LeetCode 435)**

**Description**: Find the minimum number of intervals to remove to make all intervals non-overlapping.

**Solution**:

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

if (intervals.length == 0) return 0;

Arrays.sort(intervals, (a, b) -> Integer.compare(a[1], b[1]));

int count = 0, end = intervals[0][1];

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

if (intervals[i][0] < end) count++;

else end = intervals[i][1];

}

return count;

}

**Explanation**:

* Sort intervals by end time.
* Remove overlapping intervals greedily.

**Time Complexity**: O(n log n).  
**Space Complexity**: O(1).

**Problem**: Queue Reconstruction by Height (Leetcode 406)

**Description**: Given a list of people represented by pairs (h, k), where h is the height of the person and k is the number of people in front of them who have a height greater than or equal to h, reconstruct the queue so that it satisfies the given constraints.

**Solution**:

import java.util.\*;

public class Solution {

public int[][] reconstructQueue(int[][] people) {

// Sort the people array in descending order of height (h)

// If heights are equal, sort in ascending order of k

Arrays.sort(people, (a, b) -> {

if (a[0] == b[0]) {

return a[1] - b[1]; // Ascending order of k

} else {

return b[0] - a[0]; // Descending order of h

}

});

// Use a list to insert people at the position specified by k

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

for (int[] person : people) {

result.add(person[1], person); // Insert at index k

}

// Convert the list to a 2D array

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

}

}

**Time Complexity:** **O(n^2)**

* + Sorting takes O(nlog⁡n)*O*(*n*log*n*).
  + Inserting each person into the list takes O(n)*O*(*n*) in the worst case, resulting in O(n2)*O*(*n*2).

**Space Complexity:** **O(n)**

* + The space is used to store the result list.

**Problem**: Partition to K Equal Sum Subsets (Leetcode 698)

**Description**: Given an integer array nums and an integer k, determine if it is possible to partition the array into k non-empty subsets whose sums are all equal.

**Solution**:

import java.util.\*;

public class Solution {

public boolean canPartitionKSubsets(int[] nums, int k) {

int totalSum = Arrays.stream(nums).sum(); // Calculate the total sum of the array

if (totalSum % k != 0) {

return false; // If the total sum is not divisible by k, return false

}

int targetSum = totalSum / k; // Target sum for each subset

Arrays.sort(nums); // Sort the array in ascending order

reverseArray(nums); // Reverse the array to sort in descending order

boolean[] used = new boolean[nums.length]; // Track used elements

return backtrack(nums, k, targetSum, 0, 0, used);

}

private boolean backtrack(int[] nums, int k, int targetSum, int currentSum, int start, boolean[] used) {

// Base case: if all subsets are formed, return true

if (k == 0) {

return true;

}

// If the current subset is complete, move to the next subset

if (currentSum == targetSum) {

return backtrack(nums, k - 1, targetSum, 0, 0, used);

}

// Try to form the current subset

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

if (!used[i] && currentSum + nums[i] <= targetSum) {

used[i] = true; // Mark the element as used

if (backtrack(nums, k, targetSum, currentSum + nums[i], i + 1, used)) {

return true; // Recurse for the next element

}

used[i] = false; // Backtrack (unmark the element)

}

}

return false; // No valid subset found

}

private void reverseArray(int[] nums) {

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

while (left < right) {

int temp = nums[left];

nums[left] = nums[right];

nums[right] = temp;

left++;

right--;

}

}

}

**Time Complexity:** **O(k \* 2^n)**

* + In the worst case, the algorithm explores all possible subsets, leading to exponential time complexity.

**Space Complexity:** **O(n)**

* + The space is used for the recursion stack and the used array.

**Category: Advanced Data Structures**

**Problem 54: LRU Cache (LeetCode 146)**

**Description**: Design an LRU (Least Recently Used) cache.

**Solution**:

class LRUCache {

private final int capacity;

private final Map<Integer, Node> map;

private final Node head, tail;

public LRUCache(int capacity) {

this.capacity = capacity;

this.map = new HashMap<>();

this.head = new Node(0, 0);

this.tail = new Node(0, 0);

head.next = tail;

tail.prev = head;

}

public int get(int key) {

if (!map.containsKey(key)) return -1;

Node node = map.get(key);

remove(node);

add(node);

return node.value;

}

public void put(int key, int value) {

if (map.containsKey(key)) {

remove(map.get(key));

}

Node node = new Node(key, value);

map.put(key, node);

add(node);

if (map.size() > capacity) {

Node first = head.next;

remove(first);

map.remove(first.key);

}

}

private void add(Node node) {

Node last = tail.prev;

last.next = node;

node.prev = last;

node.next = tail;

tail.prev = node;

}

private void remove(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

}

private static class Node {

int key, value;

Node prev, next;

Node(int key, int value) {

this.key = key;

this.value = value;

}

}

}

**Explanation**:

* Use a hashmap and a doubly linked list to implement the LRU cache.

**Time Complexity**: O(1) for both get and put.  
**Space Complexity**: O(capacity).

**Problem 55: LFU Cache (LeetCode 460)**

**Description**: Design an LFU (Least Frequently Used) cache.

**Solution**:

class LFUCache {

private final int capacity;

private final Map<Integer, Node> map;

private final Map<Integer, DoublyLinkedList> frequencyMap;

private int minFrequency;

public LFUCache(int capacity) {

this.capacity = capacity;

this.map = new HashMap<>();

this.frequencyMap = new HashMap<>();

this.minFrequency = 0;

}

public int get(int key) {

if (!map.containsKey(key)) return -1;

Node node = map.get(key);

updateFrequency(node);

return node.value;

}

public void put(int key, int value) {

if (capacity == 0) return;

if (map.containsKey(key)) {

Node node = map.get(key);

node.value = value;

updateFrequency(node);

} else {

if (map.size() == capacity) {

DoublyLinkedList list = frequencyMap.get(minFrequency);

Node nodeToRemove = list.removeLast();

map.remove(nodeToRemove.key);

}

Node newNode = new Node(key, value);

map.put(key, newNode);

frequencyMap.computeIfAbsent(1, k -> new DoublyLinkedList()).addFirst(newNode);

minFrequency = 1;

}

}

private void updateFrequency(Node node) {

int frequency = node.frequency;

frequencyMap.get(frequency).remove(node);

if (frequency == minFrequency && frequencyMap.get(frequency).isEmpty()) {

minFrequency++;

}

node.frequency++;

frequencyMap.computeIfAbsent(node.frequency, k -> new DoublyLinkedList()).addFirst(node);

}

private static class Node {

int key, value, frequency;

Node prev, next;

Node(int key, int value) {

this.key = key;

this.value = value;

this.frequency = 1;

}

}

private static class DoublyLinkedList {

Node head, tail;

DoublyLinkedList() {

head = new Node(0, 0);

tail = new Node(0, 0);

head.next = tail;

tail.prev = head;

}

void addFirst(Node node) {

node.next = head.next;

node.prev = head;

head.next.prev = node;

head.next = node;

}

void remove(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

}

Node removeLast() {

if (isEmpty()) return null;

Node node = tail.prev;

remove(node);

return node;

}

boolean isEmpty() {

return head.next == tail;

}

}

}

**Explanation**:

* Use a hashmap to store nodes and another hashmap to store frequency lists.
* Update the frequency of nodes on each access.

**Time Complexity**: O(1) for both get and put.  
**Space Complexity**: O(capacity).

**Problem**: Implement Trie (Prefix Tree) (Leetcode 208)

**Description**:

**Solution**:

class TrieNode {

public boolean isEnd; // Marks the end of a word

public TrieNode[] children; // Array of children nodes

public TrieNode() {

isEnd = false;

children = new TrieNode[26]; // Assuming lowercase English letters

}

}

class Trie {

private TrieNode root;

public Trie() {

root = new TrieNode(); // Initialize the root node

}

public void insert(String word) {

TrieNode node = root;

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

int index = c - 'a'; // Calculate the index for the character

if (node.children[index] == null) {

node.children[index] = new TrieNode(); // Create a new node if it doesn't exist

}

node = node.children[index]; // Move to the child node

}

node.isEnd = true; // Mark the end of the word

}

public boolean search(String word) {

TrieNode node = searchPrefix(word);

return node != null && node.isEnd; // Check if the node exists and marks the end of a word

}

public boolean startsWith(String prefix) {

TrieNode node = searchPrefix(prefix);

return node != null; // Check if the node exists

}

private TrieNode searchPrefix(String prefix) {

TrieNode node = root;

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

int index = c - 'a'; // Calculate the index for the character

if (node.children[index] == null) {

return null; // Prefix does not exist

}

node = node.children[index]; // Move to the child node

}

return node; // Return the node corresponding to the prefix

}

}

* **Time Complexity:**
  + insert(word): **O(m)**, where m is the length of the word.
  + search(word): **O(m)**, where m is the length of the word.
  + startsWith(prefix): **O(m)**, where m is the length of the prefix.
* **Space Complexity:** **O(n \* m)**
  + n is the number of words, and m is the average length of the words.

**Problem 7: Pascal's Triangle (LeetCode 118)**

**Description**: Given an integer numRows, return the first numRows of Pascal's triangle.

**Solution**:

public List<List<Integer>> generate(int numRows) {

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

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

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

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

if (j == 0 || j == i) {

row.add(1);

} else {

row.add(triangle.get(i - 1).get(j - 1) + triangle.get(i - 1).get(j));

}

}

triangle.add(row);

}

return triangle;

}

**Explanation**:

* Each row starts and ends with 1.
* Middle elements are the sum of the two elements directly above them.

**Time Complexity**: O(n^2), where n is the number of rows.  
**Space Complexity**: O(n^2) (output space).

**Problem 9: Rectangle Overlap (LeetCode 836)**

**Description**: Given two rectangles, determine if they overlap.

**Solution**:

public boolean isRectangleOverlap(int[] rec1, int[] rec2) {

return !(rec1[2] <= rec2[0] || rec1[0] >= rec2[2] || rec1[3] <= rec2[1] || rec1[1] >= rec2[3]);

}

**Explanation**:

* Check if one rectangle is to the left, right, above, or below the other.

**Time Complexity**: O(1).  
**Space Complexity**: O(1).

**Problem 14: Maximum Product of Three Numbers (LeetCode 628)**

**Description**: Given an integer array, find three numbers whose product is maximum.

**Solution**:

public int maximumProduct(int[] nums) {

Arrays.sort(nums);

int n = nums.length;

return Math.max(nums[n - 1] \* nums[n - 2] \* nums[n - 3], nums[0] \* nums[1] \* nums[n - 1]);

}

**Explanation**:

* Sort the array and consider two cases:
  1. The product of the three largest numbers.
  2. The product of the two smallest (negative) numbers and the largest number.

**Time Complexity**: O(n log n), due to sorting.  
**Space Complexity**: O(1).

**Problem 16: Island Perimeter (LeetCode 463)**

**Description**: Given a 2D grid of 1s (land) and 0s (water), calculate the perimeter of the island.

**Solution**:

public int islandPerimeter(int[][] grid) {

int perimeter = 0;

int rows = grid.length;

int cols = grid[0].length;

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

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

if (grid[i][j] == 1) {

perimeter += 4;

if (i > 0 && grid[i - 1][j] == 1) perimeter -= 2;

if (j > 0 && grid[i][j - 1] == 1) perimeter -= 2;

}

}

}

return perimeter;

}

**Explanation**:

* For each land cell, add 4 to the perimeter.
* Subtract 2 for each shared edge with adjacent land cells.

**Time Complexity**: O(n \* m), where n and m are the dimensions of the grid.  
**Space Complexity**: O(1).

**Problem 24: Isomorphic Strings (LeetCode 205)**

**Description**: Given two strings s and t, determine if they are isomorphic.

**Solution**:

public boolean isIsomorphic(String s, String t) {

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

Map<Character, Character> map = new HashMap<>();

Set<Character> used = new HashSet<>();

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

char c1 = s.charAt(i);

char c2 = t.charAt(i);

if (map.containsKey(c1)) {

if (map.get(c1) != c2) return false;

} else {

if (used.contains(c2)) return false;

map.put(c1, c2);

used.add(c2);

}

}

return true;

}

**Explanation**:

* Use a hashmap to map characters from s to t.
* Use a set to ensure no two characters in s map to the same character in t.

**Time Complexity**: O(n), where n is the length of the strings.  
**Space Complexity**: O(1) (fixed-size hashmap and set).

**Problem 30: Sqrt(x) (LeetCode 69)**

**Description**: Compute the integer square root of a non-negative integer x.

**Solution**:

public int mySqrt(int x) {

if (x == 0) return 0;

int left = 1, right = x;

while (left <= right) {

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

if (mid > x / mid) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return right;

}

**Explanation**:

* Use binary search to find the integer square root.

**Time Complexity**: O(log x).  
**Space Complexity**: O(1).

**Problem 46: Missing Number (LeetCode 268)**

**Description**: Find the missing number in an array containing numbers from 0 to n.

**Solution**:

public int missingNumber(int[] nums) {

int n = nums.length;

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

int actualSum = 0;

for (int num : nums) actualSum += num;

return expectedSum - actualSum;

}

**Explanation**:

* Calculate the expected sum and subtract the actual sum to find the missing number.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 48: Fizz Buzz (LeetCode 412)**

**Description**: Return a string array where multiples of 3 are "Fizz", multiples of 5 are "Buzz", and multiples of both are "FizzBuzz".

**Solution**:

public List<String> fizzBuzz(int n) {

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

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

if (i % 15 == 0) result.add("FizzBuzz");

else if (i % 3 == 0) result.add("Fizz");

else if (i % 5 == 0) result.add("Buzz");

else result.add(Integer.toString(i));

}

return result;

}

**Explanation**:

* Use conditional statements to check for multiples of 3, 5, and 15.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 57: Zuma Game (LeetCode 488)**

**Description**: Simulate the Zuma game to find the minimum number of balls to remove.

**Solution**:

public int findMinStep(String board, String hand) {

int[] handCount = new int[26];

for (char c : hand.toCharArray()) handCount[c - 'A']++;

return dfs(board, handCount);

}

private int dfs(String board, int[] handCount) {

if (board.isEmpty()) return 0;

int result = Integer.MAX\_VALUE;

int i = 0;

while (i < board.length()) {

int j = i;

while (j < board.length() && board.charAt(j) == board.charAt(i)) j++;

char color = board.charAt(i);

int needed = 3 - (j - i);

if (handCount[color - 'A'] >= needed) {

handCount[color - 'A'] -= needed;

String newBoard = removeConsecutive(board.substring(0, i) + board.substring(j));

int subResult = dfs(newBoard, handCount);

if (subResult != -1) result = Math.min(result, subResult + needed);

handCount[color - 'A'] += needed;

}

i = j;

}

return result == Integer.MAX\_VALUE ? -1 : result;

}

private String removeConsecutive(String board) {

int i = 0;

while (i < board.length()) {

int j = i;

while (j < board.length() && board.charAt(j) == board.charAt(i)) j++;

if (j - i >= 3) {

board = board.substring(0, i) + board.substring(j);

i = 0;

} else {

i++;

}

}

return board;

}

**Explanation**:

* Use DFS to simulate the game and backtrack to find the minimum steps.

**Time Complexity**: O(n!) (exponential).  
**Space Complexity**: O(n).

**Problem 61: Shortest Palindrome (LeetCode 214)**

**Description**: Given a string s, add the minimum number of characters in front of it to make it a palindrome.

**Solution**:

public String shortestPalindrome(String s) {

String reversed = new StringBuilder(s).reverse().toString();

String combined = s + "#" + reversed;

int[] lps = computeLPS(combined);

return reversed.substring(0, s.length() - lps[lps.length - 1]) + s;

}

private int[] computeLPS(String s) {

int[] lps = new int[s.length()];

int len = 0, i = 1;

while (i < s.length()) {

if (s.charAt(i) == s.charAt(len)) {

lps[i++] = ++len;

} else {

if (len != 0) len = lps[len - 1];

else lps[i++] = 0;

}

}

return lps;

}

**Explanation**:

* Use the Knuth-Morris-Pratt (KMP) algorithm to compute the longest prefix suffix (LPS) array.
* The LPS array helps find the longest prefix of s that is also a suffix of its reverse.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 63: Integer to English Words (LeetCode 273)**

**Description**: Convert a non-negative integer to its English words representation.

**Solution**:

private final String[] LESS\_THAN\_20 = {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};

private final String[] TENS = {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};

private final String[] THOUSANDS = {"", "Thousand", "Million", "Billion"};

public String numberToWords(int num) {

if (num == 0) return "Zero";

String words = "";

int i = 0;

while (num > 0) {

if (num % 1000 != 0) {

words = helper(num % 1000) + THOUSANDS[i] + " " + words;

}

num /= 1000;

i++;

}

return words.trim();

}

private String helper(int num) {

if (num == 0) return "";

if (num < 20) return LESS\_THAN\_20[num] + " ";

if (num < 100) return TENS[num / 10] + " " + helper(num % 10);

return LESS\_THAN\_20[num / 100] + " Hundred " + helper(num % 100);

}

**Explanation**:

* Break the number into chunks of thousands and convert each chunk to words.

**Time Complexity**: O(1) (since the number is at most 2^31 - 1).  
**Space Complexity**: O(1).

**Problem 68: Regular Expression Matching (LeetCode 10)**

**Description**: Implement regular expression matching with support for . and \*.

**Solution**:

public boolean isMatch(String s, String p) {

int m = s.length(), n = p.length();

boolean[][] dp = new boolean[m + 1][n + 1];

dp[0][0] = true;

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

if (p.charAt(j - 1) == '\*') {

dp[0][j] = dp[0][j - 2];

}

}

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

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

if (p.charAt(j - 1) == '.' || p.charAt(j - 1) == s.charAt(i - 1)) {

dp[i][j] = dp[i - 1][j - 1];

} else if (p.charAt(j - 1) == '\*') {

dp[i][j] = dp[i][j - 2];

if (p.charAt(j - 2) == '.' || p.charAt(j - 2) == s.charAt(i - 1)) {

dp[i][j] |= dp[i - 1][j];

}

}

}

}

return dp[m][n];

}

**Explanation**:

* Use dynamic programming to handle . and \* in the pattern.

**Time Complexity**: O(m \* n).  
**Space Complexity**: O(m \* n).

**Problem 69: Split Array Largest Sum (LeetCode 410)**

**Description**: Split an array into m subarrays such that the largest sum among the subarrays is minimized.

**Solution**:

public int splitArray(int[] nums, int m) {

int left = 0, right = 0;

for (int num : nums) {

left = Math.max(left, num);

right += num;

}

while (left < right) {

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

if (canSplit(nums, m, mid)) {

right = mid;

} else {

left = mid + 1;

}

}

return left;

}

private boolean canSplit(int[] nums, int m, int maxSum) {

int sum = 0, count = 1;

for (int num : nums) {

sum += num;

if (sum > maxSum) {

sum = num;

count++;

if (count > m) return false;

}

}

return true;

}

**Explanation**:

* Use binary search to find the minimum possible largest sum.

**Time Complexity**: O(n \* log(sum(nums))).  
**Space Complexity**: O(1).

**Problem 72: First Missing Positive (LeetCode 41)**

**Description**: Find the smallest missing positive integer in an unsorted array.

**Solution**:

public int firstMissingPositive(int[] nums) {

int n = nums.length;

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

while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] != nums[i]) {

swap(nums, i, nums[i] - 1);

}

}

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

if (nums[i] != i + 1) return i + 1;

}

return n + 1;

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

**Explanation**:

* Use cyclic sort to place each number in its correct position.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 73: Minimum Window Substring (LeetCode 76)**

**Description**: Find the minimum window in s that contains all characters of t.

**Solution**:

public String minWindow(String s, String t) {

int[] map = new int[128];

for (char c : t.toCharArray()) map[c]++;

int left = 0, right = 0, minLen = Integer.MAX\_VALUE, minStart = 0, count = t.length();

while (right < s.length()) {

if (map[s.charAt(right++)]-- > 0) count--;

while (count == 0) {

if (right - left < minLen) {

minLen = right - left;

minStart = left;

}

if (map[s.charAt(left++)]++ == 0) count++;

}

}

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

}

**Explanation**:

* Use a sliding window to track the minimum window containing all characters of t.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 76: Kill Process (LeetCode 582)**

**Description**: Given a list of processes and their parent-child relationships, return all processes that will be killed when a specific process is terminated.

**Solution**:

public List<Integer> killProcess(List<Integer> pid, List<Integer> ppid, int kill) {

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

for (int i = 0; i < ppid.size(); i++) {

map.computeIfAbsent(ppid.get(i), k -> new ArrayList<>()).add(pid.get(i));

}

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

Queue<Integer> queue = new LinkedList<>();

queue.add(kill);

while (!queue.isEmpty()) {

int process = queue.poll();

result.add(process);

if (map.containsKey(process)) {

queue.addAll(map.get(process));

}

}

return result;

}

**Explanation**:

* Use a hashmap to store parent-child relationships.
* Perform BFS to traverse and collect all child processes.

**Time Complexity**: O(n).  
**Space Complexity**: O(n).

**Problem 77: Candy Crush (LeetCode 723)**

**Description**: Simulate the Candy Crush game by crushing candies and dropping new ones.

**Solution**:

public int[][] candyCrush(int[][] board) {

int m = board.length, n = board[0].length;

boolean stable = false;

while (!stable) {

stable = true;

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

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

int val = Math.abs(board[i][j]);

if (val != 0 && val == Math.abs(board[i][j + 1]) && val == Math.abs(board[i][j + 2])) {

board[i][j] = board[i][j + 1] = board[i][j + 2] = -val;

stable = false;

}

}

}

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

for (int i = 0; i < m - 2; i++) {

int val = Math.abs(board[i][j]);

if (val != 0 && val == Math.abs(board[i + 1][j]) && val == Math.abs(board[i + 2][j])) {

board[i][j] = board[i + 1][j] = board[i + 2][j] = -val;

stable = false;

}

}

}

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

int index = m - 1;

for (int i = m - 1; i >= 0; i--) {

if (board[i][j] > 0) {

board[index--][j] = board[i][j];

}

}

while (index >= 0) {

board[index--][j] = 0;

}

}

}

return board;

}

**Explanation**:

* Mark candies to be crushed by setting them to negative values.
* Drop candies to fill empty spaces.

**Time Complexity**: O((m \* n)^2) (worst case).  
**Space Complexity**: O(1).

**Problem 87: Brick Wall (LeetCode 554)**

**Description**: Find the minimum number of bricks to cross to go from the top to the bottom of a brick wall.

**Solution**:

public int leastBricks(List<List<Integer>> wall) {

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

for (List<Integer> row : wall) {

int sum = 0;

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

sum += row.get(i);

map.put(sum, map.getOrDefault(sum, 0) + 1);

}

}

int max = 0;

for (int count : map.values()) {

max = Math.max(max, count);

}

return wall.size() - max;

}

**Explanation**:

* Use a hashmap to count the frequency of brick edges at each position.
* The minimum number of bricks to cross is the total number of rows minus the maximum frequency.

**Time Complexity**: O(n), where n is the total number of bricks.  
**Space Complexity**: O(n).

**Problem 90: Populating Next Right Pointers in Each Node (LeetCode 116)**

**Description**: Populate each next pointer to point to its next right node in a perfect binary tree.

**Solution**:

public Node connect(Node root) {

if (root == null) return null;

Node leftmost = root;

while (leftmost.left != null) {

Node current = leftmost;

while (current != null) {

current.left.next = current.right;

if (current.next != null) {

current.right.next = current.next.left;

}

current = current.next;

}

leftmost = leftmost.left;

}

return root;

}

**Explanation**:

* Use level-order traversal to connect nodes at the same level.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 94: Populating Next Right Pointers in Each Node II (LeetCode 117)**

**Description**: Populate each next pointer to point to its next right node in a binary tree (not necessarily perfect).

**Solution**:

public Node connect(Node root) {

Node dummy = new Node(0);

Node current = dummy;

Node head = root;

while (root != null) {

if (root.left != null) {

current.next = root.left;

current = current.next;

}

if (root.right != null) {

current.next = root.right;

current = current.next;

}

root = root.next;

if (root == null) {

root = dummy.next;

dummy.next = null;

current = dummy;

}

}

return head;

}

**Explanation**:

* Use a dummy node to connect nodes at the same level.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 95: Meeting Rooms II (LeetCode 253)**

**Description**: Find the minimum number of conference rooms required.

**Solution**:

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

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

PriorityQueue<Integer> pq = new PriorityQueue<>();

for (int[] interval : intervals) {

if (!pq.isEmpty() && pq.peek() <= interval[0]) {

pq.poll();

}

pq.offer(interval[1]);

}

return pq.size();

}

**Explanation**:

* Sort intervals by start time.
* Use a priority queue to track the end times of meetings.

**Time Complexity**: O(n log n).  
**Space Complexity**: O(n).

**Problem** 99: Add Two Numbers (LeetCode 2)

**Description**: Add two numbers represented as linked lists.

**Solution**:

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(0);

ListNode current = dummy;

int carry = 0;

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

int sum = carry;

if (l1 != null) {

sum += l1.val;

l1 = l1.next;

}

if (l2 != null) {

sum += l2.val;

l2 = l2.next;

}

carry = sum / 10;

current.next = new ListNode(sum % 10);

current = current.next;

}

return dummy.next;

}

**Explanation**:

* Use a dummy node to simplify the addition process.
* Handle carry and construct the result linked list.

**Time Complexity**: O(max(m, n)).  
**Space Complexity**: O(max(m, n)).

**Problem** 102: Evaluate Division (LeetCode 399)

**Description**: Evaluate division queries based on given equations.

**Solution**:

public double[] calcEquation(List<List<String>> equations, double[] values, List<List<String>> queries) {

Map<String, Map<String, Double>> graph = new HashMap<>();

for (int i = 0; i < equations.size(); i++) {

String u = equations.get(i).get(0), v = equations.get(i).get(1);

graph.computeIfAbsent(u, k -> new HashMap<>()).put(v, values[i]);

graph.computeIfAbsent(v, k -> new HashMap<>()).put(u, 1 / values[i]);

}

double[] result = new double[queries.size()];

for (int i = 0; i < queries.size(); i++) {

result[i] = dfs(graph, queries.get(i).get(0), queries.get(i).get(1), new HashSet<>());

}

return result;

}

private double dfs(Map<String, Map<String, Double>> graph, String u, String v, Set<String> visited) {

if (!graph.containsKey(u) || !graph.containsKey(v)) return -1.0;

if (u.equals(v)) return 1.0;

visited.add(u);

for (Map.Entry<String, Double> entry : graph.get(u).entrySet()) {

String next = entry.getKey();

if (visited.contains(next)) continue;

double result = dfs(graph, next, v, visited);

if (result != -1.0) return entry.getValue() \* result;

}

return -1.0;

}

**Explanation**:

* Build a graph of equations and values.
* Use DFS to evaluate queries.

**Time Complexity**: O(n \* q), where n is the number of equations and q is the number of queries.  
**Space Complexity**: O(n).

**Problem** 105: Valid Triangle Number (LeetCode 611)

**Description**: Count the number of valid triangles that can be formed from an array.

**Solution**:

public int triangleNumber(int[] nums) {

Arrays.sort(nums);

int count = 0;

for (int i = nums.length - 1; i >= 2; i--) {

int left = 0, right = i - 1;

while (left < right) {

if (nums[left] + nums[right] > nums[i]) {

count += right - left;

right--;

} else {

left++;

}

}

}

return count;

}

**Explanation**:

* Sort the array and use two pointers to count valid triangles.

**Time Complexity**: O(n^2).  
**Space Complexity**: O(1).

**Problem 110: Lowest Common Ancestor of a Binary Tree (LeetCode 236)**

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

**Solution**:

java

Copy

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

if (root == null || root == p || root == q) return root;

TreeNode left = lowestCommonAncestor(root.left, p, q);

TreeNode right = lowestCommonAncestor(root.right, p, q);

if (left != null && right != null) return root;

return left != null ? left : right;

}

**Explanation**:

* Use recursion to find the LCA.
* If both nodes are found in left and right subtrees, the current node is the LCA.

**Time Complexity**: O(n).  
**Space Complexity**: O(h).

**Problem 111: 3Sum (LeetCode 15)**

**Description**: Find all unique triplets in an array that sum to zero.

**Solution**:

java

Copy

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

Arrays.sort(nums);

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

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

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

int left = i + 1, right = nums.length - 1;

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

if (sum == 0) {

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

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

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

left++;

right--;

} else if (sum < 0) {

left++;

} else {

right--;

}

}

}

return result;

}

**Explanation**:

* Sort the array and use two pointers to find triplets that sum to zero.

**Time Complexity**: O(n^2).  
**Space Complexity**: O(1).

**Problem 112: Sentence Similarity II (LeetCode 737)**

**Description**: Check if two sentences are similar based on word similarity relationships.

**Solution**:

public boolean areSentencesSimilarTwo(String[] words1, String[] words2, List<List<String>> pairs) {

if (words1.length != words2.length) return false;

Map<String, String> parent = new HashMap<>();

for (List<String> pair : pairs) {

String word1 = pair.get(0), word2 = pair.get(1);

if (!parent.containsKey(word1)) parent.put(word1, word1);

if (!parent.containsKey(word2)) parent.put(word2, word2);

union(parent, word1, word2);

}

for (int i = 0; i < words1.length; i++) {

if (!find(parent, words1[i]).equals(find(parent, words2[i]))) return false;

}

return true;

}

private String find(Map<String, String> parent, String word) {

if (!parent.containsKey(word)) return word;

while (!word.equals(parent.get(word))) {

parent.put(word, parent.get(parent.get(word)));

word = parent.get(word);

}

return word;

}

private void union(Map<String, String> parent, String word1, String word2) {

String root1 = find(parent, word1), root2 = find(parent, word2);

if (!root1.equals(root2)) parent.put(root1, root2);

}

**Explanation**:

* Use Union-Find to group similar words.
* Check if corresponding words in the sentences belong to the same group.

**Time Complexity**: O(n + m), where n is the number of words and m is the number of pairs.  
**Space Complexity**: O(m).

**Problem 114: Triangle (LeetCode 120)**

**Description**: Find the minimum path sum from the top to the bottom of a triangle.

**Solution**:

public int minimumTotal(List<List<Integer>> triangle) {

int n = triangle.size();

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

for (int i = n - 1; i >= 0; i--) {

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

dp[j] = Math.min(dp[j], dp[j + 1]) + triangle.get(i).get(j);

}

}

return dp[0];

}

**Explanation**:

* Use dynamic programming to compute the minimum path sum from the bottom up.

**Time Complexity**: O(n^2).  
**Space Complexity**: O(n).

**Problem 117: Next Greater Element III (LeetCode 556)**

**Description**: Find the smallest integer greater than n with the same digits.

**Solution**:

public int nextGreaterElement(int n) {

char[] digits = String.valueOf(n).toCharArray();

int i = digits.length - 2;

while (i >= 0 && digits[i] >= digits[i + 1]) i--;

if (i < 0) return -1;

int j = digits.length - 1;

while (j >= 0 && digits[j] <= digits[i]) j--;

swap(digits, i, j);

reverse(digits, i + 1, digits.length - 1);

try {

return Integer.parseInt(new String(digits));

} catch (NumberFormatException e) {

return -1;

}

}

private void swap(char[] digits, int i, int j) {

char temp = digits[i];

digits[i] = digits[j];

digits[j] = temp;

}

private void reverse(char[] digits, int i, int j) {

while (i < j) swap(digits, i++, j--);

}

**Explanation**:

* Find the first decreasing digit from the right.
* Swap it with the smallest greater digit to its right.
* Reverse the suffix to get the smallest number.

**Time Complexity**: O(d), where d is the number of digits.  
**Space Complexity**: O(d).

**Problem 120: Permutation in String (LeetCode 567)**

**Description**: Check if one string contains a permutation of another string.

**Solution**:

public boolean checkInclusion(String s1, String s2) {

int[] count = new int[26];

for (char c : s1.toCharArray()) count[c - 'a']++;

int left = 0, right = 0, len = s1.length();

while (right < s2.length()) {

if (count[s2.charAt(right++) - 'a']-- > 0) len--;

if (len == 0) return true;

if (right - left == s1.length() && count[s2.charAt(left++) - 'a']++ >= 0) len++;

}

return false;

}

**Explanation**:

* Use a sliding window to check if a substring of s2 has the same character count as s1.

**Time Complexity**: O(n).  
**Space Complexity**: O(1).

**Problem 524: Longest Word in Dictionary through Deleting (LeetCode 524)**

**Description**: Find the longest word in a dictionary that can be formed by deleting characters from a string.

**Solution**:

public String findLongestWord(String s, List<String> d) {

String result = "";

for (String word : d) {

if (isSubsequence(word, s)) {

if (word.length() > result.length() || (word.length() == result.length() && word.compareTo(result) < 0)) {

result = word;

}

}

}

return result;

}

private boolean isSubsequence(String word, String s) {

int i = 0, j = 0;

while (i < word.length() && j < s.length()) {

if (word.charAt(i) == s.charAt(j)) i++;

j++;

}

return i == word.length();

}

**Explanation**:

* Check if each word in the dictionary is a subsequence of s.
* Track the longest valid word.

**Time Complexity**: O(n \* m), where n is the length of s and m is the number of words.  
**Space Complexity**: O(1).