

DSA FOR DATA ENGINEERS

BY - SHUBHAM WADEKAR

Copyright © Shubham Wadekar. All rights reserved.

This material is for personal use only. No part may be copied, shared, resold, or published without written permission. Unauthorized distribution is strictly prohibited and may result in action.

For permissions or licensing, contact: [shubham.p.wadekar@gmail.com]

Disclaimer: This content is for educational purposes. Accuracy is attempted but not guaranteed. No job outcome is promised.

ARRAYS

Problem 1 - Two Sum

Problem Statement -

Given an array of integers `nums` and an integer `target`, return the indices of the two numbers in the array such that they add up to the target. You may assume that each input has exactly one solution, and you may not use the same element twice. You can return the answer in any order.

Logic Explanation

The goal is to identify two distinct elements in the array whose sum equals the target and return their indices. A brute-force method checks every possible pair using nested loops, leading to $O(n^2)$ time complexity—which is inefficient for large inputs.

A significantly more efficient approach leverages a hash table (Python dictionary). As you iterate through the list, for each element `num` at index `i`, compute its complement as `target - num`. Immediately check if this complement exists in the hash map:

- If it does, it means we have already seen a number that pairs with the current `num` to make the target, so we can return the indices.
- If not, store the current `num` with its index in the map and continue.

This one-pass method achieves $O(n)$ time complexity with $O(n)$ additional space, since each element is processed exactly once and dictionary lookups are on average constant time.

This is the standard interview and LeetCode optimal solution that scales well even for large arrays.

Solution:

```
class Solution:
    def twoSum(self, nums: list[int], target: int) -> list[int]:
        num_index: dict[int, int] = {}
        for i, num in enumerate(nums):
            complement = target - num
            if complement in num_index:
                return [num_index[complement], i]
            num_index[num] = i
        raise ValueError("No solution found")
```

Practice here

[Link to the problem](#)

Problem 2 - Subarray Sum Equals K

Problem Statement

Given an array of integers `nums` and an integer `k`, return the total number of contiguous subarrays whose sum equals `k`.

Logic Explanation

We are asked to find the total number of continuous subarrays that sum up to a given value `k`. A naive way to solve this is by checking every possible subarray using two nested loops. For each starting index, we keep calculating the sum of elements until the end of the array and check whether that sum is equal to `k`. Although this works correctly, it has a time complexity of $O(n^2)$, which becomes very inefficient when the array size grows large.

To solve this in a more optimized way, we can use the concept of prefix sum along with a hashmap. The idea is to traverse the array while maintaining a cumulative sum of elements seen so far, which we refer to as `currSum`. At each point, we want to know how many times we have seen a cumulative sum that would allow the current subarray to sum up to `k`.

We do this by checking how many times `(currSum - k)` has occurred before. If it has occurred `x` times, it means there are `x` subarrays ending at the current index whose sum is exactly `k`. We keep adding this count to our answer.

To track how many times each cumulative sum has occurred, we use a dictionary (hashmap). We also initialize the hashmap with `{0: 1}` to handle cases where the subarray starting from index 0 itself adds up to `k`.

This approach only requires a single pass through the array, making the time complexity $O(n)$, and the hashmap stores at most `n` entries, so the space complexity is also $O(n)$. It is the most efficient way to solve this problem for large datasets.

Solution:

```
class Solution:
    def subarraySum(self, nums: list[int], k: int) -> int:
        count = 0
        currSum = 0
        prefixFreq = {0: 1} # Start with sum 0 having one count

        for num in nums:
            currSum += num
            # Check how many times currSum - k has appeared so far
            if (currSum - k) in prefixFreq:
                count += prefixFreq[currSum - k]
            # Update the frequency of currSum in the map
            prefixFreq[currSum] = prefixFreq.get(currSum, 0) + 1

        return count
```

Practice here

[Link to the problem](#)

Problem 3 - Find the Duplicate Number

Problem Statement

Given an array of integers `nums` containing $n + 1$ integers where each integer is in the range $[1, n]$ inclusive, and there is exactly one repeated number in `nums`. Return that repeated number without modifying the array and using only constant extra space.

Logic Explanation

We are given an array of $n + 1$ numbers where each number is between 1 and n . Since there are more numbers than the range allows, at least one number must repeat. The task is to find the duplicate without changing the array and using only constant space.

To solve this efficiently, we can use a smart approach based on cycle detection. The idea is to treat the array like a linked list where the value at each index represents the next pointer. For example, from index i , the next position is `nums[i]`. Because there's at least one duplicate, the structure will form a cycle. Our goal is to find the start of this cycle, which corresponds to the repeated number.

To detect the cycle, we use Floyd's Tortoise and Hare algorithm. We initialize two pointers, `slow` and `fast`, starting from the first element. We move `slow` one step at a time and `fast` two steps at a time. If there's a cycle, they will eventually meet at some point inside the cycle.

Once they meet, we reset one of the pointers to the beginning of the array. Then, we move both pointers one step at a time. The point at which they meet again is the start of the cycle and hence the duplicate number.

This method works in linear time and constant space because we only use two pointers and don't need any extra data structures. It also satisfies the constraint of not modifying the original array.

Solution:

```
class Solution:
    def findDuplicate(self, nums: list[int]) -> int:
        # Phase 1: Detect intersection point in the cycle
        slow = nums[0]
        fast = nums[0]
        while True:
            slow = nums[slow]
            fast = nums[nums[fast]]
            if slow == fast:
                break

        # Phase 2: Find the start of the cycle (duplicate number)
        slow = nums[0]
        while slow != fast:
            slow = nums[slow]
            fast = nums[fast]

        return slow
```

Practice here

[Link to the problem](#)

Problem 4 - Best Time to Buy and Sell Stock

Problem Statement

Given an array prices where prices[i] represents the price of a stock on day i, design an algorithm to maximize profit by choosing exactly one day to buy and a different later day to sell. If no profit is possible, return 0.

Logic Explanation

The main goal is to find the maximum profit you can earn by buying the stock on one day and selling it on a later day. One key point here is that the selling day must come after the buying day. So, for each day, we must look back to find the lowest price we could have bought the stock before that day.

A brute-force solution would be to check every pair of days (i, j) such that $i < j$ and compute $\text{prices}[j] - \text{prices}[i]$. This would involve nested loops and take $O(n^2)$ time, which is inefficient for large arrays.

To optimize this, we can use a greedy single-pass approach. The idea is to keep track of two variables while traversing the array:

- min_price: the lowest price seen so far (i.e., the best day to buy so far).
- max_profit: the highest profit we can achieve if we sell on the current day after buying at min_price.

We start by initializing min_price to a very high value (like infinity) so that any price in the array will be lower on the first comparison. We also initialize max_profit to 0, which is the default if no profit is possible.

As we loop through the prices:

- If the current price is lower than min_price, we update min_price.
- Otherwise, we calculate the profit by subtracting min_price from the current price. If this profit is greater than max_profit, we update max_profit.

This ensures that at every step, we are always aware of the lowest price to buy from the past and the best profit possible up to the current day. We never need to go back or re-check previous values, so the time complexity is $O(n)$, and only constant space is used.

This approach is both time and space efficient and directly solves the problem with a clear logical flow.

Solution:

```
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        max_profit = 0
        min_price = float('inf')

        for price in prices:
            if price < min_price:
                min_price = price
            else:
                profit = price - min_price
                if profit > max_profit:
                    max_profit = profit

        return max_profit
```

Practice here

[Link to the problem](#)

Problem 5 - Best Time to Buy and Sell Stock II

Problem Statement

You are given an integer array `prices` where `prices[i]` is the price of a stock on day `i`. On each day, you may decide to buy and/or sell the stock. You can hold at most one share at a time, but you can buy and sell on the same day. Return the maximum profit you can achieve.

Logic Explanation

In this version of the stock problem, we are allowed to make multiple buy and sell transactions, unlike the previous problem where only one transaction (one buy and one sell) was allowed. The goal here is to maximize the total profit over all transactions.

The key observation is: we should capture every increase in price. That means whenever the price goes up from one day to the next, we can assume we bought the stock the day before and sold it on the current day to gain the difference. Since we're allowed to buy and sell as often as we like, even buying and selling on consecutive days is allowed.


So, instead of trying to find local minima (buy points) and local maxima (sell points) manually, we can just look for every pair of consecutive days where the price increases, and add the difference to our total profit. This is because doing multiple small profitable trades will add up to the same result as waiting for a big peak.

For example, if prices go from $2 \rightarrow 4 \rightarrow 6$, the profit from buying at 2 and selling at 6 is $(6-2)=4$. But if we do it in parts, buying at 2 and selling at 4 gives 2, then buying at 4 and selling at 6 gives another 2. Total is still 4. So we don't need to find exact valleys and peaks; we just accumulate all upward trends.

This approach only requires a single pass through the list, checking the difference between consecutive prices. If the next day's price is higher, we add the profit to our result. If it's lower or equal, we skip it.

This greedy strategy results in $O(n)$ time complexity with $O(1)$ space, which is optimal.

Solution:

 Code

Python3   Auto

```
1 class Solution:
2     def maxProfit(self, prices: list[int]) -> int:
3         profit = 0
4         for i in range(1, len(prices)):
5             if prices[i] > prices[i - 1]:
6                 profit += prices[i] - prices[i - 1]
7         return profit
8
```

Practice here

[Link to the problem](#)

Problem 6 - Max Consecutive Ones

Problem Statement

Given a binary array `nums`, return the maximum number of consecutive 1s in the array.

Logic Explanation

This is a simple array traversal problem where the goal is to find the longest sequence of consecutive 1s in the given binary array. To solve this efficiently, we can scan through the array just once and use a counter to track how many continuous 1s we've seen so far.

We initialize two variables:

- `max_count` to store the maximum number of consecutive 1s found so far.
- `current_count` to count the number of 1s in the current ongoing sequence.

As we iterate through each element:

- If the current element is 1, we increment `current_count` by one.
- If the current element is 0, it means the sequence is broken. So, we compare `current_count` with `max_count`, update `max_count` if needed, and reset `current_count` to zero.

After finishing the loop, we do one final comparison between `current_count` and `max_count`, just in case the array ended with a sequence of 1s.

This solution runs in $O(n)$ time where n is the length of the array, and it uses $O(1)$ space.

Solution:

```
</> Code
Python3  Auto
1 class Solution:
2     def findMaxConsecutiveOnes(self, nums: list[int]) -> int:
3         max_count = 0
4         current_count = 0
5
6         for num in nums:
7             if num == 1:
8                 current_count += 1
9                 max_count = max(max_count, current_count)
10            else:
11                current_count = 0
12
13        return max_count
14
```

Practice here

[Link to the problem](#)

Problem 7 - Max Consecutive Ones III

Problem Statement

Given a binary array `nums` and an integer `k`, return the maximum number of consecutive 1s in the array if you can flip at most `k` zeros to ones.

Logic Explanation

This is a classic sliding window problem where we need to find the longest subarray that contains only 1s after flipping at most `k` zeros to 1s.

We approach this by using a window defined by two pointers: `left` and `right`. We start both pointers at the beginning of the array. The idea is to expand the `right` pointer step by step to include more elements in the window, and for each new element:

- If it is 1, we can include it without any issue.
- If it is 0, we treat it as a candidate for flipping. So we decrement `k` since we're pretending to flip this zero.

If at any point `k` becomes negative, it means we have used more than the allowed number of flips, so we need to shrink the window from the left. While shrinking, if the element at `left` was a zero, we increment `k` back since we are no longer flipping that zero. We move the `left` pointer forward until the number of zeros in the window is again at most `k`.

During this process, the size of the window (`right - left + 1`) gives us the count of consecutive 1s with at most `k` flips. We keep track of the maximum size of such a window as we slide through the array.

This method ensures each element is processed at most twice (once by `right`, once by `left`), so the time complexity is $O(n)$ and the space complexity is $O(1)$.

Solution:

```
Python3  Auto
1  class Solution:
2      def longestOnes(self, nums: list[int], k: int) -> int:
3          left = 0
4
5          for right in range(len(nums)):
6              if nums[right] == 0:
7                  k -= 1
8
9              if k < 0:
10                 if nums[left] == 0:
11                     k += 1
12                 left += 1
13
14             return right - left + 1
15
```

Practice here

[Link to the problem](#)

Problem 8 - Maximum Subarray

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Logic Explanation

This is a classic problem that can be solved efficiently using Kadane's Algorithm. The idea is to keep track of the maximum subarray sum ending at each position as we iterate through the array. At every index, we decide whether to:

1. Extend the previous subarray by including the current number, or
2. Start a new subarray from the current number.

To do this, we maintain two variables:

- `current_sum`: the maximum sum of a subarray ending at the current index.
- `max_sum`: the maximum value of `current_sum` seen so far (our final result).

We initialize both variables with the first element of the array. As we move through the rest of the array, we update `current_sum` to be the maximum of either the current element itself (starting a new subarray) or the sum of the current element and the previous `current_sum` (extending the existing subarray). Then, we compare `current_sum` with `max_sum` and update `max_sum` if needed.

This greedy and dynamic programming-based strategy ensures we always make the optimal choice at each step. Kadane's Algorithm runs in $O(n)$ time with $O(1)$ space, making it the most efficient solution for this problem.

Solution:

```
Python3  Auto
1 class Solution:
2     def maxSubArray(self, nums: list[int]) -> int:
3         current_sum = max_sum = nums[0]
4
5         for num in nums[1:]:
6             current_sum = max(num, current_sum + num)
7             max_sum = max(max_sum, current_sum)
8
9         return max_sum
10
```

Practice here

[Link to the problem](#)

Problem 9 - Maximum Average Subarray I

Problem Statement

You are given an integer array `nums` consisting of `n` elements, and an integer `k`. You need to find the contiguous subarray of length `k` that has the maximum average value and return this maximum average. The answer can be returned as a floating-point number.

Logic Explanation

To solve this problem efficiently, we use the sliding window technique. The goal is to find the subarray of length exactly `k` that has the highest average, which means we want the one with the maximum total sum, since the average is just the sum divided by `k`.

We start by calculating the sum of the first `k` elements in the array. This is our initial window. Then we slide the window one element at a time across the array. At each step:

- We subtract the element that is leaving the window (the leftmost one),
- And we add the new element that is entering the window (the rightmost one).

This gives us the new sum for the current window in constant time. At each move, we check if the current window's sum is greater than the previously recorded maximum sum and update it if needed.

Once we've gone through all windows of size `k`, we divide the maximum sum by `k` to get the maximum average.

This method only requires a single pass through the array, so the time complexity is $O(n)$ and the space complexity is $O(1)$.

Solution:

```
</> Code
Python3  Auto

1 class Solution:
2     def findMaxAverage(self, nums: list[int], k: int) -> float:
3         window_sum = sum(nums[:k])
4         max_sum = window_sum
5
6         for i in range(k, len(nums)):
7             window_sum = window_sum - nums[i - k] + nums[i]
8             max_sum = max(max_sum, window_sum)
9
10        return max_sum / k
11
```

Practice here

[Link to the problem](#)

Problem 10 - Product of Array Except Self

Problem Statement

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 solve it without using division and in $O(n)$ time.

Logic Explanation

The key challenge in this problem is to compute the result without using division and in linear time. To do this, we split the problem into two parts for each element:

- The product of all elements to the left of the current index.
- The product of all elements to the right of the current index.

The final result at index i will be the product of these two parts:

`answer[i] = left_product[i] * right_product[i]`

But instead of using extra arrays for left and right products, we can optimize it using just the output array and a temporary variable.

Step-by-step strategy:

1. First Pass (Left Products):

Initialize the result array `answer` such that `answer[i]` contains the product of all elements to the left of i . We start with `answer[0] = 1`, because there are no elements to the left of index 0. Then for each index i from 1 to $n - 1$, we compute:

`answer[i] = answer[i - 1] * nums[i - 1]`

2. Second Pass (Right Products):

Now we walk the array in reverse to multiply each `answer[i]` by the product of all elements to the right. We use a variable R initialized as 1 to represent the running product from the right. At each step, we do:

`answer[i] *= R`

Then update $R = R * \text{nums}[i]$

This two-pass solution avoids using extra space beyond the output array and a few variables. It runs in $O(n)$ time and uses $O(1)$ extra space (not counting the output array).

Solution:

```
Python3 ▾ 🔒 Auto
1 class Solution:
2     def productExceptSelf(self, nums: list[int]) -> list[int]:
3         n = len(nums)
4         answer = [1] * n
5
6         # First pass: left products
7         for i in range(1, n):
8             answer[i] = answer[i - 1] * nums[i - 1]
9
10        # Second pass: right products
11        R = 1
12        for i in range(n - 1, -1, -1):
13            answer[i] *= R
14            R *= nums[i]
15
16        return answer
17
```

Practice here

[Link to the problem](#)

Problem 11 - Majority Element

Problem Statement

Given an array `nums` of size `n`, return the majority element. The majority element is the element that appears more than $\lfloor n / 2 \rfloor$ times. You may assume that the majority element always exists in the array.

Logic Explanation

To solve this problem efficiently, we use the Boyer-Moore Voting Algorithm, which is specifically designed to find the majority element in linear time and constant space. The idea is to keep track of a candidate element and a counter while scanning through the array.

Here's how the algorithm works:

- Initially, we set the candidate to `None` and the count to 0.
- As we iterate through the array:
 - If count is 0, we set the current number as the new candidate.
 - If the current number is equal to the candidate, we increment count by 1.
 - Otherwise, we decrement count by 1.

The intuition is that the majority element will cancel out all other elements in terms of frequency. Since it appears more than $n / 2$ times, even after pairing it off against all other elements, it will still remain as the final candidate.

This algorithm runs in $O(n)$ time and uses $O(1)$ space, which is optimal for this problem.

Solution:

```
Python3  Auto
1 class Solution:
2     def majorityElement(self, nums: list[int]) -> int:
3         count = 0
4         candidate = None
5
6         for num in nums:
7             if count == 0:
8                 candidate = num
9                 count += 1 if num == candidate else -1
10
11         return candidate
12
```

Practice here

[Link to the problem](#)

Problem 12 - Rotate Array

Problem Statement

Given an integer array `nums`, rotate the array to the right by `k` steps, where `k` is non-negative.

Logic Explanation

The goal is to shift each element in the array `k` steps to the right, wrapping around to the beginning as needed. A naive solution would be to shift elements one-by-one `k` times, which would result in $O(n \times k)$ time complexity and is inefficient for large inputs.

A more optimal solution uses the idea of array reversal. The core insight is that rotating the array to the right by `k` positions is equivalent to:

1. Reversing the entire array,
2. Reversing the first `k` elements,
3. Reversing the remaining `n - k` elements.

Let's break this down with an example:

- Original: [1, 2, 3, 4, 5, 6, 7], `k` = 3
- Step 1: Reverse all → [7, 6, 5, 4, 3, 2, 1]
- Step 2: Reverse first 3 → [5, 6, 7, 4, 3, 2, 1]
- Step 3: Reverse last 4 → [5, 6, 7, 1, 2, 3, 4] (which is the rotated array)

This technique works because rotating right by `k` is like taking the last `k` elements and moving them to the front, while keeping the order of the rest intact. The reversal trick allows us to do this in-place with $O(n)$ time and $O(1)$ space.

We also use `k = k % n` to handle cases where `k` is greater than the array size.

Solution:

```
Python3  Auto
1 class Solution:
2     def rotate(self, nums: list[int], k: int) -> None:
3         n = len(nums)
4         k %= n # In case k > n
5
6         def reverse(start: int, end: int) -> None:
7             while start < end:
8                 nums[start], nums[end] = nums[end], nums[start]
9                 start += 1
10                end -= 1
11
12        # Reverse the whole array
13        reverse(0, n - 1)
14        # Reverse the first k elements
15        reverse(0, k - 1)
16        # Reverse the remaining elements
17        reverse(k, n - 1)
18
```

Practice here

[Link to the problem](#)

Problem 13 - Replace Elements with Greatest Element on Right Side

Problem Statement

Given an array `arr`, replace every element in that array with the greatest element among the elements to its right, and replace the last element with -1. Return the resulting array.

Logic Explanation

The goal is to replace each element with the maximum element to its right, and for the last element, we simply set it to -1. A brute-force approach would be to, for each element, scan all elements to its right to find the maximum. However, this would result in $O(n^2)$ time complexity, which is inefficient for large arrays.

We can solve this more efficiently by scanning the array from right to left. The idea is:

- Start with the last element and initialize a variable `max_right` as -1 (since the last element has no elements to its right).
- Traverse the array in reverse.
- For each element, store its value in a temporary variable, then replace it with the current `max_right`.
- After the replacement, update `max_right` to be the maximum of its current value and the original value of the current element.

This approach ensures that at every step, we are only comparing and updating the maximum value seen so far from the right, without re-scanning the array. The process is done in a single pass from right to left and uses $O(1)$ extra space.

The overall time complexity is $O(n)$ and space complexity is $O(1)$.

Solution:

```
Python3  Auto
1 class Solution:
2     def replaceElements(self, arr: list[int]) -> list[int]:
3         max_right = -1
4
5         for i in range(len(arr) - 1, -1, -1):
6             current = arr[i]
7             arr[i] = max_right
8             max_right = max(max_right, current)
9
10        return arr
11
```


Practice here

[Link to the problem](#)

Problem 14 - Contiguous Array

Problem Statement

Given a binary array `nums`, return the maximum length of a contiguous subarray with an equal number of 0s and 1s.

Logic Explanation

The key idea is to convert this problem into one of finding the longest subarray with sum zero, using a transformation and a hash map.

We begin by converting the binary array so that all 0s are treated as -1. With this transformation:

- A subarray with an equal number of 0s and 1s will have a total sum of 0.

Now the problem reduces to finding the longest subarray in the modified array whose elements sum to zero. This can be efficiently solved using prefix sums and a hash map.

Here's how:

1. Initialize a variable `prefix_sum` to keep track of the running sum.
2. Use a hash map `sum_index_map` to store the first index where each prefix sum is seen.
3. Start by inserting `{0: -1}` into the map to handle the case where a subarray from the beginning has a sum of zero.
4. As we iterate through the array:
 - Convert 0 to -1, and keep 1 as is.
 - Add the current value to `prefix_sum`.
 - If `prefix_sum` has been seen before, it means the subarray from the first occurrence to the current index has a total sum of zero, so we update `max_length`.
 - If `prefix_sum` is not in the map, we store the current index.

This approach works in $O(n)$ time and $O(n)$ space, which is optimal for this problem.

Solution:

```
Python3  Auto

1 class Solution:
2     def findMaxLength(self, nums: list[int]) -> int:
3         prefix_sum = 0
4         max_length = 0
5         sum_index_map = {0: -1} # prefix_sum -> first index
6
7         for i, num in enumerate(nums):
8             # Treat 0 as -1
9             prefix_sum += -1 if num == 0 else 1
10
11             if prefix_sum in sum_index_map:
12                 length = i - sum_index_map[prefix_sum]
13                 max_length = max(max_length, length)
14             else:
15                 sum_index_map[prefix_sum] = i
16
17         return max_length
18
```

Practice here

[Link to the problem](#)

Problem 15 - Minimum Size Subarray Sum

Problem Statement

Given an array of positive integers `nums` and an integer `target`, return the minimal length of a contiguous subarray of which the sum is greater than or equal to `target`. If there is no such subarray, return 0 instead.

Logic Explanation

The goal is to find the shortest possible subarray such that the sum of its elements is greater than or equal to the target. Since all numbers in the array are positive, once a subarray sum exceeds the target, moving the start pointer to the right will only decrease the sum. This is a perfect scenario to apply the sliding window technique.

Here's how the approach works:

1. We use two pointers: `start` and `end` to define a sliding window.
2. We maintain a variable `current_sum` to keep track of the sum of the current window.
3. As we iterate through the array using the `end` pointer:
 - We keep adding `nums[end]` to `current_sum`.
 - Once `current_sum` becomes greater than or equal to `target`, we try to shrink the window from the left (move `start` forward) as long as the condition still holds.
 - Every time the condition is satisfied, we update the `min_length` to the smaller value between the current minimum and the current window size.

This algorithm ensures that each element is processed at most twice: once when expanding the window, and once when shrinking it. So the overall time complexity is $O(n)$, and it uses $O(1)$ extra space.

If no valid subarray is found by the end of the loop, we return 0.

Solution:

```
Python3  Auto
1 class Solution:
2     def minSubArrayLen(self, target: int, nums: list[int]) -> int:
3         start = 0
4         current_sum = 0
5         min_length = float('inf')
6
7         for end in range(len(nums)):
8             current_sum += nums[end]
9
10            while current_sum >= target:
11                min_length = min(min_length, end - start + 1)
12                current_sum -= nums[start]
13                start += 1
14
15            return 0 if min_length == float('inf') else min_length
16
```

Practice here

[Link to the problem](#)

Problem 16 - Sort List

Problem Statement

Given the head of a singly linked list, return the list after sorting it in ascending order.

Logic Explanation

To sort a singly linked list efficiently, we should aim for an **$O(n \log n)$** time complexity with **$O(1)$** auxiliary space (excluding recursion stack). Since we can't randomly access elements like an array, common algorithms like quicksort aren't ideal for linked lists. Instead, we use **merge sort**, which is very well suited for linked lists.

Why Merge Sort?

- Merge sort doesn't require random access; it only needs sequential access, which matches how linked lists work.
- It can be implemented with $O(\log n)$ space due to recursion and $O(n \log n)$ time.
- It splits the list into halves, sorts each half recursively, and merges them back together in sorted order.

Key Steps:

1. **Divide the list into two halves:**
Use the fast and slow pointer technique to find the middle of the list. Slow moves one step at a time, fast moves two steps. When fast reaches the end, slow is at the midpoint.
2. **Recursively sort both halves:**
Apply the same logic on each half recursively until base condition is met (list is empty or has one node).
3. **Merge the sorted halves:**
Use a helper function to merge two sorted linked lists into one sorted list.

This approach ensures that we always divide and merge in $O(n \log n)$ time.

Solution:

```
Python3  ▾  🔒 Auto

1  class ListNode:
2      def __init__(self, val=0, next=None):
3          self.val = val
4          self.next = next
5
6  class Solution:
7      def sortList(self, head: ListNode) -> ListNode:
8          if not head or not head.next:
9              return head
10
11         # Step 1: Find the middle of the list
12         slow, fast = head, head.next
13         while fast and fast.next:
14             slow = slow.next
15             fast = fast.next.next
16
17         mid = slow.next
18         slow.next = None # Split the list into two halves
19
20         # Step 2: Recursively sort both halves
21         left = self.sortList(head)
22         right = self.sortList(mid)
23
24         # Step 3: Merge the sorted halves
25         return self.merge(left, right)
26
27     def merge(self, l1: ListNode, l2: ListNode) -> ListNode:
28         dummy = ListNode()
29         tail = dummy
30
31         while l1 and l2:
32             if l1.val < l2.val:
33                 tail.next = l1
34                 l1 = l1.next
35             else:
36                 tail.next = l2
37                 l2 = l2.next
38             tail = tail.next
39
40         # Attach remaining nodes
41         tail.next = l1 if l1 else l2
42         return dummy.next
```

Practice here

[Link to the problem](#)

Problem 17 - Find Subarray with Given Sum (GeeksforGeeks)

Problem Statement

Given an unsorted array of non-negative integers and a sum S , find a **contiguous subarray** that adds up to the given sum. Return the 1-based starting and ending indices of such a subarray. If there are multiple such subarrays, return the one with the smallest starting index. If no such subarray exists, return -1.

Logic Explanation

The key here is that all array elements are non-negative. This allows us to use the sliding window technique effectively.

Idea:

- We maintain a sliding window with two pointers: start and end.
- We keep track of the sum of the current window (current_sum).
- As we move the end pointer forward:
 - Add $\text{arr}[\text{end}]$ to current_sum.
 - If current_sum exceeds the target sum S , we move the start pointer to the right (shrinking the window from the left), and subtract $\text{arr}[\text{start}]$ from current_sum.
 - If current_sum becomes equal to S , we return the current subarray's 1-based start + 1 and end + 1.

This method works in $O(n)$ time and $O(1)$ space, since each element is visited at most twice.

Why it works only for non-negative numbers?

Because with non-negative numbers, once the sum becomes too big, shrinking the window is guaranteed to decrease the sum. This wouldn't be reliable with negative numbers

Solution:

```
class Solution:
    def subArraySum(self, arr, n, s):
        start = 0
        current_sum = 0

        for end in range(n):
            current_sum += arr[end]

            # Shrink window from the left while sum is greater than s
            while current_sum > s and start < end:
                current_sum -= arr[start]
                start += 1

            # Check for target match
            if current_sum == s:
                return [start + 1, end + 1] # 1-based indexing

        return [-1]
```

Problem 18 - Count Positive and Negative Numbers in a List (GeeksforGeeks)

Problem Statement

Given a list of integers, count how many numbers are positive and how many are negative. Zero is neither positive nor negative and should be ignored.

Logic Explanation

To solve this problem, we need to iterate through each element in the list and:

- Increment a counter if the number is greater than zero (positive).
- Increment a different counter if the number is less than zero (negative).
- Ignore the number if it is exactly zero.

This can be easily done with a single loop and two variables (positive_count and negative_count). We do not need any sorting or extra space.

The time complexity is **O(n)** where n is the number of elements in the list, and the space complexity is **O(1)**.

Solution:

```
def count_positive_negative(nums):  
    positive_count = 0  
    negative_count = 0  
  
    for num in nums:  
        if num > 0:  
            positive_count += 1  
        elif num < 0:  
            negative_count += 1  
  
    return positive_count, negative_count
```

Example Usage:

```
nums = [2, -3, 0, 5, -1, 7, -6]  
pos, neg = count_positive_negative(nums)  
print("Positive numbers:", pos)  
print("Negative numbers:", neg)
```


Output:

```
Positive numbers: 3
Negative numbers: 3
```

Problem 19 - Find Second Largest Number in a List (GeeksforGeeks)

Problem Statement

Given a list of integers, find and return the second largest unique number in the list. If there is no such number (i.e., all elements are the same or the list has fewer than two unique values), handle the condition appropriately.

Logic Explanation

To find the second largest number efficiently, we want to do it in a single pass without sorting the list, which would take extra time. The optimal approach involves tracking:

- first: the largest number seen so far.
- second: the second largest number seen so far.

We initialize both to None (or float negative infinity) and iterate over the list:

1. If the current number is greater than first, then update second to be first, and set first to the current number.
2. Else if the current number is smaller than first but greater than second, and not equal to first, then update second.

This ensures we capture the second largest unique number.

This approach runs in $O(n)$ time and uses $O(1)$ space.

Solution:

```
def find_second_largest(nums):
    if len(nums) < 2:
        return None # Not enough elements

    first = second = float('-inf')

    for num in nums:
        if num > first:
            second = first
            first = num
        elif first > num > second:
            second = num

    return second if second != float('-inf') else None
```

Example Usage:

```
nums = [12, 35, 1, 10, 34, 1]
result = find_second_largest(nums)
if result is not None:
    print("Second largest number:", result)
else:
    print("Second largest number not found.")
```

Output:

```
Second largest number: 34
```

Problem 20 - Reverse a List in Python (GeeksforGeeks)

Problem Statement

Given a list of elements, reverse the list so that the last element becomes the first, the second last becomes the second, and so on.

Logic Explanation

Reversing a list means rearranging its elements in the opposite order. Python offers several ways to do this, ranging from built-in slicing to in-place swaps.

Let's look at three commonly used and efficient approaches:

1. Using Slicing [::-1] (Most Pythonic)

This is the simplest and most readable way.

`list[::-1]` creates a reversed copy of the original list.

It does not modify the original list unless explicitly assigned.

```
def reverse_list_slicing(lst):
    return lst[::-1]
```

2. Using the reverse() Method (In-Place)

This reverses the list **in-place**, meaning the original list is modified.

It returns None.

```
def reverse_list_in_place(lst):
    lst.reverse()
    return lst
```

3. Using the reversed() Function

This returns an **iterator**, so we need to convert it back to a list.
Does **not modify** the original list.

```
def reverse_list_reversed(lst):  
    return list(reversed(lst))
```

4. Manual Two-Pointer Swap (In-Place)

Useful when you want full control or are asked to implement it manually in interviews.

```
def reverse_list_manual(lst):  
    left = 0  
    right = len(lst) - 1  
  
    while left < right:  
        lst[left], lst[right] = lst[right], lst[left]  
        left += 1  
        right -= 1  
  
    return lst
```

Example Usage:

```
nums = [10, 20, 30, 40, 50]  
  
print("Slicing:", reverse_list_slicing(nums))  
print("In-place:", reverse_list_in_place(nums[:])) # Use copy to preserve original  
print("Reversed():", reverse_list_reversed(nums))  
print("Manual:", reverse_list_manual(nums[:]))
```

Problem 21 - Count Primes

Problem Statement

Given an integer n , return the number of prime numbers that are less than n .

Logic Explanation

To solve this efficiently, we use the Sieve of Eratosthenes algorithm, which is a classic technique for finding all prime numbers less than a given number n .

What is a Prime Number?

A number greater than 1 that has no divisors other than 1 and itself.

So, we want to count how many such numbers exist in the range $[2, n-1]$.

Why not check each number one by one?

A brute-force solution would check each number for primality individually, which is slow - $O(n\sqrt{n})$.

But the Sieve of Eratosthenes improves this to $O(n \log \log n)$ time and $O(n)$ space.

Sieve of Eratosthenes – How it Works:

1. Create a boolean list `is_prime` of size n , initially set to `True`.
(We'll mark `False` for non-prime numbers.)
2. Mark 0 and 1 as not prime.
3. Starting from 2, for each number:
 - If it's still marked `True`, it's a prime.
 - Then mark all multiples of that number (like $i*2$, $i*3$, etc.) as `False` since they can't be prime.
4. Continue this process up to the square root of n .
(Because if $i * i > n$, all smaller factors have already been handled.)

Finally, we count how many `True` values remain in the `is_prime` list.

Solution:

```
Python3  Auto
1 class Solution:
2     def countPrimes(self, n: int) -> int:
3         if n <= 2:
4             return 0
5
6         is_prime = [True] * n
7         is_prime[0] = is_prime[1] = False
8
9         for i in range(2, int(n**0.5) + 1):
10            if is_prime[i]:
11                for j in range(i * i, n, i):
12                    is_prime[j] = False
13
14            return sum(is_prime)
15
```

Example:

n = 10

Prime numbers less than 10 are: 2, 3, 5, 7

Output: 4

Practice here

[Link to the problem](#)

STRING

Problem 22 - Longest Substring Without Repeating Characters

Problem Statement

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

Logic Explanation

The goal is to find the longest sequence of characters in the string such that all characters are unique and appear only once. Since we're looking at substrings (which must be contiguous), and characters are being added and removed based on duplicates, a sliding window approach works best here.

We use two pointers, start and end, to define the current window in the string. We also use a hash map (or set) to store the characters currently in the window and their indices for quick lookup.

Here's how the logic works:

- Move the end pointer one step at a time, adding the character at end to the window.
- If that character has already appeared (i.e., it's in the map and its index is \geq start), we found a duplicate. To fix this, we move start to the position right after the previous occurrence of the duplicate character. This shrinks the window from the left to make it valid again.
- Keep updating the maximum window length as we go.

This method guarantees that:

- Every character is added and removed from the window at most once.
- The time complexity is $O(n)$ because we only traverse the string once.
- The space complexity is $O(\min(n, m))$, where m is the size of the character set (e.g., 26 for lowercase letters, 128 for ASCII).

Solution:

```
Python3  ▾  🔒 Auto
1  class Solution:
2      def lengthOfLongestSubstring(self, s: str) -> int:
3          char_index = {}
4          max_length = 0
5          start = 0
6
7          for end in range(len(s)):
8              if s[end] in char_index and char_index[s[end]] >= start:
9                  start = char_index[s[end]] + 1
10
11             char_index[s[end]] = end
12             max_length = max(max_length, end - start + 1)
13
14         return max_length
15
```

Practice here

[Link to the problem](#)

Problem 23 - Valid Parentheses

Problem Statement

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

1. Open brackets are closed by the same type of brackets.
2. Open brackets are closed in the correct order.
3. Every closing bracket has a corresponding open bracket of the same type.

Logic Explanation

To check if a sequence of parentheses is valid, we need to match every opening bracket with the correct type of closing bracket and in the correct order. The ideal data structure for this is a stack because it works in a Last-In-First-Out (LIFO) manner — the most recently opened bracket must be the first one to close.

Here's the step-by-step logic:

1. We loop through each character in the string.
2. If the character is an opening bracket ((), {}, []), we push it onto the stack.
3. If the character is a closing bracket ((), },]), we check:
 - If the stack is empty \rightarrow it's invalid because there's no opening bracket to match it.
 - Otherwise, we pop from the stack and check if the popped opening bracket matches the current closing bracket using a mapping.
4. If at any point the match fails, we return False.
5. At the end, the stack should be empty — meaning all opened brackets were closed correctly. If not, return False.

This approach ensures linear time complexity $O(n)$ and space complexity $O(n)$ in the worst case (if all characters are opening brackets).

Solution:

```
Python3  Auto
1 class Solution:
2     def isValid(self, s: str) -> bool:
3         stack = []
4         bracket_map = {'(': ')', '[': ']', '{': '}'}
5
6         for char in s:
7             if char in bracket_map:
8                 if not stack or stack.pop() != bracket_map[char]:
9                     return False
10            else:
11                stack.append(char)
12
13        return not stack
```

Practice here

[Link to the problem](#)

Problem 24 - Remove All Occurrences of a Substring

Problem Statement

Given two strings *s* and *part*, repeatedly remove all occurrences of the substring *part* from *s* until *s* no longer contains *part*. Return the final string after all such removals.

Logic Explanation

The task is to repeatedly remove all occurrences of a specific substring *part* from the main string *s* until that substring no longer exists in *s*. This is a classic example of a greedy and iterative string manipulation problem.

There are two main ways to solve this:

1. Built-in `replace()` method with a while loop — check and replace as long as *part* exists.
2. Stack-based approach — useful when *s* is very long and frequent slicing becomes inefficient.

We'll go with the stack-based approach, which is both clean and efficient:

- We iterate through each character of *s* one by one.
- We use a list as a stack to build the resulting string.
- For each character, we add it to the stack.
- After every addition, we check whether the last few characters in the stack match *part*. If they do, we remove those characters from the stack.
- Finally, we join the stack to return the final string.

This method avoids repeatedly scanning and slicing the string like in a while `s.find(part)` loop, making it more optimal for larger inputs.

Time complexity is $O(n \times m)$ in worst case where *n* is the length of *s* and *m* is the length of *part*, but in practice, the stack-based approach performs faster due to reduced re-scanning.

Solution:

```
Python3  Auto
1 class Solution:
2     def removeOccurrences(self, s: str, part: str) -> str:
3         stack = []
4         part_len = len(part)
5
6         for char in s:
7             stack.append(char)
8             # Check if the last characters in stack form the 'part'
9             if len(stack) >= part_len and ''.join(stack[-part_len:]) == part:
10                 # Remove the matched part
11                 for _ in range(part_len):
12                     stack.pop()
13
14         return ''.join(stack)
```

Practice here

[Link to the problem](#)

Problem 25 - Find First Non-Repeating Character in a String

Problem Statement

Given a string consisting of lowercase or uppercase letters, find the first non-repeating character and return it. If there is no such character, return a special value or message indicating that.

Logic Explanation

To solve this problem, we need to find the first character in the string that appears only once — meaning it is non-repeating. The simplest and most efficient way to do this is by using a two-pass approach:

Step 1: Count Frequencies

- First, we traverse the string and count how many times each character appears.
- We can use a dictionary (or an array of size 26 or 128 if we restrict ourselves to ASCII letters) to store the frequency of each character.

Step 2: Find First Unique Character

- In a second pass over the string, we check each character in order and return the first one whose count is exactly 1.

This approach ensures that we preserve the order of characters and efficiently identify the first non-repeating one.

The time complexity is $O(n)$ and the space complexity is $O(1)$ assuming a fixed character set (like lowercase English letters or ASCII). If we consider Unicode characters, space becomes $O(k)$, where k is the number of unique characters.

Solution:

```
def first_non_repeating_char(s):  
    freq = {}  
  
    # Step 1: Count frequency of each character  
    for char in s:  
        freq[char] = freq.get(char, 0) + 1  
  
    # Step 2: Find the first character with frequency 1  
    for char in s:  
        if freq[char] == 1:  
            return char  
  
    return None # Or any special value like '_', -1, etc.
```

Example Usage:

```
s = "geeksforgeeks"
result = first_non_repeating_char(s)
if result:
    print("First non-repeating character is:", result)
else:
    print("No non-repeating character found.")
```

Output:

```
First non-repeating character is: f
```

Practice here

<https://www.geeksforgeeks.org/dsa/given-a-string-find-its-first-non-repeating-character/>

Problem 26 - Longest Common Prefix

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 "".

Logic Explanation

The goal is to find the longest starting substring (prefix) that is common to all strings in the input list.

A simple and efficient way to do this is by using vertical scanning:

1. We take the first string as a reference and iterate through each of its characters.
2. For each character position, we check whether all other strings have the same character at that position.
3. If we find a mismatch at any point, we return the prefix found so far.
4. If we finish scanning the entire first string without any mismatch, then the whole first string is the common prefix.

This solution works efficiently because we stop checking as soon as a mismatch is found. It also guarantees that we never check beyond the shortest string length.

The time complexity is $O(n \times m)$ where:

- n is the number of strings,
- m is the length of the shortest string (worst case when all strings are the same).

This is optimal given that we must compare characters until we hit a mismatch.

Solution:

```
Python3  Auto
1 class Solution:
2     def longestCommonPrefix(self, strs: list[str]) -> str:
3         if not strs:
4             return ""
5
6         for i in range(len(strs[0])):
7             char = strs[0][i]
8             for s in strs[1:]:
9                 if i >= len(s) or s[i] != char:
10                    return strs[0][:i]
11
12         return strs[0]
13
```

Practice here

[Link to the problem](#)

Problem 27 - Length of Last Word

Problem Statement

Given a string *s* consisting of words and spaces, return the length of the last word in the string. A word is a maximal substring consisting of non-space characters only.

Logic Explanation

To find the length of the last word, we need to handle cases where:

- The string may have trailing spaces.
- The last word can be anywhere in the string, and we need to ignore all spaces after the last word.

Here's how we can do it efficiently:

1. Start from the end of the string and skip all trailing spaces.
2. Once we find a non-space character, start counting until we hit a space or the beginning of the string.
3. The count at this point will be the length of the last word.

This approach avoids extra space and avoids splitting the string, making it faster for long inputs.

Time complexity is $O(n)$ where n is the length of the string, and space complexity is $O(1)$.

Solution:

```
Python3  Auto
1 class Solution:
2     def lengthOfLastWord(self, s: str) -> int:
3         i = len(s) - 1
4         length = 0
5
6         # Skip trailing spaces
7         while i >= 0 and s[i] == ' ':
8             i -= 1
9
10        # Count characters in the last word
11        while i >= 0 and s[i] != ' ':
12            length += 1
13            i -= 1
14
15        return length
16
```

Practice here

[Link to the problem](#)

Problem 28 - Reverse String

Problem Statement

Write a function that reverses a string. The input string is given as an array of characters `s`. You must do this by modifying the input array in-place with $O(1)$ extra memory.

Logic Explanation

Since the string is given as a **list of characters** and we must reverse it **in-place**, the best way to approach this is with the **two-pointer technique**.

Here's the step-by-step idea:

1. Use two pointers — one starting at the beginning (left) and one at the end (right) of the list.
2. Swap the characters at these two positions.
3. Move the left pointer forward and the right pointer backward.
4. Continue this process until the two pointers meet or cross.

Because each character is swapped only once, the time complexity is **$O(n)$** , where n is the length of the string. The space complexity is **$O(1)$** since we do not use any extra data structures (just a few variables).

This is the most optimal and direct solution given the in-place requirement.

Solution:

```
Python3  Auto
1 class Solution:
2     def reverseString(self, s: list[str]) -> None:
3         left, right = 0, len(s) - 1
4
5         while left < right:
6             s[left], s[right] = s[right], s[left]
7             left += 1
8             right -= 1
9
```

Practice here

[Link to the problem](#)

Problem 29 - Minimum Window Substring

Problem Statement

Given two strings s and t of lengths m and n respectively, return the minimum window substring of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return the empty string `""`.

Logic Explanation

This is a classic sliding window + hash map problem. The challenge is to find the smallest window in s that contains all characters of t with correct frequencies, and do so efficiently in linear time.

Here's the step-by-step idea:

1. Track the frequency of each character in t using a dictionary called `need`.
2. Use two pointers, `left` and `right`, to create a sliding window over string s .
3. Expand the `right` pointer to include characters from s into the window and update a window frequency map accordingly.
4. When the current window contains all the characters from t (i.e., when the current window satisfies the need), try shrinking the window from the left to find the smallest valid window.
5. During this shrinking phase, keep checking if the current window is the smallest seen so far and update the result accordingly.
6. Continue expanding and shrinking the window until the end of s is reached.

To check if the current window satisfies t , we maintain a variable `have` that counts how many unique characters in the window meet the required frequency. When `have == need_count`, we know the window is valid.

This algorithm runs in $O(m + n)$ time because each character is processed at most twice (once added, once removed from the window), and uses $O(n)$ space for the frequency maps.

Solution:

```
Python3 v Auto
1 from collections import Counter
2
3 class Solution:
4     def minWindow(self, s: str, t: str) -> str:
5         if not s or not t:
6             return ""
7
8         need = Counter(t)
9         window = {}
10        have, need_count = 0, len(need)
11        left = 0
12        min_len = float('inf')
13        result = ""
14
15        for right in range(len(s)):
16            char = s[right]
17            window[char] = window.get(char, 0) + 1
18
19            if char in need and window[char] == need[char]: # <-- FIXED: added colon
20                have += 1
21
22            while have == need_count:
23                # Update result if current window is smaller
24                window_len = right - left + 1
25                if window_len < min_len:
26                    min_len = window_len
27                    result = s[left:right+1]
28
29                # Shrink window from the left
30                window[s[left]] -= 1
31                if s[left] in need and window[s[left]] < need[s[left]]:
32                    have -= 1
33                left += 1
34
35        return result
36
```

Practice here

[Link to the problem](#)

Problem 30 - Valid Anagram

Problem Statement

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

Logic Explanation

To determine whether two strings are anagrams, we need to check that:

- They contain the same characters
- Each character occurs the same number of times in both strings

There are multiple ways to do this, but the most efficient and clean method is to use a hash map (dictionary) to count character frequencies.

Here's the step-by-step logic:

1. If the lengths of the two strings are not the same, they cannot be anagrams.
2. Use a dictionary (or `collections.Counter`) to count the frequency of each character in both strings.
3. Compare the two dictionaries — if they match exactly, the strings are anagrams.

This solution has time complexity $O(n)$, where *n* is the length of the strings, and space complexity $O(1)$ if we assume the character set is fixed (like lowercase English letters).

Solution:

```
Python3  Auto
1  from collections import Counter
2
3  class Solution:
4      def isAnagram(self, s: str, t: str) -> bool:
5          if len(s) != len(t):
6              return False
7          return Counter(s) == Counter(t)
8
```

Alternatively, if you want to do it manually without Counter, here's a version using a single dictionary:

```
1 class Solution:
2     def isAnagram(self, s: str, t: str) -> bool:
3         if len(s) != len(t):
4             return False
5
6         count = {}
7
8         for char in s:
9             count[char] = count.get(char, 0) + 1
10
11        for char in t:
12            if char not in count or count[char] == 0:
13                return False
14            count[char] -= 1
15
16        return True
17
```

Practice here

[Link to the problem](#)

Problem 31 - Most Common Word

Problem Statement

Given a string paragraph and a list of strings banned, return the most frequent word that is not banned. It is guaranteed there is at least one word that isn't banned, and that the answer is unique. Words in paragraph are case-insensitive and consist of letters, punctuation, and spaces. The answer should be returned in lowercase.

Logic Explanation

The task is to find the most frequent word in the paragraph that is not in the banned list. The core steps for solving this problem are:

1. Clean and normalize the input:
 - Convert the entire paragraph to lowercase to handle case-insensitivity.
 - Remove all punctuation so we only deal with words.
 - Split the paragraph into individual words.
2. Filter banned words:
 - Convert the banned list to a set for faster lookups.
 - Ignore any word that is present in this banned set.
3. Count frequencies:
 - Use a dictionary (or collections.Counter) to count how many times each valid word appears.
4. Find the most frequent valid word:
 - Traverse the word frequency dictionary and return the word with the highest count.

The time complexity is $O(n)$ where n is the number of characters in the paragraph, and space complexity is $O(k)$ where k is the number of unique non-banned words.

Solution:

```
Python3 ▾ 🔒 Auto
1 import re
2 from collections import Counter
3
4 class Solution:
5     def mostCommonWord(self, paragraph: str, banned: list[str]) -> str:
6         # Normalize the paragraph: lowercase and remove punctuation
7         words = re.findall(r'\w+', paragraph.lower())
8
9         banned_set = set(banned)
10        valid_words = [word for word in words if word not in banned_set]
11
12        counts = Counter(valid_words)
13        return counts.most_common(1)[0][0]
14
```

Practice here

[Link to the problem](#)

Problem 32 – Find the Index of the First Occurrence in a String

Problem Statement

Given two strings haystack and needle, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Logic Explanation

This problem is about substring search — we need to find the first position in haystack where needle fully matches. If no match is found, we return -1.

There are multiple ways to solve this, including advanced algorithms like KMP (Knuth-Morris-Pratt), but for most real-world and interview purposes, a sliding window comparison works well and is easy to implement.

Here's the step-by-step logic for the sliding window approach:

1. Let n be the length of haystack and m be the length of needle.
2. We only need to iterate from index 0 to $n - m$ in haystack because a match can't start beyond that.
3. At each index i , take a substring of length m from haystack starting at i , and compare it to needle.
4. If a match is found, return i immediately.
5. If no match is found throughout the loop, return -1.

This has time complexity $O(n \times m)$ in the worst case (especially when repeated characters are involved), and space complexity is $O(1)$ if slicing is done carefully.

Solution:

```
Python3  Auto
1 class Solution:
2     def strStr(self, haystack: str, needle: str) -> int:
3         if not needle:
4             return 0
5
6         for i in range(len(haystack) - len(needle) + 1):
7             if haystack[i:i + len(needle)] == needle:
8                 return i
9
10        return -1
11
```

Practice here

[Link to the problem](#)

Problem 33 - Remove Characters from the First String Which Are Present in the Second String

Problem Statement

Given two strings, the task is to remove all characters from the first string that are present in the second string. Return the updated first string after these removals.

Logic Explanation

We are given two strings:

- str1: The base string from which characters will be removed
- str2: The string containing characters to be removed from str1

The goal is to remove all characters from str1 that appear in str2.

To solve this efficiently:

1. We first convert str2 into a set, which allows for $O(1)$ time character lookups.
2. Then, we iterate through each character in str1, and if it's not in the set, we keep it.
3. Finally, we join and return the characters that are kept.

This avoids nested loops and ensures the solution is efficient even for large inputs.

- Time complexity: $O(n + m)$ where n is the length of str1, and m is the length of str2.
- Space complexity: $O(m + n)$ for the result and the set of removable characters.

Solution:

```
def removeChars(str1, str2):  
    remove_set = set(str2) # Characters to remove  
    result = []  
  
    for char in str1:  
        if char not in remove_set:  
            result.append(char)  
  
    return ''.join(result)
```

Example:

```
str1 = "computer"  
str2 = "cat"  
print(removeChars(str1, str2)) # Output: "ompuer"
```

Practice here - <https://www.geeksforgeeks.org/dsa/remove-characters-from-the-first-string-which-are-present-in-the-second-string/>

Problem 34 – Longest Palindromic Substring

Problem Statement

Given a string s , return the longest palindromic substring in s .

Logic Explanation

A palindrome is a string that reads the same forward and backward. To solve this problem, we must identify the longest substring in the given string s that satisfies this property.

The most intuitive and optimized approach (without using dynamic programming) is the Expand Around Center technique.

Why Expand Around Center?

Any palindrome mirrors around its center. So for each character (or pair of characters) in the string, we can try to expand outward to check for palindromes:

- For odd-length palindromes: consider a single character as the center (e.g., 'racecar')
- For even-length palindromes: consider two adjacent characters as the center (e.g., 'abba')

Step-by-step logic:

1. Loop through each character in the string.
2. For each index, expand around it as a single center and as a pair center.
3. While expanding, keep checking if the characters to the left and right are the same.
4. Track the longest palindrome found during all expansions.
5. Finally, return the longest palindromic substring.

This approach avoids recomputation and skips over the need to check all substrings explicitly.

- Time complexity: $O(n^2)$ — due to expanding around each center
- Space complexity: $O(1)$ — no extra space used, just pointers and indices

Solution:

```
1 class Solution:
2     def longestPalindrome(self, s: str) -> str:
3         if not s or len(s) == 1:
4             return s
5
6         start, end = 0, 0
7
8         for i in range(len(s)):
9             # Odd-length palindrome
10            len1 = self.expandAroundCenter(s, i, i)
11            # Even-length palindrome
12            len2 = self.expandAroundCenter(s, i, i + 1)
13
14            max_len = max(len1, len2)
15
16            if max_len > end - start:
17                start = i - (max_len - 1) // 2
18                end = i + max_len // 2
19
20            return s[start:end + 1]
21
22     def expandAroundCenter(self, s: str, left: int, right: int) -> int:
23         while left >= 0 and right < len(s) and s[left] == s[right]:
24             left -= 1
25             right += 1
26         return right - left - 1
27
```

Practice here

[Link to the problem](#)

LINKED LIST

Problem 35 - Add Two Numbers

Problem Statement

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list. You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Logic Explanation

We are simulating the addition of two numbers stored in reverse as linked lists. Each node in the list contains one digit. Since the digits are reversed, we can simply start from the head of each list and add corresponding digits, just like how we do elementary addition from the rightmost digit.

We use a dummy node to keep track of the result list and a pointer called current to keep adding new nodes. At each step, we add digits from both lists (if available) along with any carry from the previous step. If a list is shorter, we treat its missing digits as zero. If the sum at any position is 10 or more, we set $\text{carry} = \text{sum} // 10$, and the current node's value becomes $\text{sum} \% 10$.

After finishing both lists, we check if a non-zero carry is left. If yes, we add a final node with that carry.

This method works in linear time and constant space (excluding the output list) and handles numbers of any length.

Solution:

```
Python3  Auto
1 class Solution:
2     def addTwoNumbers(self, l1, l2):
3         dummy = ListNode()
4         current = dummy
5         carry = 0
6
7         while l1 or l2 or carry:
8             val1 = l1.val if l1 else 0
9             val2 = l2.val if l2 else 0
10
11             total = val1 + val2 + carry
12             carry = total // 10
13             current.next = ListNode(total % 10)
14
15             current = current.next
16             if l1: l1 = l1.next
17             if l2: l2 = l2.next
18
19         return dummy.next
```

Practice here - <https://leetcode.com/problems/add-two-numbers/description/>

Problem 36 - Remove Nth Node From End of List

Problem Statement

Given the head of a linked list, remove the nth node from the end of the list and return its head.

Logic Explanation

To remove the nth node from the end, we need to locate that node efficiently without making multiple full passes through the list. A straightforward way would be to first compute the length of the list, calculate the position of the node from the start, and then traverse again to remove it. However, this requires two passes.

A better approach is to use the two-pointer technique — also called the fast and slow pointer method — which completes the task in just one pass.

We start by creating a dummy node that points to the head of the list. This simplifies edge cases like when the head itself needs to be removed. Then we initialize two pointers, both starting at the dummy node.

We first move the fast pointer n steps ahead. Now the gap between the fast and slow pointers is n nodes. Then we move both pointers one step at a time until the fast pointer reaches the end of the list. At this point, the slow pointer is positioned right before the node we need to delete.

We then adjust the slow pointer's next reference to skip over the node to be removed. Finally, we return the list starting from dummy.next which reflects any potential changes to the head.

This method ensures one full traversal and uses constant extra space.

Solution:

```
Python3  Auto
1 class Solution:
2     def removeNthFromEnd(self, head, n):
3         dummy = ListNode(0, head)
4         fast = slow = dummy
5
6         # Move fast n steps ahead
7         for _ in range(n):
8             fast = fast.next
9
10        # Move both pointers until fast reaches the end
11        while fast.next:
12            fast = fast.next
13            slow = slow.next
14
15        # Skip the node
16        slow.next = slow.next.next
17
18        return dummy.next
```

Practice here

<https://leetcode.com/problems/remove-nth-node-from-end-of-list/description/>

Problem 37 - Odd Even Linked List

Problem Statement

Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list. The relative order of the odd and even nodes should be preserved.

Logic Explanation

This problem requires rearranging the linked list such that all nodes at odd indices come first, followed by all nodes at even indices. Here, index refers to the node's position in the list (starting from 1), not the node's value.

The key idea is to:

1. Use two pointers — one for odd-positioned nodes and one for even-positioned nodes.
2. Traverse the list once and connect the odd nodes to each other and the even nodes to each other in parallel.
3. After the traversal, link the last node of the odd list to the head of the even list.

We start by saving the head of the even list (`even_head`) because we will need it later to attach it after the odd list. Then we move both pointers step by step, skipping alternate nodes. This way, we separate the list into two parts: one containing all odd-indexed nodes and the other all even-indexed nodes, while maintaining their original order.

Finally, we link the last odd node to the even list's head and return the modified list.

This solution works in a single pass with $O(1)$ space and $O(n)$ time.

Solution:

```
Python3  Auto
1  class Solution:
2      def oddEvenList(self, head):
3          if not head or not head.next:
4              return head
5
6          odd = head
7          even = head.next
8          even_head = even
9
10         while even and even.next:
11             odd.next = even.next
12             odd = odd.next
13
14             even.next = odd.next
15             even = even.next
16
17         odd.next = even_head
18         return head
19
```

Practice here

<https://leetcode.com/problems/odd-even-linked-list/description/>

Problem 38 - Remove Duplicates from Sorted List

Problem Statement

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.

Logic Explanation

Since the input linked list is already sorted, all duplicate elements will appear consecutively. This simplifies our task because we only need to check each node and compare it with the next node. If the current node's value is the same as the next node's value, we skip the next node by adjusting the next pointer to point to the node after the duplicate. If the values are different, we move to the next node as usual.

We start with the head of the list and iterate until we reach the end. There is no need to create a new list — we modify the original list in place.

This approach ensures that each node is visited exactly once, so it runs in linear time. It also uses constant space since no extra structures are needed.

Solution:

```
Python3  Auto
1 class Solution:
2     def deleteDuplicates(self, head):
3         current = head
4
5         while current and current.next:
6             if current.val == current.next.val:
7                 current.next = current.next.next
8             else:
9                 current = current.next
10
11         return head
12
```

Practice here

<https://leetcode.com/problems/remove-duplicates-from-sorted-list/description/>

Problem 39 - Intersection of Two Linked Lists

Problem Statement

Given the heads of two singly linked lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection, return null.

Logic Explanation

To find the intersection point of two singly linked lists, we need to identify the node where both lists begin to share the same reference. That means from a certain node onward, both lists are identical (i.e., the nodes are the exact same objects in memory).

A clever and efficient approach is to use two pointers, one for each list. We traverse both lists simultaneously:

- When pointer A reaches the end of list A, we move it to the start of list B.
- Similarly, when pointer B reaches the end of list B, we move it to the start of list A.

This technique ensures that both pointers traverse equal lengths overall. If the two lists intersect, the pointers will meet at the intersection node after at most two passes. If they don't intersect, both pointers will become null at the same time, and we return null.

This solution runs in linear time and uses constant space, with no need to calculate lengths or use extra memory.

Solution:

```
Python3  Auto
1 class Solution:
2     def getIntersectionNode(self, headA, headB):
3         if not headA or not headB:
4             return None
5
6         ptrA, ptrB = headA, headB
7
8         while ptrA != ptrB:
9             ptrA = ptrA.next if ptrA else headB
10            ptrB = ptrB.next if ptrB else headA
11
12        return ptrA
13
```

Practice here

<https://leetcode.com/problems/intersection-of-two-linked-lists/description/>

Problem 40 - Remove Linked List Elements

Problem Statement

Given the head of a linked list and an integer val, remove all the nodes of the linked list that have node.val == val, and return the new head.

Logic Explanation

In this problem, we are asked to remove all nodes from a singly linked list that match a given value. This can involve removing nodes from any part of the list — the head, middle, or end — and possibly all nodes.

To handle all cases uniformly, we use a dummy node. The dummy node is placed before the actual head of the list and points to the current head. This simplifies edge cases where the head itself might need to be removed.

We use a pointer called current, initially pointing to the dummy node. We traverse the list using this pointer and check the value of current.next. If current.next.val matches the target value, we skip the node by adjusting the next pointer to current.next.next. If it doesn't match, we simply move current one step forward.

By the end of the loop, all nodes with the given value are removed, and we return dummy.next as the new head of the list. This approach ensures the solution is clean and efficient.

The solution runs in $O(n)$ time and uses $O(1)$ extra space.

Solution:

```
1 class Solution:
2     def removeElements(self, head, val):
3         dummy = ListNode(0)
4         dummy.next = head
5         current = dummy
6
7         while current.next:
8             if current.next.val == val:
9                 current.next = current.next.next
10            else:
11                current = current.next
12
13        return dummy.next
14
```

Practice here

<https://leetcode.com/problems/remove-linked-list-elements/description/>

Problem 41 - Palindrome Linked List

Problem Statement

Given the head of a singly linked list, return true if the linked list is a palindrome, otherwise return false.

Logic Explanation

To determine if a linked list is a palindrome, we need to compare the first half of the list with the reversed second half. A palindrome reads the same forward and backward, so if both halves match in order, the list is a palindrome.

The most efficient way to do this without using extra space is:

1. **Find the middle of the list** using the fast and slow pointer technique. The slow pointer moves one step at a time while the fast pointer moves two steps. When the fast pointer reaches the end, the slow pointer will be at the middle.
2. **Reverse the second half of the list** starting from the middle node. We do this in-place by modifying the next pointers of the nodes.
3. **Compare the first half and the reversed second half.** We start one pointer from the beginning of the list and the other from the head of the reversed half. If all corresponding nodes match, it's a palindrome.
4. Optionally, **restore the original list structure** by reversing the second half again, though the problem does not require this.

This approach avoids using additional space for arrays or stacks and completes the task in linear time.

Time complexity is $O(n)$, and space complexity is $O(1)$ since we are modifying the list in place.

Solution:

Python3   Auto

```
1 class Solution:
2     def isPalindrome(self, head):
3         if not head or not head.next:
4             return True
5
6         # Step 1: Find the middle
7         slow, fast = head, head
8         while fast and fast.next:
9             slow = slow.next
10            fast = fast.next.next
11
12        # Step 2: Reverse the second half
13        prev = None
14        while slow:
15            next_temp = slow.next
16            slow.next = prev
17            prev = slow
18            slow = next_temp
19
20        # Step 3: Compare first and second halves
21        first, second = head, prev
22        while second:
23            if first.val != second.val:
24                return False
25            first = first.next
26            second = second.next
27
28        return True
29
```

Practice here

<https://leetcode.com/problems/palindrome-linked-list/description/>

Problem 42 - Middle of the Linked List

Problem Statement

Given the head of a singly linked list, return the middle node of the linked list. If there are two middle nodes, return the second middle node.

Logic Explanation

To find the middle node of a singly linked list efficiently, we use the two-pointer technique: a **slow** pointer and a **fast** pointer. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time.

As the fast pointer moves twice as quickly, when it reaches the end of the list, the slow pointer will be at the middle. This technique works whether the length of the list is odd or even:

- If the list has an odd number of nodes, the slow pointer will stop at the exact middle.
- If the list has an even number of nodes, the slow pointer will stop at the **second middle** node, which satisfies the problem's condition.

This approach completes the task in a single pass and uses constant space.

Time complexity is $O(n)$, and space complexity is $O(1)$.

Solution:

```
Python3  Auto
1 class Solution:
2     def middleNode(self, head):
3         slow = fast = head
4
5         while fast and fast.next:
6             slow = slow.next
7             fast = fast.next.next
8
9         return slow
10
```

Practice here

<https://leetcode.com/problems/middle-of-the-linked-list/description/>

Problem 43 - Partition List

Problem Statement

Given the head of a linked list and a value x , partition it such that all nodes less than x come before nodes greater than or equal to x . The original relative order of the nodes in each of the two partitions should be preserved.

Logic Explanation

The goal is to rearrange the nodes in the linked list such that:

- All nodes with values less than x appear first.
- Then, all nodes with values greater than or equal to x appear after them.
- The relative order of nodes in each group must be preserved, meaning we can't sort or reverse them.

To achieve this, we use two dummy nodes to represent the heads of two separate lists:

1. One list for nodes less than x .
2. One list for nodes greater than or equal to x .

We traverse the original list once. For each node:

- If its value is less than x , we append it to the "before" list.
- Otherwise, we append it to the "after" list.

After the traversal, we connect the end of the "before" list to the start of the "after" list. We also ensure that the "after" list is properly terminated by setting its last node's next to None.

This solution preserves the relative order of nodes and completes in a single pass with constant extra space.

Time complexity is $O(n)$, and space complexity is $O(1)$ (excluding output list).

Solution:

```
Python3  Auto
1 class Solution:
2     def partition(self, head, x):
3         before_head = ListNode(0)
4         after_head = ListNode(0)
5
6         before = before_head
7         after = after_head
8
9         while head:
10            if head.val < x:
11                before.next = head
12                before = before.next
13            else:
14                after.next = head
15                after = after.next
16            head = head.next
17
18        after.next = None
19        before.next = after_head.next
20
21        return before_head.next
22
```

Practice here

<https://leetcode.com/problems/partition-list/description/>

Problem 44 - Linked List Cycle

Problem Statement

Given the head of a linked list, determine if the linked list has a cycle in it. A cycle occurs when a node's next pointer points to a previous node in the list, forming a loop. Return true if there is a cycle, otherwise return false.

Logic Explanation

To detect whether a cycle exists in a linked list, we can use the Floyd's Cycle Detection Algorithm, also known as the Tortoise and Hare approach. This method uses two pointers:

- The slow pointer moves one step at a time.
- The fast pointer moves two steps at a time.

If there is no cycle, the fast pointer will eventually reach the end of the list (i.e., None). However, if there is a cycle, the fast pointer will eventually catch up to the slow pointer inside the loop — because it's moving faster and will keep circling through the cycle.

This method is very efficient:

- It uses only constant space (no need to store visited nodes in a set).
- It runs in linear time because each pointer makes at most $O(n)$ steps.

This approach is optimal for detecting cycles in singly linked lists without modifying the structure or using additional data structures.

Solution:

```
Python3  Auto
1 class Solution:
2     def hasCycle(self, head):
3         slow = fast = head
4
5         while fast and fast.next:
6             slow = slow.next
7             fast = fast.next.next
8
9             if slow == fast:
10                return True
11
12        return False
13
```

Practice here

<https://leetcode.com/problems/linked-list-cycle/description/>

Problem 45 - Swap Nodes in Pairs

Problem Statement

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

Logic Explanation

To swap every two adjacent nodes in a linked list, we need to change the actual node connections instead of just swapping their values. For this, we can use a dummy node to simplify the edge cases, especially when the head of the list is involved in a swap.

We start by creating a dummy node that points to the head of the list. Then, we use a pointer called prev to track the node just before the pair that we are about to swap.

For each pair:

- We identify the first node (first) and the second node (second) in the pair.
- We perform the swap by:
 - Pointing prev.next to second
 - Setting first.next to second.next
 - Pointing second.next to first
- Then, move prev to first to prepare for the next pair.

This process continues until there are fewer than two nodes left to process. Since we traverse each node once and perform constant-time operations for each pair, the time complexity is $O(n)$. No extra space is used beyond the dummy node and a few pointers, so space complexity is $O(1)$.

Solution:

```
Python3  Auto
1  class Solution:
2      def swapPairs(self, head):
3          dummy = ListNode(0)
4          dummy.next = head
5          prev = dummy
6
7          while head and head.next:
8              first = head
9              second = head.next
10
11             # Swapping
12             prev.next = second
13             first.next = second.next
14             second.next = first
15
16             # Re-positioning for next swap
17             prev = first
18
```

Practice here

<https://leetcode.com/problems/swap-nodes-in-pairs/>

Problem 46 - Reverse Linked List

Problem Statement

Given the head of a singly linked list, reverse the list and return the reversed list.

Logic Explanation

To reverse a singly linked list, we need to reverse the direction of the next pointers for all nodes. The main challenge is that once we change a node's next pointer, we lose the reference to the rest of the list unless we save it beforehand.

We use three pointers:

- prev which initially is None and will eventually become the new head.
- current which starts at the head and moves through the list.
- next_node to temporarily hold the next node so we don't lose access to the rest of the list during the reversal.

In each iteration:

1. Store current.next in next_node.
2. Change current.next to point to prev, reversing the link.
3. Move prev and current one step forward.

After the loop ends, prev points to the new head of the reversed list.

This approach performs an in-place reversal in $O(n)$ time and $O(1)$ space.

Solution:

```
Python3  Auto
1  class Solution:
2      def reverseList(self, head):
3          prev = None
4          current = head
5
6          while current:
7              next_node = current.next
8              current.next = prev
9              prev = current
10             current = next_node
11
12         return prev
13
```

Practice here

<https://leetcode.com/problems/reverse-linked-list/description/>

Problem 47 - Merge Two Sorted Lists

Problem Statement

You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted linked list and return the head of the new list.

Logic Explanation

The goal is to merge two sorted linked lists into a single sorted linked list. Since the input lists are already sorted, we can take advantage of this by comparing the nodes one by one and building the result list in sorted order.

We use a dummy node to simplify the process of building the result list. A pointer called current starts at the dummy and keeps track of the last node in the new list.

We iterate over both lists:

- At each step, we compare the current nodes of both lists.
- We attach the smaller node to current.next and advance the pointer in that list.
- Move the current pointer forward.

Once one of the lists is fully traversed, we attach the remaining part of the other list to the result. Since the lists are sorted, the remaining part is already in order.

Finally, we return dummy.next, which points to the head of the merged list.

This approach runs in $O(n + m)$ time, where n and m are the lengths of the two lists, and uses $O(1)$ extra space.

Solution:

```
Python3  Auto
1 class Solution:
2     def mergeTwoLists(self, list1, list2):
3         dummy = ListNode(0)
4         current = dummy
5
6         while list1 and list2:
7             if list1.val < list2.val:
8                 current.next = list1
9                 list1 = list1.next
10            else:
11                current.next = list2
12                list2 = list2.next
13            current = current.next
14
15        # Attach remaining nodes, if any
16        current.next = list1 if list1 else list2
17
18        return dummy.next
19
```


Practice here

<https://leetcode.com/problems/merge-two-sorted-lists/description/>

Problem 48 - Linked List Cycle

Problem Statement

Given the head of a linked list, determine if the linked list contains a cycle. A cycle exists if some node in the list can be reached again by continuously following the next pointer. Return true if there is a cycle, otherwise return false.

Logic Explanation

To detect a cycle in a linked list, we can use Floyd's Cycle Detection Algorithm, also known as the Tortoise and Hare approach. This is an efficient and space-optimized technique.

The idea is to use two pointers:

- slow moves one node at a time.
- fast moves two nodes at a time.

If there's no cycle, the fast pointer will reach the end of the list (i.e., become None). However, if there is a cycle, the fast pointer will eventually "lap" and meet the slow pointer within the cycle. This is because fast is moving faster and re-enters the cycle repeatedly, while slow is still inside it.

This method ensures we detect cycles in linear time and without any extra space — making it more efficient than using a hash set to track visited nodes.

Time complexity is $O(n)$, and space complexity is $O(1)$.

Solution:

```
Python3  Auto
1 class Solution:
2     def hasCycle(self, head):
3         slow = fast = head
4
5         while fast and fast.next:
6             slow = slow.next
7             fast = fast.next.next
8
9         if slow == fast:
10            return True
11
12        return False
13
```

Practice here

<https://leetcode.com/problems/linked-list-cycle/description/>

BINARY SEARCH TREE

Problem 49 - Validate Binary Search Tree

Problem Statement

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

Logic Explanation

To validate whether a binary tree is a binary search tree, we must ensure that for every node:

- All nodes in its left subtree have values less than the current node.
- All nodes in its right subtree have values greater than the current node.
- This condition must hold true for the entire tree, not just direct children.

A simple mistake would be to check only `node.left.val < node.val < node.right.val`, but that doesn't work for deeper levels. So instead, we use a recursive approach with value boundaries.

We perform a depth-first traversal, where each node must lie within a valid range (`min_val`, `max_val`). Initially, this range is `(-infinity, +infinity)`:

- For the left child, the max allowed value becomes the current node's value.
- For the right child, the min allowed value becomes the current node's value.

If at any point a node's value is not within the allowed range, we return `False`.

This method ensures that the entire structure satisfies the BST rules. It traverses each node only once, resulting in $O(n)$ time complexity and $O(h)$ space complexity, where h is the height of the tree (due to recursion stack).

Solution:

```
Python3  Auto
1 class Solution:
2     def isValidBST(self, root):
3         def validate(node, low, high):
4             if not node:
5                 return True
6             if not (low < node.val < high):
7                 return False
8             return (validate(node.left, low, node.val) and
9                     validate(node.right, node.val, high))
10
11         return validate(root, float('-inf'), float('inf'))
12
```

Practice here

<https://leetcode.com/problems/validate-binary-search-tree/description/>

Problem 50 - Lowest Common Ancestor of a Binary Search Tree

Problem Statement

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes p and q in the BST. The lowest common ancestor is defined as the lowest node in the tree that has both p and q as descendants (where a node can be a descendant of itself).

Logic Explanation

In a Binary Search Tree (BST), all nodes in the left subtree of a node have values less than the node's value, and all nodes in the right subtree have values greater than the node's value. This property helps us find the LCA efficiently without checking the entire tree.

We start from the root and compare both p and q with the current node:

- If both p and q are less than the current node, then the LCA must be in the left subtree, so we move left.
- If both p and q are greater than the current node, then the LCA must be in the right subtree, so we move right.
- If p and q lie on either side of the current node (one is smaller and one is larger), or if one of them matches the current node, then the current node is the lowest common ancestor.

Since we follow the structure of the tree and ignore irrelevant parts, this solution is very efficient. It takes $O(h)$ time where h is the height of the tree. Space complexity is $O(1)$ if implemented iteratively, or $O(h)$ if done recursively.

Solution:

```
Python3  Auto
1 class Solution:
2     def lowestCommonAncestor(self, root, p, q):
3         while root:
4             if p.val < root.val and q.val < root.val:
5                 root = root.left
6             elif p.val > root.val and q.val > root.val:
7                 root = root.right
8             else:
9                 return root
10
```

Practice here

<https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/>

Problem 51 - Search in a Binary Search Tree

Problem Statement

You are given the root of a binary search tree (BST) and an integer `val`. Find the node in the BST that has the value equal to `val`, and return the subtree rooted at that node. If the node does not exist, return `None`.

Logic Explanation

This problem takes advantage of the Binary Search Tree property:

- For any node, values in the left subtree are smaller.
- Values in the right subtree are larger.

We start from the root and compare the value at the current node with the target `val`:

- If `val` is equal to the current node's value, we return the current node.
- If `val` is smaller, we search in the left subtree.
- If `val` is greater, we search in the right subtree.

We can implement this logic either recursively or iteratively. Both versions will only traverse one path from the root to a leaf, not the entire tree. Hence, the time complexity is $O(h)$, where h is the height of the tree. The space complexity is $O(h)$ for the recursive version due to the call stack, and $O(1)$ for the iterative version.

Solution:

```
Python3  ▾  🔒  Auto

1  class Solution:
2      def searchBST(self, root, val):
3          while root:
4              if root.val == val:
5                  return root
6              elif val < root.val:
7                  root = root.left
8              else:
9                  root = root.right
10         return None
11
```

Practice here

<https://leetcode.com/problems/search-in-a-binary-search-tree/>

Problem 52 - Insert into a Binary Search Tree

Problem Statement

You are given the root of a binary search tree (BST) and a value to insert into the tree. Insert the value into the BST such that the tree remains a valid BST after the insertion. Return the root of the BST after the insertion. It is guaranteed that the value does not exist in the original BST.

Logic Explanation

To insert a value into a BST, we leverage its core property:

- All nodes in the left subtree are less than the current node.
- All nodes in the right subtree are greater than the current node.

We start from the root and compare the given value with the current node:

- If the value is less than the current node's value, we go left.
- If it's greater, we go right.
- We continue this until we find a None position where we can insert the new node.

This can be implemented either recursively or iteratively. In the recursive version, each call checks whether the current subtree is the right place to insert. When we hit a None, we create a new node and return it. On the way back up the call stack, we reconstruct the subtree by reattaching the newly inserted node.

Since each decision halves the search space, the time complexity is $O(h)$, where h is the height of the tree. The space complexity is $O(h)$ due to recursion stack in the recursive approach.

Solution:

```
Python3  Auto
1 class Solution:
2     def insertIntoBST(self, root, val):
3         if not root:
4             return TreeNode(val)
5
6         if val < root.val:
7             root.left = self.insertIntoBST(root.left, val)
8         else:
9             root.right = self.insertIntoBST(root.right, val)
10
11         return root
12
```

Practice here

<https://leetcode.com/problems/insert-into-a-binary-search-tree/>

Problem 54 - Delete Node in a BST

Problem Statement

You are given the root of a binary search tree (BST) and a key. Delete the node with the given key in the BST, and return the root of the modified tree. The deletion must maintain the BST property.

Logic Explanation

Deleting a node from a BST is slightly more complex than insertion because we must handle three different cases depending on the number of children the node to be deleted has.

We search the tree to locate the node with the given key:

1. If the key is smaller than the current node's value, we move left.
2. If it's larger, we move right.
3. If it matches, we've found the node to delete.

Once we find the node, we consider the following deletion cases:

- **Node has no children (leaf):** Just return None to delete the node.
- **Node has one child:** Return the non-null child to replace the node.
- **Node has two children:**
 - Find the in-order successor, which is the smallest node in the right subtree.
 - Replace the value of the current node with the in-order successor's value.
 - Recursively delete the successor node from the right subtree.

This approach ensures the tree remains a valid BST after deletion.

The time complexity is $O(h)$, where h is the height of the tree, because in the worst case we traverse from root to leaf. The space complexity is $O(h)$ due to recursion stack.

Solution:

```
Python3  Auto
1 class Solution:
2     def deleteNode(self, root, key):
3         if not root:
4             return None
5
6         if key < root.val:
7             root.left = self.deleteNode(root.left, key)
8         elif key > root.val:
9             root.right = self.deleteNode(root.right, key)
10        else:
11            # Node to be deleted found
12            if not root.left:
13                return root.right
14            elif not root.right:
15                return root.left
16
17            # Node with two children
18            min_larger_node = self.findMin(root.right)
19            root.val = min_larger_node.val
20            root.right = self.deleteNode(root.right, min_larger_node.val)
21
22        return root
23
24    def findMin(self, node):
25        while node.left:
26            node = node.left
27        return node
28
```

Practice here

<https://leetcode.com/problems/delete-node-in-a-binary-search-tree/description/>