



## WEEK 1 - Data Structures and Algorithms

Leet-Code Documentation

SUBMITTED TO

PEOPLE TECH GROUP INC

GOVIND SHARMA

RECRUITMENT LEAD

SUBMITTED BY

SANJEEV REDDY SIRIPINANE

## Leet-Code Problems:

### All My Submissions

Time Submitted	Question	Status
10 hours, 20 minutes ago	<a href="#">Reverse Nodes in k-Group</a>	Accepted
11 hours, 17 minutes ago	<a href="#">Longest Palindromic Substring</a>	Accepted
20 hours, 24 minutes ago	<a href="#">Shortest Subarray to be Removed to Make Array Sorted</a>	Accepted
21 hours, 32 minutes ago	<a href="#">Reverse Linked List</a>	Accepted
22 hours, 9 minutes ago	<a href="#">Remove Linked List Elements</a>	Accepted
22 hours, 51 minutes ago	<a href="#">Pascal's Triangle</a>	Accepted
23 hours, 33 minutes ago	<a href="#">Path Sum</a>	Accepted
1 day, 1 hour ago	<a href="#">Intersection of Two Linked Lists</a>	Accepted
1 day, 9 hours ago	<a href="#">Intersection of Two Linked Lists</a>	Accepted
1 day, 10 hours ago	<a href="#">Convert Sorted Array to Binary Search Tree</a>	Accepted
1 day, 12 hours ago	<a href="#">Minimum Depth of Binary Tree</a>	Accepted
1 day, 13 hours ago	<a href="#">Maximum Depth of Binary Tree</a>	Accepted
1 day, 13 hours ago	<a href="#">Binary Tree Inorder Traversal</a>	Accepted

1 day, 18 hours ago	<a href="#">Remove Duplicates from Sorted List</a>	Accepted
1 day, 19 hours ago	<a href="#">Search Insert Position</a>	Accepted
1 day, 20 hours ago	<a href="#">Longest Common Prefix</a>	Accepted
1 day, 23 hours ago	<a href="#">Remove Element</a>	Accepted
2 days ago	<a href="#">Palindrome Number</a>	Accepted
2 days, 2 hours ago	<a href="#">Remove Duplicates from Sorted Array</a>	Accepted
2 days, 3 hours ago	<a href="#">Merge Two Sorted Lists</a>	Accepted

Time Submitted	Question	Status
2 days, 19 hours ago	<a href="#">Linked List Cycle</a>	Accepted
3 days, 1 hour ago	<a href="#">Final Array State After K Multiplication Operations I</a>	Accepted
3 days, 8 hours ago	<a href="#">Climbing Stairs</a>	Accepted
3 days, 11 hours ago	<a href="#">Min Cost Climbing Stairs</a>	Accepted
3 days, 12 hours ago	<a href="#">Implement Stack using Queues</a>	Accepted
3 days, 12 hours ago	<a href="#">Implement Queue using Stacks</a>	Accepted

## Implementing Stack Using Queues:

The screenshot shows the LeetCode interface for problem 225, "Implement Stack using Queues". The problem is categorized as "Easy". The description states: "Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty)." The code editor shows a Java solution using two linked lists, q1 and q2, to simulate a stack. The test case shows an input sequence of operations: ["MyStack", "push", "push", "top", "pop", "empty"], which results in an output of [null, null, null, 2, 2, false].

**Testcase** | **Test Result**

**Accepted** Runtime: 0 ms

**Case 1**

**Input**

```
["MyStack", "push", "push", "top", "pop", "empty"]
```

**Output**

```
[null, null, null, 2, 2, false]
```

**Expected**

```
[null, null, null, 2, 2, false]
```

**225. Implement Stack using Queues**

**Easy** | **Topics** | **Companies**

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the `MyStack` class:

- `void push(int x)` Pushes element x to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean isEmpty()` Returns true if the stack is empty.

```
1 import java.util.LinkedList;
2 import java.util.Queue;
3
4 class MyStack {
5
6     private Queue<Integer> q1;
7     private Queue<Integer> q2;
8     public MyStack() {
9         q1 = new LinkedList<>();
10        q2 = new LinkedList<>();
11    }
12    public void push(int x) {
13        q1.add(x);
14    }
15    public int pop() {
16        while (q1.size() > 1) {
17            q2.add(q1.remove());
18        }
19        int removed_item = q1.remove();
20        Queue<Integer> temp = q1;
21        q1 = q2;
22        q2 = temp;
23        return removed_item;
24    }
25 }
```

Ln 1, Col 1 | Saved

**Run** **Submit**

## Description of the Procedures:

- 1. Push:** We only update q1 with the new element.
- 2. Pop:** We move every element from q1 to q2, with the exception of the final one. We then delete and return the top of the stack, which is the final thing left in q1. To make sure q1 has the items of the current stack, we then switch q1 and q2.
- 3. Top:** Like pop, but with the final piece retained in q2 and returned as the top of the stack. We switch q1 and q2 once again.
- 4. Empty:** Just see if the stack is empty by checking if q1 is empty.

This method uses just the common queue operations to guarantee the stack's LIFO behavior.

**Challenge:** Implementing a stack using two queues can be challenging because stacks and queues work in opposite ways: stacks use last-in-first-out (LIFO) ordering, while queues use first-in-first-out (FIFO) ordering. To mimic stack behavior, we need to reverse the order of elements when performing "pop" or "top" actions. This means we have to move all elements except the last one from one queue (q1) to another queue (q2), then swap the roles of the two queues. This approach allows the stack to function as intended, but it's less efficient, as each pop or top operation requires moving elements around and adds extra steps.

## Implementation of Queues using Stack:

The screenshot displays the LeetCode interface for the problem "232. Implement Queue using Stacks". The problem is marked as "Solved" and "Easy". The description states: "Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).". The input sequence is ["MyQueue", "push", "push", "peek", "pop", "empty"] and the output sequence is [null, null, null, 1, 1, false]. The Java solution uses two stacks, s1 and s2, to implement the queue operations. The code is as follows:

```
1 import java.util.Stack;
2
3 class MyQueue {
4
5     private Stack<Integer> s1;
6     private Stack<Integer> s2;
7
8     public MyQueue() {
9         s1 = new Stack<>();
10        s2 = new Stack<>();
11    }
12
13    public void push(int x) {
14        s1.push(x);
15    }
16
17    public int pop() {
18        if (s2.isEmpty()) {
19            while (!s1.isEmpty()) {
20                s2.push(s1.pop());
21            }
22        }
23        return s2.pop();
24    }
25
26    public int peek() {
27        if (s2.isEmpty()) {
28            while (!s1.isEmpty()) {
29                s2.push(s1.pop());
30            }
31        }
32        return s2.peek();
33    }
34
35    public boolean empty() {
36        return s1.isEmpty() && s2.isEmpty();
37    }
38}
```

### Methods Explanation:

1. **Push:** Just updated S1 with the new element.

2. **Pop:**

- Move every element from s1 to s2 if s2 is empty.
- Take out s2's top element, which is the queue's front entry.

3. **Sneak:**

- Move every element from s1 to s2 if s2 is empty.
- Return the first entry in the queue, which is the top element of s2.

4. **Empty:** Verify if S1 and S2 are both empty.

### Walkthrough Example:

1. **Push 1 Push 2:** We induce s1 to contain 1 and 2. At this point, s2 is empty and s1 has [1, 2].

2. **Peek:** We shift everything from s1 to s2 since s2 is empty. At this point, s2 has [2, 1] while s1 is empty. Peek returns 1 since the top of s2 is 1.

3. **Pop:** We eliminate 1 as s2's top element. S2 now has [2].

4. **Empty:** Return true if both stacks are empty, and false otherwise. Two stacks are used in this design to effectively accomplish queue behavior.

**Challenges Faced:** The main challenge when building a queue with two stacks is maintaining the correct order of elements. Stacks follow a "last in, first out" (LIFO) approach, whereas queues require "first in, first out" (FIFO) ordering. To achieve this with stacks, every time we want to remove or view an element from the queue, we must transfer all items from one stack to another, reversing

the order. This process can become slow, especially with many elements, as each operation takes time to shuffle items around, making it less efficient compared to a standard queue.

## Min Cost Climbing Stairs:

The screenshot displays the LeetCode interface for the problem '746. Min Cost Climbing Stairs'. On the left, the 'Testcase' tab shows 'Accepted' status with a runtime of 0 ms. The 'Input' is `cost = [10, 15, 20]` and the 'Output' is `15`. The 'Expected' output is also `15`. The 'Description' tab shows the problem statement: 'You are given an integer array cost where cost[i] is the cost of i<sup>th</sup> step on a staircase. Once you pay the cost, you can either climb one or two steps. You can either start from the step with index 0, or the step with index 1. Return the minimum cost to reach the top of the floor.' An example is provided: 'Input: cost = [10, 15, 20], Output: 15, Explanation: You will start at index 1. - Pay 15 and climb two steps to reach the top.' The 'Code' tab shows a Java solution: 

```
1 class Solution {
2     public int minCostClimbingStairs(int[] cost) {
3         int n = cost.length;
4
5         int step_1 = cost[0];
6         int step_2 = cost[1];
7
8         for (int i = 2; i < n; i++) {
9             int current = cost[i] + Math.min(step_1, step_2);
10            step_1 = step_2;
11            step_2 = current;
12        }
13        return Math.min(step_1, step_2);
14    }
15 }
16
```

### Explanation:

1. First, we initialize step\_1 and step\_2 using the first two steps' costs.
2. We determine the minimal cost to achieve each step (current) between 2 and n-1.
3. As we proceed to the following phase, update the step\_1 and step\_2.
4. Lastly, reimburse the lowest of the final two expenses (the step\_1 and the step\_2).

### Example

For cost = [10, 15, 20]:

- step\_1 = 10
- step\_2 = 15
- For i = 2: current = cost[2] + Math.min(step\_1, step\_2) = 20 + 10 = 30
- The answer is Math.min(15, 30) = 15.

**Challenge Faced:** Observing the lowest expenses as we proceed through each phase is the primary problem with this strategy. To get to the current stage, we must always select the least expensive option, but we must also accurately update the prices as we proceed. This may get challenging, particularly when working with longer expense lists or when there is significant variation in the costs. We must be careful not to lose the minimal cost while updating the numbers since it is simple to make mistakes when keeping track of the previous and current stages.

## Climbing Stairs:

The screenshot shows the LeetCode interface for the problem "70. Climbing Stairs". The left sidebar displays the "Test Result" tab, indicating the solution is "Accepted" with a runtime of 0 ms. The main content area shows the problem description: "You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?". It includes two examples: Example 1 with input n=2 and output 2, and Example 2 with input n=3 and output 3. The right sidebar shows the "Code" tab with a Java solution. The solution uses a loop to calculate the number of ways to reach each step, updating two variables, step1 and step2, to represent the number of ways to reach the previous two steps.

**70. Climbing Stairs** Solved

Easy Topics Companies Hint

You are climbing a staircase. It takes  $n$  steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Example 1:**

**Input:**  $n = 2$   
**Output:** 2  
**Explanation:** There are two ways to climb to the top.  
1. 1 step + 1 step  
2. 2 steps

**Example 2:**

**Input:**  $n = 3$   
**Output:** 3  
**Explanation:** There are three ways to climb to the top.  
1. 1 step + 1 step + 1 step  
2. 1 step + 2 steps  
3. 2 steps + 1 step

```
1 class Solution {
2     public int climbStairs(int n) {
3         if (n == 1) return 1;
4         if (n == 2) return 2;
5         int step1 = 1;
6         int step2 = 2;
7         int inc = 0;
8
9         for (int i = 3; i <= n; i++) {
10             inc = step1 + step2;
11             step1 = step2;
12             step2 = inc;
13         }
14         return step2;
15     }
16 }
```

### Approach:

This code uses a technique similar to dynamic programming to tackle the "climbing stairs" problem. Each step on an  $n$ -step staircase may be accessed by either taking two steps from two steps below or one step from the step before it. We begin by examining the simplest scenarios: if there is a single step, there is only one method to ascend it; if there are two stairs, there are two ways to ascend them. The number of ways to get to each step starting at step 3 is then determined using a loop that adds the number of ways to go to the two steps before it. Since the answer to each step depends on the sum of the two steps before it, this is comparable to computing Fibonacci numbers.

### Challenge:

Effective memory management as the number of steps increases is a challenge in this topic. To avoid utilizing excessive space for an array of all outcomes, we merely save the latest two results and update them in each iteration. As a result, the code is efficient and memory utilization is kept low.

## Palindrome:

The screenshot shows the LeetCode interface for the problem '9. Palindrome Number'. The left sidebar shows the 'Test Result' tab with 'Accepted' status and 'Runtime: 0 ms'. The main content area shows the problem description: 'Given an integer x, return true if x is a palindrome, and false otherwise.' It includes two examples: Example 1 with input x = 121 and output true, and Example 2 with input x = -121 and output false. The right sidebar shows the 'Code' editor with a Java solution. The solution is a class 'Solution' with a method 'isPalindrome' that checks if the number is a palindrome by reversing its digits. The code is as follows:

```
1 class Solution {
2     public boolean isPalindrome(int x) {
3         if (x < 0 || (x % 10 == 0 && x != 0)) {
4             return false;
5         }
6         int reversed = 0;
7         int original = x;
8         while (x > reversed) {
9             reversed = reversed * 10 + x % 10;
10            x /= 10;
11        }
12        return x == reversed || x == reversed / 10;
13    }
14 }
```

### Approach:

By flipping the number's digits and comparing the two, we may determine if a given number is a palindrome. It is a palindrome if they are the same. It is not a palindrome if they are not.

**Reverse the Digits:** Take each last digit of the supplied number x, add it to a new reversed number, and then take it out of x.

**In contrast:** Check to see if the reversed number matches the original once it has been created. Return true if it does, and false otherwise.

### Challenge:

Managing situations when reversing the full number might result in overflow is a major difficulty, particularly for huge numbers. This problem can be avoided by reversing only half of the digits. This streamlines the operation and improves handling of edge circumstances by just reversing enough of the number to check for equality rather than the entire number.

## Final Array State After K Multiplication Operations:

### Approach:

For a specific number of iterations (j), the code multiplies the array's smallest element by a given factor (mul) to modify the array's numbers. Finding the array's lowest member, multiplying it by mul, and updating the array with the new value are the steps involved in each iteration. The array's final state is returned when this operation is carried out j times.

## Challenge:

One difficulty is that, for big arrays or high values of  $j$ , utilizing a nested loop to determine the minimal element in each iteration may be inefficient, leading to increased time complexity. Finding the minimal value in each loop iteration by repeatedly scanning the array can be slow, and improving it may call for the use of various data structures, such as a priority queue, to increase performance.

The screenshot shows a coding problem interface for "3264. Final Array State After K Multiplication Operations I". The problem is marked as "Solved". The input is an array `nums = [2, 1, 3, 5, 6]`, `k = 5`, and `multiplier = 2`. The description states: "You are given an integer array `nums`, an integer `k`, and an integer `multiplier`. You need to perform `k` operations on `nums`. In each operation: Find the minimum value `x` in `nums`. If there are multiple occurrences of the minimum value, select the one that appears first. Replace the selected minimum value `x` with `x * multiplier`. Return an integer array denoting the final state of `nums`." The code editor shows a Java solution:

```
1 class Solution {
2     public int[] getFinalState(int[] numbers, int j, int mul) {
3         for (int i = 0; i < j; i++) {
4             int min_Index = 0;
5             for (int k = 1; k < numbers.length; k++) {
6                 if (numbers[k] < numbers[min_Index]) {
7                     min_Index = k;
8                 }
9             }
10            numbers[min_Index] = numbers[min_Index]*mul;
11        }
12        return numbers;
13    }
14 }
15
```

## Linked List Cycle:

The screenshot shows a coding problem interface for "141. Linked List Cycle". The problem is marked as "Solved". The input is `head = [3, 2, 0, -4]` and `pos = 1`. The output is `true`. The description states: "Given `head`, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. Note that `pos` is not passed as a parameter. Return `true` if there is a cycle in the linked list. Otherwise, return `false`." The code editor shows a Java solution:

```
12 public class Solution {
13     public boolean hasCycle(ListNode head) {
14         if (head == null || head.next == null) {
15             return false;
16         }
17         ListNode slow = head;
18         ListNode fast = head.next;
19         while (fast != null && fast.next != null) {
20             if (slow == fast) {
21                 return true;
22             }
23             slow = slow.next;
24             fast = fast.next.next;
25         }
26         return false;
27     }
28 }
29
```

Approach: This method uses two pointers, slow and fast, to determine whether a linked list contains a loop. Fast takes two steps at a time, whereas slow takes one. Fast will ultimately catch up to slow if the list contains a loop, therefore they will eventually meet. The function returns true,



signifying the existence of a cycle, if they meet. Fast returns false if it reaches the end of the list, where there isn't a next node, indicating that there isn't a cycle. This method is effective and does not require more room.

Challenge: Managing edged circumstances, such as empty or single-element lists where a cycle isn't feasible, was the biggest obstacle. Careful loop conditions were also needed to make sure the fast pointer didn't move out of bounds. The secret to effectively identifying cycles without more space was the two-pointer approach.

## Merge Two Sorted Lists:

The screenshot shows a coding platform interface for the problem '21. Merge Two Sorted Lists'. The left panel displays the test results, showing 'Accepted' status with 0 ms execution time. The input lists are `list1 = [1, 2, 4]` and `list2 = [1, 3, 4]`, and the expected output is `[1, 1, 2, 3, 4, 4]`. The middle panel shows the problem description, which states: 'You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.' An example diagram shows a linked list with nodes 1, 2, and 4. The right panel shows the Java code solution, which uses a dummy node and a while loop to merge the two lists.

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode duplicate = new ListNode(0);
        ListNode present = duplicate;
        while (list1 != null && list2 != null) {
            if (list1.val < list2.val) {
                present.next = list1;
                list1 = list1.next;
            } else {
                present.next = list2;
                list2 = list2.next;
            }
            present = present.next;
        }
        if (list1 != null) {
            present.next = list1;
        } else if (list2 != null) {
            present.next = list2;
        }
        return duplicate.next;
    }
}
```

Approach: To make the process of merging two sorted linked lists easier, a fake node is used. We shift the pointer of the corresponding list after comparing the nodes in the two lists and adding the smaller one to the combined list. We add the leftover nodes from the other list once one list is depleted. To make sure the outcome is ordered, we lastly return the combined list, beginning with the node that comes after the dummy.

Challenge: Making sure that the final list appropriately contains any nodes that remain from either list once one of them is exhausted was the biggest issue encountered during the coding process. It was challenging to handle the edge circumstances when one list was longer than the other and make sure no nodes were overlooked, but the reasoning was made simpler by utilizing a fake node.

## Remove Duplicates from Sorted Array:

Approach: The method uses a two-pointer strategy to eliminate duplicates in a sorted array. To keep track of unique components, we establish a reference called `unique_position`. When a new unique element is discovered when iterating over the array, we move `unique_position` to the element's new location. Lastly, to show how many unique components there are in the array, we return `unique_position + 1`. This approach requires no additional space and operates in  $O(n)$  time.

Challenge: Using the array in place is a significant difficulty with this method, necessitating careful index control. Additionally, the algorithm must keep the pieces in the correct sequence and make sure that duplicates are successfully eliminated without erasing crucial data.

The screenshot shows a coding problem interface for "26. Remove Duplicates from Sorted Array". The left panel displays the test result as "Accepted" with a runtime of 0 ms. It shows the input array [1, 1, 2] and the expected output [1, 2]. The middle panel contains the problem description: "Given an integer array `nums` sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in `nums`." It also provides hints and a list of requirements. The right panel shows the code editor with a Java solution:

```
1 class Solution {
2     public int removeDuplicates(int[] numbers) {
3         if (numbers.length == 0) return 0;
4         int unique_position = 0;
5         for (int i = 1; i < numbers.length; i++) {
6             if (numbers[i] != numbers[unique_position]) {
7                 unique_position++;
8                 numbers[unique_position] = numbers[i];
9             }
10        }
11        return unique_position + 1;
12    }
13 }
14
```

## Remove Element:

The screenshot shows a coding problem interface for "27. Remove Element". The left panel displays the test result as "Accepted" with a runtime of 0 ms. It shows the input array [3, 2, 2, 3] and the value to remove 3. The expected output is [2, 2]. The middle panel contains the problem description: "Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` in-place. The order of the elements may be changed. Then return the number of elements in `nums` which are not equal to `val`." It also provides hints and a list of requirements. The right panel shows the code editor with a Java solution:

```
1 class Solution {
2     public int removeElement(int[] numbers, int val) {
3         int index = 0;
4         for (int i = 0; i < numbers.length; i++) {
5             if (numbers[i] != val) {
6                 numbers[index] = numbers[i];
7                 index++;
8             }
9         }
10        return index;
11    }
12 }
13
```

Approach: The approach uses a two-pointer technique: one pointer iterates through the array, while the other keeps track of the position to place elements that aren't equal to the target value. As we encounter elements not equal to the target, we move them to the correct position. The challenge is modifying the array in-place without extra space, ensuring we don't overwrite valid elements. This solution is efficient with  $O(n)$  time complexity and  $O(1)$  space complexity.

Challenge: Efficiently eliminating members from the array without utilizing additional space while preserving the order of the remaining components is the primary problem here. It necessitates updating the array's new length without erasing crucial data and managing pointers carefully to guarantee that valid components are moved appropriately.

## Longest Common Prefix:

The screenshot shows a coding platform interface for the 'Longest Common Prefix' problem. The interface is divided into three main sections: Testcase, Description, and Code.

**Testcase Section:** Shows the problem status as 'Accepted' with a runtime of 0 ms. It displays the input and output for a test case. The input is an array of strings: ["flower", "flow", "flight"]. The output is the longest common prefix: "fl".

**Description Section:** Contains the problem statement: 'Write a function to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string "".' It also provides two examples: Example 1 with input ["flower", "flow", "flight"] and output "fl", and Example 2 with input ["dog", "racecar", "car"] and output "".

**Code Section:** Shows the Java code for the solution. The code defines a class 'Solution' with a method 'longestCommonPrefix' that takes an array of strings and returns the longest common prefix.

```
1 class Solution {
2     public String longestCommonPrefix(String[] str) {
3         if (str == null || str.length == 0) return "";
4         String prefix = str[0];
5         for (int i = 1; i < str.length; i++) {
6             while (str[i].indexOf(prefix) != 0) {
7                 prefix = prefix.substring(0, prefix.length() - 1);
8                 if (prefix.isEmpty()) return "";
9             }
10        }
11        return prefix;
12    }
13 }
14
```

### Approach:

This code begins with the first string as a possible prefix and then iteratively examines each of the other strings to get the longest common prefix among a collection of strings. If the current prefix does not match the beginning of each string, it shortens the prefix one character at a time. The procedure keeps going until it either finds a common prefix for all strings or finds none, in which case it produces an empty string.

### Challenge:

Managing situations when strings have different beginning characters or when one string is significantly shorter than the others is the most difficulty. To avoid wasting time on pointless comparisons, the code must effectively shorten the prefix length in certain situations and terminate early if no match can be found.

## Search Insert position:

The screenshot shows the LeetCode interface for the problem '35. Search Insert Position'. The 'Testcase' tab is active, showing 'Case 1' with 'nums = [1,3,5,6]' and 'target = 5'. The 'Description' tab shows the problem statement: 'Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You must write an algorithm with  $O(\log n)$  runtime complexity.' Example 1 shows 'Input: nums = [1,3,5,6], target = 5' and 'Output: 2'. The 'Code' tab shows a Java solution using binary search.

```
1 class Solution {
2     public int searchInsert(int[] nums, int target) {
3         int left = 0, right = nums.length - 1;
4
5         while (left <= right) {
6             int mid = left + (right - left) / 2;
7
8             if (nums[mid] == target) {
9                 return mid;
10            } else if (nums[mid] < target) {
11                left = mid + 1;
12            } else {
13                right = mid - 1;
14            }
15        }
16        return left;
17    }
18 }
19
```

### Approach:

The code uses a technique known as binary search to determine where to insert a target number in a sorted list. Consider opening a dictionary in the middle, determining if you are too early or late, and then narrowing your search to only half the remaining pages to look up a word. This method is comparable. We begin by examining the center of the list in this instance. We're done if the desired number is reached. We continue to search in the left half if the target is smaller, and the right half if it is larger. This procedure is repeated until we either locate the target or pinpoint the precise location on the list where it would fit, at which point we return it.

### Challenge:

Managing situations in which the desired number is absent from the list is a problem. For instance, the function must appropriately return the starting position if the goal is less than all numbers, or the end position if the target is greater than all numbers. Making sure we don't compute the midway point incorrectly is another challenging aspect that might result in mistakes or an endless cycle in longer lists.

## Remove Duplicate from Sorted List:

### Approach:

We begin at the head of a sorted linked list and proceed through each node to eliminate duplicates. As we go, we verify that the value of the current node and the subsequent node are same. If they do, we modify the next pointer to "bypass" the duplicate and move on to the next node. We just go to the next node if the values vary. In this manner, as we advance, we simply retain the distinctive aspects

**83. Remove Duplicates from Sorted List** Solved

Easy Topics Companies

Given the `head` of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list **sorted** as well.

**Example 1:**

```

graph LR
    1((1)) --> 1_2((1))
    1_2 --> 2((2))
  
```

Input: `head = [1,1,2]`  
Output: `[1,2]`  
Expected: `[1,2]`

Runtime: 0 ms

8.9K 92 63 Online

```

class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode present = head;
        while (present != null && present.next != null) {
            if (present.val == present.next.val) {
                present.next = present.next.next;
            } else {
                present = present.next;
            }
        }
        return head;
    }
}
  
```

## Challenge:

The key issue with this problem is smoothly managing edge circumstances, such as when the list is empty or has all identical items. Another is to ensure that when duplicates are discovered, we appropriately modify the next pointers to avoid mistakenly losing nodes or triggering problems. This necessitates careful pointer handling to prevent skipping any unique components while keeping the algorithm efficient at  $O(n)$  time complexity without consuming extra space.

## Binary Tree Inorder Traversal:

**Binary Tree Inorder Traversal**

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4

Input: `root = [1,null,2,3]`  
Output: `[1,3,2]`  
Expected: `[1,3,2]`

Runtime: 0 ms | Beats 100.00%  
Memory: 41.68 MB | Beats 30.61%

```

class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        inorderHelper(root, result);
        return result;
    }
    private void inorderHelper(TreeNode node, List<Integer> result) {
        if (node == null) {
            return;
        }
        inorderHelper(node.left, result);
        result.add(node.val);
        inorderHelper(node.right, result);
    }
}
  
```

### Approach:

The approach uses a recursive helper function to traverse the tree in-order (left, root, right). Starting with the root, it recursively explores the left subtree, adds the current node's value to the result list, and then explores the right subtree. The challenge here is ensuring the recursion handles all nodes and edge cases, such as an empty tree (null root), without causing stack overflow or missing nodes. The recursion also maintains the correct order by following the left-root-right sequence.

### Challenger:

The biggest issue when coding was successfully controlling the recursion, particularly the base case in which the node is null. Ensuring that the traverse followed the right in-order sequence, without missing any nodes or generating a stack overflow, needed careful consideration.

## Algorithm Problems:

### Max Depth of Binary Tree:

The screenshot shows the LeetCode interface for the problem "104. Maximum Depth of Binary Tree". The left sidebar shows the "Test Result" tab with "Accepted" status and "Runtime: 0 ms". The main content area displays the problem description: "Given the root of a binary tree, return its maximum depth." and an example of a tree with root 3. The right sidebar shows the "Code" editor with a Java solution. The solution defines a recursive function `maxDepth` that returns 0 for a null root and the maximum of the left and right subtree depths plus 1 for the current node.

```
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int l_Depth = maxDepth(root.left);
        int r_Depth = maxDepth(root.right);
        return 1 + Math.max(l_Depth, r_Depth);
    }
}
```

### Approach:

The function uses recursion to get the maximum depth of a binary tree. It starts at the root and verifies the depth of the left and right subtrees. It increases the maximum depth between the left and right subtrees by one (counting the current node). When it hits a null node, it returns zero, indicating the end of the branch.

### Challenge:

The main problem is managing deep recursion effectively. Deep recursion can cause a stack overflow fault when dealing with particularly long or imbalanced trees. Furthermore, addressing edge circumstances such as an empty tree (where root is null) is critical to ensuring the code's reliability.

## Minimum Depth of Binary Tree:

The screenshot shows the LeetCode interface for problem 111. On the left, the 'Testcase' tab is active, showing 'Accepted' status with a runtime of 0 ms. The input is an array [3, 9, 20, null, null, 15, 7] and the output is 2. The 'Description' tab in the center explains the problem: finding the minimum depth of a binary tree, defined as the number of nodes along the shortest path from the root to a leaf. It includes a note that a leaf is a node with no children and an example diagram of a tree with root 3 and two children. On the right, the 'Code' tab shows a Java solution using a recursive approach to find the minimum depth by comparing the depths of the left and right subtrees.

### Approach:

To find the minimum depth of a binary tree, we start from the root and look for the shortest path to a leaf node (a node with no children). If the tree is empty, the depth is 0. If a node has only one child, we go down the non-null subtree. When both children exist, we calculate the minimum depth of each subtree and add 1 for the current node, thus capturing the shortest path to a leaf. This recursive approach ensures we find the minimum depth efficiently.

### Challenge:

A key challenge in finding the minimum depth of a binary tree is correctly handling nodes with only one child. Unlike the maximum depth problem (where you can freely explore both sides of each node), the minimum depth problem requires attention to cases where only one child exists. If only one child is present, you must avoid counting the non-existent path (which would be 0), and instead, continue down the existing child path. This ensures you accurately calculate the shortest path to a true leaf, avoiding incorrect results.

## Convert Sorted Array to Binary Search Tree:

### Approach:

This method uses a divide-and-conquer strategy to create a balanced Binary Search Tree (BST) from a sorted array. The main concept is to maintain the tree balanced by always choosing the middle element of each subarray as the root of each subtree. Beginning with the full array, the middle element forms the root of the BST. The function then recursively chooses the center elements of the left and right halves as the roots of the left and right subtrees, respectively. This recursive breakdown continues until there are no elements left in a subarray, at which time we return null, finishing the branch.





lists until the pointers coincide, signifying the intersection node. If there is no intersection, both points will approach null simultaneously, and we will return null.

Challenge:

The key issues are keeping list alignment when lists vary in length and guaranteeing efficient traversal without using extra memory. Edge cases include lists that do not intersect or lists that are empty. This technique is efficient, having  $O(n)$  time complexity and  $O(1)$  space complexity, making it ideal for huge lists.

## Path Sum:

The screenshot shows the LeetCode interface for problem 112, "Path Sum". The left sidebar displays the "Testcase" tab with "Test Result" showing "Accepted" in green. Below this, the input is defined as a binary tree with root value 5, left child 4, and right child 8. The target sum is 22. The output is "true". The main content area shows the problem description: "Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum. A leaf is a node with no children." An example 1 shows a tree with root 5, left child 4, and right child 8. The right sidebar shows the "Code" editor with a Java solution. The solution is a recursive function that checks if the current node is a leaf and if the remaining sum equals the node's value. If not, it recursively checks the left and right subtrees.

```
12  *      this.right = right;
13  *      }
14  *      }
15  */
16  public class Solution {
17      public boolean hasPathSum(TreeNode root, int targetSum) {
18          if (root == null) {
19              return false;
20          }
21
22          if (root.left == null && root.right == null) {
23              return targetSum == root.val;
24          }
25
26          int remainingSum = targetSum - root.val;
27          return hasPathSum(root.left, remainingSum) || hasPathSum(root.right, remainingSum);
28      }
29  }
30
```

Approach:

To determine if a binary tree has a root-to-leaf path that sums up to a given target, we start at the root and subtract the current node's value from the target. If we reach a leaf node and the remaining sum equals the leaf's value, we know we've found a valid path.

Otherwise, we recursively check the left and right subtrees with the updated target. If either subtree has a valid path, we return true. This ensures we explore all possible paths efficiently.

Challenge:

The primary challenge was ensuring logic correctly identifies paths that end specifically at leaf nodes, avoiding false positives from incomplete paths. Managing edge cases like an empty tree or trees with only one node added complexity. Additionally, designing the recursion to handle varying tree structures while maintaining clarity required careful thought.

## Pascal Triangle:

### Approach

To build Pascal's Triangle with a specified number of rows (numRows), we begin with the first row, which has only one element, 1. Each succeeding row is constructed by adding nearby components from the preceding row and inserting 1 at both ends of the row. This method ensures that the triangle's structure is retained. We utilize a loop to iteratively generate each row depending on its predecessor before adding it to the final list.

### Challenge:

The main challenge was ensuring correct handling of the indexing while calculating sums for the new rows. Managing edge cases like numRows = 0 or 1 required careful initialization. Efficiently updating the triangle while maintaining its integrity at each step also demanded attention to detail.

The screenshot shows the LeetCode interface for problem 118, "Pascal's Triangle". The problem description states: "Given an integer numRows, return the first numRows of Pascal's triangle. In Pascal's triangle, each number is the sum of the two numbers directly above it as shown:". A diagram of the first 5 rows of Pascal's triangle is displayed, with numbers 1, 3, 3, 1 in the third row highlighted in orange. The input field shows "numRows = 5". The output field shows the expected result: "[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]". The "Accepted" status is shown with a runtime of 0 ms. The code editor on the right shows a Java solution:

```
1 class Solution {
2     public List<List<Integer>> generate(int numRows) {
3         List<List<Integer>> triangle = new ArrayList<>();
4
5         for (int i = 0; i < numRows; i++) {
6             List<Integer> row = new ArrayList<>();
7             row.add(1);
8
9             for (int j = 1; j < i; j++) {
10                row.add(triangle.get(i - 1).get(j - 1) + triangle.get(i - 1).get(j));
11            }
12
13            if (i > 0) {
14                row.add(1);
15            }
16
17            triangle.add(row);
18        }
19
20        return triangle;
21    }
22 }
23
```

## Remove Linked List Elements:

### Approach:

The purpose is to delete all nodes in a linked list that have a certain value (val). The technique calls for the use of a dummy node to address edge circumstances, such as when the target value is included within the head itself. We go over the list and inspect each node. If the following node contains the intended value, we can simply skip it by modifying the pointers. This procedure continues until the entire list has been scanned, at which time we return the list beginning with dummy.next, pointing to the new head.

The screenshot shows a coding platform interface for problem 203, "Remove Linked List Elements". The interface is divided into three main sections: Testcase, Description, and Code.

- Testcase:** Shows the problem status as "Accepted" with a runtime of 0 ms. It includes input fields for "head" (value: [1,2,6,3,4,5,6]) and "val" (value: 6). The output field shows [1,2,3,4,5], and the expected output is also [1,2,3,4,5].
- Description:** Contains the problem statement: "Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return the new head." It includes an example diagram showing a linked list [1, 2, 6, 3, 4, 5, 6] being transformed into [1, 2, 3, 4, 5] by removing nodes with value 6.
- Code:** Displays a Java solution for the problem. The code defines a ListNode class and a removeElements method that iterates through the linked list, removing nodes with the specified value.

Challenge:

One major problem was ensuring that the algorithm handled edge circumstances correctly, such as when the item to be eliminated is at the top of the list or when the list is empty. Furthermore, carefully managing pointer changes to prevent losing sections of the list when deleting members was critical. The dummy node simplified these scenarios, but pointer handling need careful attention to avoid mistakes and guarantee the list remained intact after changes.

## Reverse a Singly Linked List:

The screenshot shows a coding platform interface for problem 206, "Reverse Linked List". The interface is divided into three main sections: Testcase, Description, and Code.

- Testcase:** Shows the problem status as "Accepted" with a runtime of 0 ms. It includes input fields for "head" (value: [1,2,3,4,5]) and "Output" (value: [5,4,3,2,1]). The expected output is also [5,4,3,2,1].
- Description:** Contains the problem statement: "Given the head of a singly linked list, reverse the list, and return the reversed list." It includes an example diagram showing a linked list [1, 2, 3, 4, 5] being transformed into [5, 4, 3, 2, 1] by reversing the order of the nodes.
- Code:** Displays a Java solution for the problem. The code defines a ListNode class and a reverseList method that iterates through the linked list, reversing the order of the nodes.

Approach:

The purpose is to reverse a single linked list. We go over the list, reversing the orientation of the pointers as we go. We keep three pointers during the traversal: previous, current, and next. Initially, predecessor is null, and current moves to the top of the list. In each repetition, we store the next node, reverse the current node's pointer to refer to predecessor, and advance prev and current by one step. Finally, after the loop is completed, prev will refer to the new head of the inverted list.

Challenge:

Carefully managing pointer manipulation is the primary difficulty while reversing a linked list. It's important to preserve the remaining nodes because we're traversing the list in a different way. Data loss can be avoided by keeping track of the next node before changing the pointer of the current node. Making sure the list's new head is appropriately returned following the reversal presents another difficulty. It takes care to handle these pointer references correctly, particularly in edge circumstances like an empty list or a list with only one entry.

### Shortest Subarray to be removed to make Array Sorted:

The screenshot displays a coding platform interface with three main panels. The left panel, titled 'Testcase | Test Result', shows the problem status as 'Accepted' with a runtime of 0 ms. It lists three test cases, with 'Case 1' selected. The input for Case 1 is an array `arr = [1, 2, 3, 10, 4, 2, 3, 5]`, and the output is `3`. The expected output is also `3`. The middle panel, titled 'Description', contains the problem statement for '1574. Shortest Subarray to be Removed to Make Array Sorted'. It specifies that the array is non-decreasing and asks for the length of the shortest subarray to remove. An example is provided: Input `arr = [1, 2, 3, 10, 4, 2, 3, 5]`, Output `3`, and Explanation: The shortest subarray we can remove is `[10, 4, 2]`. The right panel, titled 'Code', shows a Java solution. The code uses two pointers, `left` and `right`, to find the longest sorted subarray. It then calculates the length of the shortest subarray to remove as `n - (right - left + 1)`. The code is as follows:

```
4 int left = 0;
5 int right = n - 1;
6
7 while (left < n - 1 && arr[left] <= arr[left + 1]) {
8     left++;
9 }
10
11 if (left == n - 1) {
12     return 0;
13 }
14
15 while (right > 0 && arr[right - 1] <= arr[right]) {
16     right--;
17 }
18
19 int result = Math.min(n - left - 1, right);
20
21 int i = 0, j = right;
22 while (i <= left && j < n) {
23     if (arr[i] <= arr[j]) {
24         result = Math.min(result, j - i - 1);
25         i++;
26     } else {
27         j++;
28     }
29 }
```

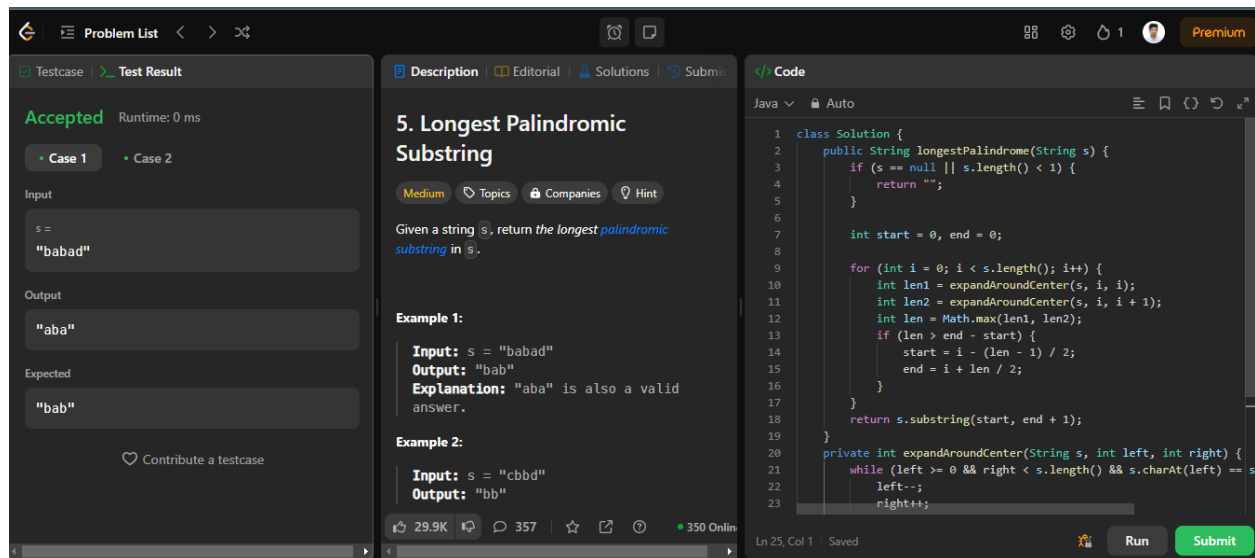
Approach:

To solve this problem, consider the array as having three parts: a sorted beginning (prefix), a sorted ending (suffix), and an unsorted middle. Your aim is to eliminate the smallest area of the center, allowing the beginning and finish to link into a single smooth, sorted array. First, determine how long the sorted sections are at the beginning and end. Then, check to see if the two sorted portions can merge straight or if one of the prefixes or suffixes has to be trimmed to fit together precisely.

Challenge:

The tough aspect is figuring out how to handle the middle segment efficiently. Do you delete it totally, or can you discover a little section that, once removed, allows the remainder of the array to remain sorted? Balancing this choice without manually testing every conceivable combination is the true issue, especially when the array is huge or contains numerous unsorted parts.

## Longest Palindromic Substring:



The screenshot displays a coding platform interface for the "5. Longest Palindromic Substring" problem. The interface is divided into three main sections:

- Testcase Panel (Left):** Shows the problem status as "Accepted" with a runtime of 0 ms. It includes two cases: "Case 1" and "Case 2". For Case 1, the input is `s = "babad"` and the output is `"aba"`. The expected output is `"bab"`. A button "Contribute a testcase" is visible at the bottom.
- Description Panel (Center):** Contains the problem title "5. Longest Palindromic Substring" with a "Medium" difficulty level. It includes tags for "Topics", "Companies", and "Hint". The description states: "Given a string `s`, return the longest palindromic substring in `s`." It provides two examples:
  - Example 1:** Input: `s = "babad"`, Output: `"bab"`, Explanation: `"aba"` is also a valid answer.
  - Example 2:** Input: `s = "cbbd"`, Output: `"bb"`.
- Code Panel (Right):** Shows a Java solution. The code defines a `Solution` class with a `longestPalindrome` method. It uses a helper method `expandAroundCenter` to find the longest palindrome centered at each index. The code is as follows:

```
1 class Solution {
2     public String longestPalindrome(String s) {
3         if (s == null || s.length() < 1) {
4             return "";
5         }
6
7         int start = 0, end = 0;
8
9         for (int i = 0; i < s.length(); i++) {
10             int len1 = expandAroundCenter(s, i, i);
11             int len2 = expandAroundCenter(s, i, i + 1);
12             int len = Math.max(len1, len2);
13             if (len > end - start) {
14                 start = i - (len - 1) / 2;
15                 end = i + len / 2;
16             }
17         }
18         return s.substring(start, end + 1);
19     }
20     private int expandAroundCenter(String s, int left, int right) {
21         while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
22             left--;
23             right++;
24         }
25         return right - left - 1;
26     }
27 }
```

### Approach:

Consider the string to be a sequence of characters, and the goal is to discover the largest chunk that reads the same forward and backward (a palindrome). To do this, consider each letter (or set of letters) as the "center" of a possible palindrome. Then, extend outward in both directions to determine how far the symmetry maintains. Keep note of the longest palindrome you discover as you progress down the string. By conclusion, you will have found the largest mirrored area of the text.

### Challenge:

The difficult element is effectively examining all possible centers while avoiding superfluous effort. Palindromes can be odd or even in length, therefore you must consider both single-letter and double-letter centers. Additionally, managing extreme circumstances such as very short strings or strings with no major palindromes might be difficult. Balancing clarity in implementation with performance is critical.

## Reverse Nodes in K-Groups:

### Approach:

Imagine a line of people holding hands, where you need to reverse groups of exactly  $k$  people at a time. Start at the beginning of the line and check if there are enough people in the current group to reverse (at least  $k$  individuals). If there aren't enough, leave them as they are and move on.

For each group that can be reversed, imagine temporarily breaking their hands, turning the group around, and reconnecting them to the rest of the line. Use placeholders (like markers at the start and end of the group) to keep track of where to reconnect. After reversing one group, move forward to the next group of  $k$  people and repeat the process. By the end, you'll have a transformed line where groups of  $k$  people are reversed while the overall structure stays intact.

The screenshot shows the LeetCode interface for the problem "25. Reverse Nodes in k-Group". The left sidebar shows the "Testcase" tab with "Case 1" selected, displaying input `head = [1,2,3,4,5]` and `k = 2`, and the expected output `[2,1,4,3,5]`. The main content area shows the problem description: "Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list. k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is. You may not alter the values in the list's nodes, only nodes themselves may be changed." Below the description is "Example 1:" with a diagram of a linked list: 1 → 2 → 3 → 4 → 5. Nodes 1 and 2 are blue, 3 and 4 are red, and 5 is blue. The right sidebar shows the "Code" tab with a Java solution:

```
10 /
11 class Solution {
12     public ListNode reverseKGroup(ListNode head, int k) {
13         if (head == null || k == 1) {
14             return head;
15         }
16         ListNode dummy = new ListNode(0);
17         dummy.next = head;
18         ListNode prevGroupEnd = dummy;
19         ListNode curr = head;
20         while (true) {
21             ListNode kthNode = getKthNode(curr, k);
22             if (kthNode == null) {
23                 break;
24             }
25             ListNode nextGroupStart = kthNode.next;
26             ListNode[] reversed = reverse(curr, kthNode);
27             prevGroupEnd.next = reversed[0];
28             reversed[1].next = nextGroupStart;
29             prevGroupEnd = reversed[1];
30             curr = nextGroupStart;
31         }
32         return dummy.next;
33     }
34 }
```

### Challenge:

The major issue here is to manage the connections while reversing each group without losing sight of the remainder of the line. When you reverse a group, you may become confused about where it begins and ends, as well as how to reconnect it to the remainder of the list. You must also ensure that you only reverse whole groups of  $k$ —if there are less than  $k$  persons remaining at the end, leave them alone. The challenge is to balance these tasks while keeping the process efficient and error-free.