

Array and Hashing

Problem: LeetCode 217 - Contains Duplicate

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

Key Idea:

To check if there are any duplicates in the given list, we can use a hash set (set in Python) to store the unique elements as we traverse the list. For each element, we check if it is already present in the set. If it is, then we have found a duplicate, and we return `True`. If we traverse the entire list without finding any duplicates, we return `False`.

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        hashset = set()
        for n in nums:
            if n in hashset:
                return True
            hashset.add(n)
        return False
```

Problem: LeetCode 242 - Valid Anagram

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, typically using all the original letters exactly once.

Example 1:

Input: `s = "anagram", t = "nagaram"`

Output: `true`

Example 2:

Input: `s = "rat", t = "car"`

Output: `false`

Key Idea:

To determine if two given strings are anagrams of each other, we can compare their character frequencies. An anagram of a string contains the same characters with the same frequency, just arranged differently. We can use a hash map (dictionary in Python) to keep track of the character frequencies for each string. If the character frequencies of both strings match, then they are anagrams.

```

class Solution:

    def isAnagram(self, s: str, t: str) -> bool:

        if len(s) != len(t):

            return False

        char_frequency = {}

        # Build character frequency map for string s
        for char in s:

            char_frequency[char] = char_frequency.get(char, 0) + 1

        # Compare with string t
        for char in t:

            if char not in char_frequency or char_frequency[char] == 0:

                return False

            char_frequency[char] -= 1

        return True

```

Problem: LeetCode 1 - Two Sum

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

You may assume that each input would have ***exactly one solution***, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

Input: `nums = [2,7,11,15]`, `target = 9`

Output: `[0,1]`

Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`

Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`

Output: `[0,1]`

Key Idea:

The key idea to solve this problem efficiently is by using a hash map (dictionary in Python) to keep track of the elements we have traversed so far. For each element in the input list, we calculate the difference between the target and the current element. If this difference exists in the hash map, then we have found the pair that sums to the target, and we return their indices. Otherwise, we add the current element to the hash map and continue with the next element.

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        prevMap = {} # val -> index

        for i, n in enumerate(nums):
            diff = target - n
            if diff in prevMap:
                return [prevMap[diff], i]
            prevMap[n] = i
```

Problem: LeetCode 49 – Group Anagrams

Given an array of strings `strs`, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: `strs = ["eat", "tea", "tan", "ate", "nat", "bat"]`

Output: `[["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]`

Example 2:

Input: `strs = [""]`

Output: `[[""]]`

Example 3:

Input: `strs = ["a"]`

Output: `[["a"]]`

Key Idea:

To group anagrams together, we can use a hash map (dictionary in Python) where the key is a sorted version of each word, and the value is a list of words that are anagrams of each other. By iterating through the list of words, we can group them into the hash map based on their sorted versions.

```
class Solution:

    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:

        anagrams_map = {}

        for word in strs:

            sorted_word = "".join(sorted(word))

            if sorted_word in anagrams_map:

                anagrams_map[sorted_word].append(word)

            else:

                anagrams_map[sorted_word] = [word]

        return list(anagrams_map.values())
```

Problem: LeetCode 347 - Top K Frequent Elements

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

Example 1:

Input: *nums* = [1,1,1,2,2,3], *k* = 2

Output: [1,2]

Example 2:

Input: *nums* = [1], *k* = 1

Output: [1]

Key Idea:

To find the *k* most frequent elements in the given list, we can use a hash map (dictionary in Python) to keep track of the frequency of each element. We then use a min-heap (priority queue) to keep the *k* most frequent elements at the top. We traverse the list once to build the frequency map, and then we traverse the map to keep the *k* most frequent elements in the min-heap.

```

import heapq

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        frequency_map = {}
        for num in nums:
            frequency_map[num] = frequency_map.get(num, 0) + 1

        min_heap = []
        for num, frequency in frequency_map.items():
            heapq.heappush(min_heap, (frequency, num))
            if len(min_heap) > k:
                heapq.heappop(min_heap)

        return [num for frequency, num in min_heap]

```

Problem: LeetCode 238 - Product of Array Except Self

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]`*.

The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in $O(n)$ time and without using the division operation.

Example 1:

Input: `nums = [1,2,3,4]`

Output: `[24,12,8,6]`

Example 2:

Input: `nums = [-1,1,0,-3,3]`

Output: `[0,0,9,0,0]`

Key Idea:

To solve this problem, we can first calculate the product of all elements to the left of each index and store it in a list. Then, we calculate the product of all elements to the right of each index and update the result list accordingly by multiplying it with the previously calculated left product. In this way, each element in the result list will contain the product of all elements except the one at that index.

```

class Solution:

    def productExceptSelf(self, nums: List[int]) -> List[int]:

        n = len(nums)

        result = [1] * n

        # Calculate the left product of each element
        left_product = 1
        for i in range(n):
            result[i] *= left_product
            left_product *= nums[i]

        # Calculate the right product of each element and update the result list
        right_product = 1
        for i in range(n - 1, -1, -1):
            result[i] *= right_product
            right_product *= nums[i]

        return result

```

Problem: LeetCode 36 - Valid Sudoku

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits 1-9 without repetition.
2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

Key Idea:

To determine if a given Sudoku board is valid, we need to check three conditions:

1. Each row must have distinct digits from 1 to 9.
2. Each column must have distinct digits from 1 to 9.
3. Each 3x3 sub-grid must have distinct digits from 1 to 9.

We can use three nested loops to traverse the entire board and use sets to keep track of digits seen in each row, column, and sub-grid.

```
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        seen = set()

        for i in range(9):
            for j in range(9):
                if board[i][j] != ".":
                    num = board[i][j]
                    if (
                        (i, num) in seen
                        or (num, j) in seen
                        or (i // 3, j // 3, num) in seen
                    ):
                        return False
                    seen.add((i, num))
                    seen.add((num, j))
                    seen.add((i // 3, j // 3, num))

        return True
```

Problem: LeetCode 128 -Longest Consecutive Sequence

Given an unsorted array of integers nums, return *the length of the longest consecutive elements sequence*.

You must write an algorithm that runs in $O(n)$ time.

Example 1:

Input: nums = [100,4,200,1,3,2]

Output: 4

Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

Example 2:

Input: nums = [0,3,7,2,5,8,4,6,0,1]

Output: 9

Key Idea:

To find the longest consecutive subsequence, we first create a set of all the elements in the input array 'nums'. Then, for each element in the array, we check if it is the starting element of a consecutive subsequence. To do this, we check if the element before the current element (i.e., $\text{nums}[i] - 1$) exists in the set. If it doesn't, it means $\text{nums}[i]$ is the starting element of a consecutive subsequence. From here, we keep incrementing the current element until the consecutive subsequence ends (i.e., the next element does not exist in the set). We keep track of the length of the consecutive subsequence and update the maximum length found so far.

```
class Solution:

    def longestConsecutive(self, nums: List[int]) -> int:

        num_set = set(nums)

        max_length = 0

        for num in num_set:

            if num - 1 not in num_set:

                current_num = num

                current_length = 1

                while current_num + 1 in num_set:

                    current_num += 1

                    current_length += 1

                max_length = max(max_length, current_length)

        return max_length
```

Two Pointer

Problem: LeetCode 125 - Valid Palindrome

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string *s*, return true *if it is a palindrome*, or false *otherwise*.

Example 1:

Input: *s* = "A man, a plan, a canal: Panama"

Output: true

Explanation: "amanaplanacanalpanama" is a palindrome.

Example 2:

Input: *s* = "race a car"

Output: false

Explanation: "raceacar" is not a palindrome.

Key Idea:

To determine if a given string is a valid palindrome, we can use two pointers approach. We initialize two pointers, one at the beginning of the string (left) and the other at the end of the string (right). We then compare characters at these two pointers. If they are both alphanumeric characters and equal in value (ignoring case), we move both pointers towards the center of the string. If they are not equal, we know the string is not a palindrome. We continue this process until the two pointers meet or cross each other.

```
class Solution:
    def isPalindrome(self, s: str) -> bool:
        left, right = 0, len(s) - 1
        while left < right:
            while left < right and not s[left].isalnum():
                left += 1
            while left < right and not s[right].isalnum():
                right -= 1
            if s[left].lower() != s[right].lower():
                return False
            left += 1
            right -= 1
        return True
```

Problem: LeetCode 167 - Two Sum II

Given a **1-indexed** array of integers numbers that is already *sorted in non-decreasing order*, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index₁] and numbers[index₂] where $1 \leq \text{index}_1 < \text{index}_2 \leq \text{numbers.length}$.

Return *the indices of the two numbers*, index₁ and index₂, **added by one** as an integer array [index₁, index₂] of length 2.

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

Example 1:

Input: numbers = [2,7,11,15], target = 9

Output: [1,2]

Explanation: The sum of 2 and 7 is 9. Therefore, index₁ = 1, index₂ = 2. We return [1, 2].

Example 2:

Input: numbers = [2,3,4], target = 6

Output: [1,3]

Explanation: The sum of 2 and 4 is 6. Therefore index₁ = 1, index₂ = 3. We return [1, 3].

Key Idea:

The input array 'numbers' is already sorted in non-decreasing order. To find the two numbers that add up to the target, we can use a two-pointer approach. We initialize two pointers, one at the beginning of the array (left) and the other at the end of the array (right). We then check the sum of the elements at these two pointers. If the sum is equal to the target, we have found the pair. If the sum is less than the target, it means we need to increase the sum, so we move the left pointer one step to the right. If the sum is greater than the target, it means we need to decrease the sum, so we move the right pointer one step to the left. We continue this process until we find the pair or the two pointers meet or cross each other.

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        left, right = 0, len(numbers) - 1
        while left < right:
            current_sum = numbers[left] + numbers[right]
            if current_sum == target:
                return [left + 1, right + 1]
            elif current_sum < target:
                left += 1
            else:
                right -= 1
        # No solution found
        return [-1, -1]
```

Problem: LeetCode 15 - 3Sum

Given an integer array `nums`, return all the triplets `[nums[i], nums[j], nums[k]]` such that $i \neq j$, $i \neq k$, and $j \neq k$, and $nums[i] + nums[j] + nums[k] = 0$.

Notice that the solution set must not contain duplicate triplets.

Example 1:

Input: `nums = [-1,0,1,2,-1,-4]`

Output: `[[-1,-1,2],[-1,0,1]]`

Explanation:

$nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0$.

$nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0$.

$nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0$.

The distinct triplets are `[-1,0,1]` and `[-1,-1,2]`.

Notice that the order of the output and the order of the triplets does not matter.

Example 2:

Input: `nums = [0,1,1]`

Output: `[]`

Explanation: The only possible triplet does not sum up to 0.

Example 3:

Input: `nums = [0,0,0]`

Output: `[[0,0,0]]`

Explanation: The only possible triplet sums up to 0.

Key Idea:

To find all unique triplets that sum to zero, we can use a three-pointer approach. First, we sort the input array 'nums' in non-decreasing order. Then, we iterate through the array with a fixed first element (i). For each fixed first element, we use two pointers (left and right) to find the other two elements that sum to the negation of the fixed first element. As the array is sorted, we can move these two pointers towards each other to efficiently find all possible triplets.

```
class Solution:
```

```
    def threeSum(self, nums: List[int]) -> List[List[int]]:
```

```
        nums.sort()
```

```
        result = []
```

```
        n = len(nums)
```

```
        for i in range(n - 2):
```

```
            if i > 0 and nums[i] == nums[i - 1]:
```

```
                continue
```

```
            left, right = i + 1, n - 1
```

```
            while left < right:
```

```
                current_sum = nums[i] + nums[left] + nums[right]
```

```
                if current_sum == 0:
```

```
                    result.append([nums[i], nums[left], nums[right]])
```

```
                    # Skip duplicates
```

```
                    while left < right and nums[left] == nums[left + 1]:
```

```
                        left += 1
```

```
                    while left < right and nums[right] == nums[right - 1]:
```

```
                        right -= 1
```

```
                    left += 1
```

```
                    right -= 1
```

```
                elif current_sum < 0:
```

```
                    left += 1
```

```
                else:
```

```
                    right -= 1
```

```
        return result
```

Problem: LeetCode 11 - Container With Most Water

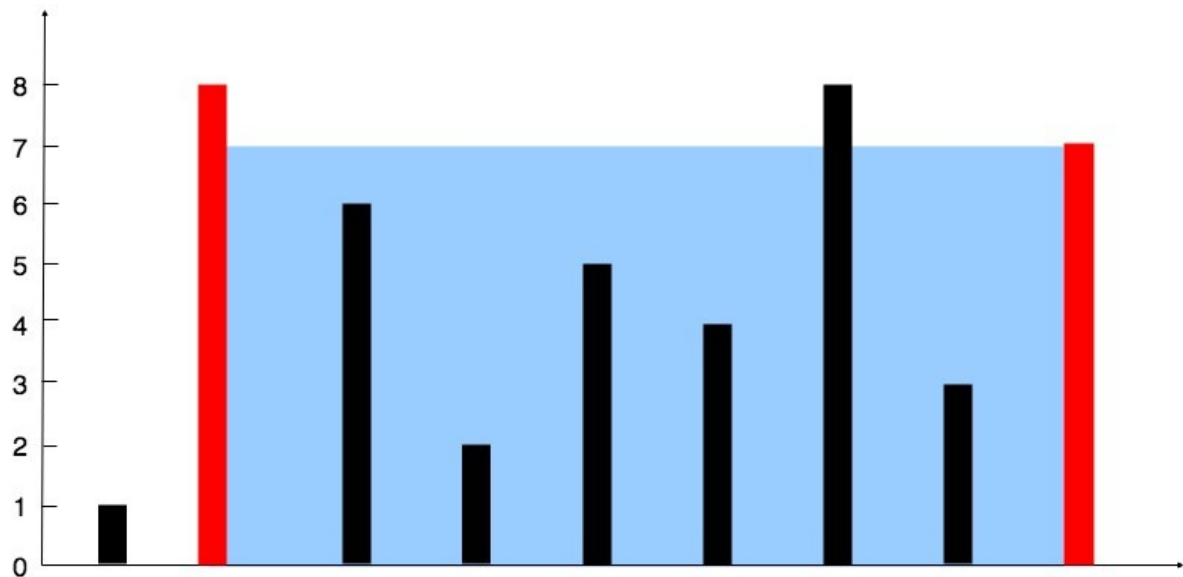
You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the i^{th} line are $(i, 0)$ and $(i, \text{height}[i])$.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.

Example 1:



Input: `height = [1,8,6,2,5,4,8,3,7]`

Output: 49

Explanation: The above vertical lines are represented by array `[1,8,6,2,5,4,8,3,7]`. In this case, the max area of water (blue section) the container can contain is 49.

Example 2:

Input: `height = [1,1]`

Output: 1

Key Idea:

To find the maximum area of water that can be held between two vertical lines, we can use a two-pointer approach. We initialize two pointers, one at the beginning of the input array (left) and the other at the end of the array (right). The area between the two vertical lines is calculated as the minimum of the heights at the two pointers multiplied by the distance between them. We then update the maximum area found so far and move the pointer with the smaller height towards the other pointer. We continue this process until the two pointers meet or cross each other.

```

class Solution:

    def maxArea(self, height: List[int]) -> int:

        left, right = 0, len(height) - 1

        max_area = 0

        while left < right:

            current_area = min(height[left], height[right]) * (right - left)

            max_area = max(max_area, current_area)

            if height[left] < height[right]:

                left += 1

            else:

                right -= 1

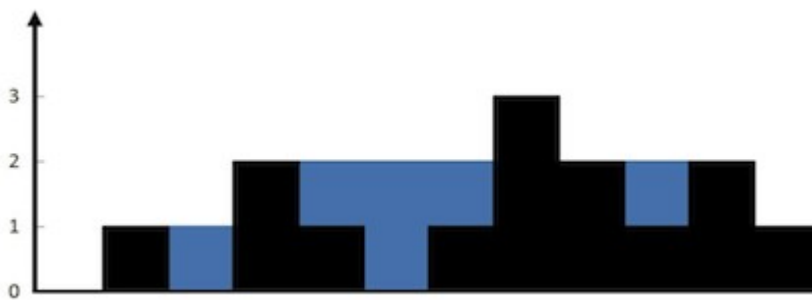
        return max_area

```

Problem: LeetCode 42 - Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example 1:



Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]

Output: 6

Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Example 2:

Input: height = [4,2,0,3,2,5]

Output: 9

Key Idea:

To find the amount of trapped rainwater in the given elevation histogram represented by the input array 'height', we can use a two-pointer approach. We initialize two pointers, one at the beginning of the array (left) and the other at the end of the array (right). We also initialize two variables to keep track of the maximum left height and maximum right height seen so far. While the left pointer is less than the right pointer, we compare the height at the left and right pointers. If the height at the left pointer is less than or equal to the height at the right pointer, it means we can trap water between the left pointer and the maximum left height. Otherwise, we can trap water between the right pointer and the maximum right height. At each step, we update the trapped water amount and move the pointers and update the maximum heights accordingly.

```
class Solution:
```

```
    def trap(self, height: List[int]) -> int:
```

```
        left, right = 0, len(height) - 1
```

```
        max_left, max_right = 0, 0
```

```
        trapped_water = 0
```

```
        while left < right:
```

```
            if height[left] <= height[right]:
```

```
                if height[left] >= max_left:
```

```
                    max_left = height[left]
```

```
            else:
```

```
                trapped_water += max_left - height[left]
```

```
            left += 1
```

```
        else:
```

```
            if height[right] >= max_right:
```

```
                max_right = height[right]
```

```
            else:
```

```
                trapped_water += max_right - height[right]
```

```
            right -= 1
```

```
        return trapped_water
```

Problem: LeetCode 121 - Best Time to Buy and Sell Stock

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:

Input: `prices = [7,1,5,3,6,4]`

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = $6 - 1 = 5$.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: `prices = [7,6,4,3,1]`

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Key Idea:

To find the maximum profit from a single buy and sell transaction in the input array 'prices', we can use a simple one-pass approach. We initialize two variables, 'min_price' to keep track of the minimum price seen so far, and 'max_profit' to store the maximum profit. We iterate through the 'prices' array, and for each price, we update the 'min_price' if we find a smaller price. We also calculate the potential profit if we sell at the current price and update 'max_profit' if the current profit is greater than the previous maximum profit.

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0

        min_price = float("inf")
        max_profit = 0

        for price in prices:
            min_price = min(min_price, price)
            max_profit = max(max_profit, price - min_price)

        return max_profit
```

Problem: LeetCode 3 - Longest Substring Without Repeating Characters

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

Example 1:

Input: *s* = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

Example 2:

Input: *s* = "bbbbb"

Output: 1

Explanation: The answer is "b", with the length of 1.

Example 3:

Input: *s* = "pwwkew"

Output: 3

Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Key Idea:

To find the length of the longest substring without repeating characters in the input string '*s*', we can use the sliding window approach. We use two pointers, 'left' and 'right', to represent the current window. As we move the 'right' pointer to the right, we expand the window and add characters to a set to keep track of unique characters in the window. If we encounter a repeating character, we move the 'left' pointer to the right to shrink the window until the repeating character is no longer in the window. At each step, we update the maximum length of the window (i.e., the length of the longest substring without repeating characters).

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        left, right = 0, 0
        max_length = 0
        unique_chars = set()

        while right < len(s):
            if s[right] not in unique_chars:
                unique_chars.add(s[right])
                max_length = max(max_length, right - left + 1)
                right += 1
```

```
else:
    unique_chars.remove(s[left])
    left += 1

return max_length
```

Problem: LeetCode 424 - Longest Repeating Character Replacement

You are given a string *s* and an integer *k*. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most *k* times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations*.

Example 1:

Input: *s* = "ABAB", *k* = 2

Output: 4

Explanation: Replace the two 'A's with two 'B's or vice versa.

Example 2:

Input: *s* = "AABABBA", *k* = 1

Output: 4

Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBBA".

The substring "BBBB" has the longest repeating letters, which is 4.

There may exist other ways to achieve this answer too.

Key Idea:

To find the maximum length of a substring with at most *k* distinct characters in the input string '*s*', we can use the sliding window approach. We use two pointers, 'left' and 'right', to represent the current window. As we move the 'right' pointer to the right, we expand the window and add characters to a dictionary to keep track of their frequencies. If the number of distinct characters in the window exceeds *k*, we move the 'left' pointer to the right to shrink the window until the number of distinct characters is *k* again. At each step, we update the maximum length of the window.

```

class Solution:

    def characterReplacement(self, s: str, k: int) -> int:

        left, right = 0, 0

        max_length = 0

        char_freq = {}

        max_freq = 0

        while right < len(s):

            char_freq[s[right]] = char_freq.get(s[right], 0) + 1

            max_freq = max(max_freq, char_freq[s[right]])

            if (right - left + 1) - max_freq > k:

                char_freq[s[left]] -= 1

                left += 1

            max_length = max(max_length, right - left + 1)

            right += 1

        return max_length

```

Problem: LeetCode 567 - Permutation in String

Given two strings *s1* and *s2*, return true *if s2 contains a permutation of s1*, or false otherwise.

In other words, return true if one of *s1*'s permutations is the substring of *s2*.

Example 1:

Input: *s1* = "ab", *s2* = "eidbaooo"

Output: true

Explanation: *s2* contains one permutation of *s1* ("ba").

Example 2:

Input: *s1* = "ab", *s2* = "eidboao"

Output: false

Key Idea:

To check whether a string '*s2*' contains a permutation of another string '*s1*', we can use a sliding window approach. First, we create a frequency dictionary for characters in '*s1*'. Then, we initialize two pointers, '*left*' and '*right*', to represent the current window in '*s2*'. As we move the '*right*' pointer to the right, we add the character to a temporary frequency dictionary and check if it becomes equal to the frequency dictionary of '*s1*'. If it does, it means we found a permutation of '*s1*' in '*s2*', and we return True. If the window size exceeds the length of '*s1*', we remove the character at the '*left*' pointer from the temporary dictionary and move the '*left*' pointer to the right to shrink the window. We continue this process until we find a permutation or reach the end of '*s2*'.

```
class Solution:

    def checkInclusion(self, s1: str, s2: str) -> bool:

        if len(s1) > len(s2):

            return False

        char_freq_s1 = {}

        for char in s1:

            char_freq_s1[char] = char_freq_s1.get(char, 0) + 1

        left, right = 0, 0

        char_freq_temp = {}

        while right < len(s2):

            char_freq_temp[s2[right]] = char_freq_temp.get(s2[right], 0) + 1

            if right - left + 1 == len(s1):

                if char_freq_temp == char_freq_s1:

                    return True

                char_freq_temp[s2[left]] -= 1

                if char_freq_temp[s2[left]] == 0:

                    del char_freq_temp[s2[left]]

                left += 1

            right += 1

        return False
```

STACK

Problem: LeetCode 20 - Valid Parentheses

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

An input string is valid if:

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

Example 1:

Input: *s* = "()"

Output: true

Example 2:

Input: *s* = "()[]{}"

Output: true

Example 3:

Input: *s* = "["

Output: false

Key Idea:

To determine if a given string of parentheses '*s*' is valid, we can use a stack data structure. We iterate through each character in '*s*', and if the character is an opening parenthesis ('(', '{', '['), we push it onto the stack. If the character is a closing parenthesis (')', '}', ']'), we check if the stack is empty or if the top element of the stack does not match the current closing parenthesis. If either of these conditions is met, we know the string is not valid. Otherwise, we pop the top element from the stack. At the end, if the stack is empty, the string is valid.

```
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        parentheses_map = {"(": ")", "{": "}", "[": "]"}

        for char in s:
            if char in parentheses_map.values():
                stack.append(char)
```



```

elif char in parentheses_map:
    if not stack or stack[-1] != parentheses_map[char]:
        return False
    stack.pop()
else:
    return False

return not stack

```

Problem: LeetCode 155 - Min Stack

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

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

Example 1:

Input

```

["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[ ]]

```

Output

```

[null,null,null,null,-3,null,0,-2]

```

Explanation

```

MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();

```

```
minStack.top(); // return 0  
minStack.getMin(); // return -2
```

Key Idea:

To implement a stack that supports finding the minimum element in constant time, we can use two stacks: one for storing the actual elements (stack) and another for keeping track of the minimum elements (min_stack). The min_stack will always have the current minimum element at the top. When pushing an element onto the stack, we compare it with the top element of the min_stack and push the smaller element onto the min_stack. When popping an element from the stack, we check if the element being popped is the same as the top element of the min_stack. If it is, we also pop the element from the min_stack. This way, the top element of the min_stack will always be the minimum element in the stack.

```
class MinStack:  
    def __init__(self):  
        self.stack = []  
        self.min_stack = []  
  
    def push(self, val: int) -> None:  
        self.stack.append(val)  
        if not self.min_stack or val <= self.min_stack[-1]:  
            self.min_stack.append(val)  
  
    def pop(self) -> None:  
        if self.stack:  
            if self.stack[-1] == self.min_stack[-1]:  
                self.min_stack.pop()  
            self.stack.pop()  
  
    def top(self) -> int:  
        if self.stack:  
            return self.stack[-1]  
  
    def getMin(self) -> int:  
        if self.min_stack:  
            return self.min_stack[-1]
```

Problem: LeetCode 150 - Evaluate Reverse Polish Notation

You are given an array of strings tokens that represents an arithmetic expression in a [Reverse Polish Notation](#).

Evaluate the expression. Return *an integer that represents the value of the expression*.

Note that:

- The valid operators are '+', '-', '*', and '/'.
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a **32-bit** integer.

Example 1:

Input: tokens = ["2","1","+","3","*"]

Output: 9

Explanation: $((2 + 1) * 3) = 9$

Example 2:

Input: tokens = ["4","13","5","/","+"]

Output: 6

Explanation: $(4 + (13 / 5)) = 6$

Example 3:

Input: tokens = ["10","6","9","3","+","-11","*","/", "*", "17","+","5","+"]

Output: 22

Explanation: $((10 * (6 / ((9 + 3) * -11))) + 17) + 5$

$= ((10 * (6 / (12 * -11))) + 17) + 5$

$= ((10 * (6 / -132)) + 17) + 5$

$= ((10 * 0) + 17) + 5$

$= (0 + 17) + 5$

$= 17 + 5$

$= 22$

Key Idea:

To evaluate a given reverse Polish notation expression, we can use a stack data structure. We iterate through the tokens in the expression, and for each token, if it is a number, we push it onto the stack. If it is an operator ('+',

'-', '*', '/'), we pop the top two elements from the stack, apply the operator to them, and push the result back onto the stack. At the end, the top element of the stack will be the final result of the expression.

```
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = []

        for token in tokens:
            if token.isdigit() or (token[0] == "-" and token[1:].isdigit()):
                stack.append(int(token))
            else:
                num2 = stack.pop()
                num1 = stack.pop()
                if token == "+":
                    stack.append(num1 + num2)
                elif token == "-":
                    stack.append(num1 - num2)
                elif token == "*":
                    stack.append(num1 * num2)
                elif token == "/":
                    stack.append(int(num1 / num2))

        return stack[0]
```

Problem: LeetCode 22 - Generate Parentheses

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

Example 1:

Input: $n = 3$

Output: ["((())", "(())", "(())", "()", "000"]

Example 2:

Input: $n = 1$

Output: ["()"]

Key Idea:

To generate all valid combinations of parentheses, we can use backtracking. We start with an empty string and two counters, one for the open parentheses and one for the close parentheses. At each step, we have two choices: add an open parenthesis if the count of open parentheses is less than the total number of pairs, or add a close parenthesis if the count of close parentheses is less than the count of open parentheses. We continue this process recursively until we reach the desired length of the string. If the string becomes valid, we add it to the result.

```
class Solution:

    def generateParenthesis(self, n: int) -> List[str]:

        def backtrack(s, open_count, close_count):

            if len(s) == 2 * n:

                result.append(s)

                return

            if open_count < n:

                backtrack(s + "(", open_count + 1, close_count)

            if close_count < open_count:

                backtrack(s + ")", open_count, close_count + 1)

        result = []

        backtrack("", 0, 0)

        return result
```

Problem: LeetCode 739 - Daily Temperatures

Given an array of integers temperatures represents the daily temperatures, return *an array answer such that answer[i] is the number of days you have to wait after the ith day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead.

Example 1:

Input: temperatures = [73,74,75,71,69,72,76,73]

Output: [1,1,4,2,1,1,0,0]

Example 2:

Input: temperatures = [30,40,50,60]

Output: [1,1,1,0]

Example 3:**Input:** temperatures = [30,60,90]**Output:** [1,1,0]**Key Idea:**

To find the daily temperatures that are warmer in the input array 'temperatures', we can use a stack. We iterate through the temperatures in reverse order, and for each temperature, we pop elements from the stack while they are smaller than the current temperature. This indicates that the current temperature is the first warmer temperature for the popped elements. We keep track of the indices of these warmer temperatures in the result array. Then, we push the current temperature's index onto the stack. At the end, the result array will contain the number of days until the next warmer temperature for each day.

```
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        stack = []
        result = [0] * len(temperatures)

        for i in range(len(temperatures) - 1, -1, -1):
            while stack and temperatures[i] >= temperatures[stack[-1]]:
                stack.pop()
            if stack:
                result[i] = stack[-1] - i
            stack.append(i)

        return result
```

Problem: LeetCode 853 - car Fleet

There are n cars at given miles away from the starting mile 0, traveling to reach the mile target.

You are given two integer array position and speed, both of length n, where position[i] is the starting mile of the ith car and speed[i] is the speed of the ith car in miles per hour.

A car cannot pass another car, but it can catch up and then travel next to it at the speed of the slower car.

A **car fleet** is a car or cars driving next to each other. The speed of the car fleet is the **minimum** speed of any car in the fleet.

If a car catches up to a car fleet at the mile target, it will still be considered as part of the car fleet.

Return the number of car fleets that will arrive at the destination.

Example 1:

Input: target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]

Output: 3

Explanation:

- The cars starting at 10 (speed 2) and 8 (speed 4) become a fleet, meeting each other at 12. The fleet forms at target.
- The car starting at 0 (speed 1) does not catch up to any other car, so it is a fleet by itself.
- The cars starting at 5 (speed 1) and 3 (speed 3) become a fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.

Example 2:

Input: target = 10, position = [3], speed = [3]

Output: 1

Explanation:

There is only one car, hence there is only one fleet.

Example 3:

Input: target = 100, position = [0,2,4], speed = [4,2,1]

Output: 1

Explanation:

- The cars starting at 0 (speed 4) and 2 (speed 2) become a fleet, meeting each other at 4. The car starting at 4 (speed 1) travels to 5.
- Then, the fleet at 4 (speed 2) and the car at position 5 (speed 1) become one fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.

Key Idea:

To determine the number of car fleets that can reach the target destination in the input arrays 'target' and 'position', we can simulate the car movements and calculate the time it takes for each car to reach the target. We can then sort the cars based on their positions and iterate through them. For each car, we calculate its time to reach the target and compare it with the previous car. If the time for the current car is greater, it means the previous car cannot catch up to it, so we consider the current car as a new fleet. Otherwise, the previous car can catch up to the current car, so they form a fleet together.

```
class Solution:
    def carFleet(self, target: int, position: List[int], speed: List[int]) -> int:
        cars = sorted(zip(position, speed), reverse=True)
        fleets = 0
        prev_time = -1.0
```

```
for pos, spd in cars:
    time = (target - pos) / spd
    if time > prev_time:
        fleets += 1
        prev_time = time

return fleets
```


Binary Search

Problem: LeetCode 704 - Binary Search

Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search `target` in `nums`. If `target` exists, then return its index. Otherwise, return `-1`.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

Input: `nums = [-1,0,3,5,9,12]`, `target = 9`

Output: `4`

Explanation: `9` exists in `nums` and its index is `4`

Key Idea:

Binary search is an efficient technique to search for a target element in a sorted array. In each step, we compare the middle element of the array with the target. If the middle element is equal to the target, we have found the element and return its index. If the middle element is greater than the target, we narrow down the search to the left half of the array. If the middle element is smaller, we narrow down the search to the right half of the array. We repeat this process until we find the target element or the search space is exhausted.

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                left = mid + 1
            else:
                right = mid - 1

        return -1
```

Problem: LeetCode 74 - Search a 2D Matrix

You are given an $m \times n$ integer matrix `matrix` with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer `target`, return `true` *if target is in matrix* or `false` *otherwise*.

You must write a solution in $O(\log(m * n))$ time complexity.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

Input: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 3`

Output: `true`

Example 2:

1	3	5	7
10	11	16	20
23	30	34	60

Input: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 13`

Output: `false`

Key Idea:

Since both the rows and columns in the input 2D matrix are sorted, we can treat the matrix as a one-dimensional sorted array and perform binary search to find the target element. We can map the 2D indices to the corresponding index in the 1D array and then apply binary search to locate the target element.

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        if not matrix or not matrix[0]:
            return False
        rows, cols = len(matrix), len(matrix[0])
        left, right = 0, rows * cols - 1
        while left <= right:
            mid = left + (right - left) // 2
```

```
num = matrix[mid // cols][mid % cols]

if num == target:
    return True
elif num < target:
    left = mid + 1
else:
    right = mid - 1

return False
```

Problem: LeetCode 875 - Koko Eating Bananas

Koko loves to eat bananas. There are n piles of bananas, the i^{th} pile has $\text{piles}[i]$ bananas. The guards have gone and will come back in h hours.

Koko can decide her bananas-per-hour eating speed of k . Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return *the minimum integer k such that she can eat all the bananas within h hours.*

Example 1:

Input: $\text{piles} = [3, 6, 7, 11]$, $h = 8$

Output: 4

Example 2:

Input: $\text{piles} = [30, 11, 23, 4, 20]$, $h = 5$

Output: 30

Example 3:

Input: $\text{piles} = [30, 11, 23, 4, 20]$, $h = 6$

Output: 23

Key Idea:

The key idea is to perform binary search to find the minimum value of the integer ' k ' such that Koko can eat all the bananas within ' h ' hours. We can define a binary search space for ' k ' and perform binary search to find the smallest ' k ' that satisfies the given condition.

```
class Solution:

    def minEatingSpeed(self, piles: List[int], h: int) -> int:

        left, right = 1, max(piles)

        while left < right:
```

```

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

hours = sum((pile + mid - 1) // mid for pile in piles)

if hours > h:
    left = mid + 1
else:
    right = mid
return left

```

Problem: LeetCode 153 - Find Minimum in Rotated Sorted Array

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

- `[4,5,6,7,0,1,2]` if it was rotated 4 times.
- `[0,1,2,4,5,6,7]` if it was rotated 7 times.

Notice that **rotating** an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in $O(\log n)$ time.

Example 1:

Input: `nums = [3,4,5,1,2]`

Output: 1

Explanation: The original array was `[1,2,3,4,5]` rotated 3 times.

Example 2:

Input: `nums = [4,5,6,7,0,1,2]`

Output: 0

Explanation: The original array was `[0,1,2,4,5,6,7]` and it was rotated 4 times.

Example 3:

Input: `nums = [11,13,15,17]`

Output: 11

Explanation: The original array was `[11,13,15,17]` and it was rotated 4 times.

Key Idea:

The key idea is to perform binary search to find the minimum element in the rotated sorted array. We compare the middle element with its neighbors to determine if it is the minimum element. Depending on the comparison, we narrow down the search to the unsorted part of the array.

```

class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1

        while left < right:
            mid = left + (right - left) // 2

            if nums[mid] > nums[right]:
                left = mid + 1
            else:
                right = mid

        return nums[left]

```

Problem: LeetCode 33 - Search in Rotated Sorted Array

There is an integer array `nums` sorted in ascending order (with **distinct** values).

Prior to being passed to your function, `nums` is **possibly rotated** at an unknown pivot index `k` ($1 \leq k < \text{nums.length}$) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index 3 and become `[4,5,6,7,0,1,2]`.

Given the array `nums` **after** the possible rotation and an integer `target`, return *the index of target if it is in nums, or -1 if it is not in nums*.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

Input: `nums = [4,5,6,7,0,1,2]`, `target = 0`

Output: 4

Example 2:

Input: `nums = [4,5,6,7,0,1,2]`, `target = 3`

Output: -1

Example 3:

Input: `nums = [1]`, `target = 0`

Output: -1

Key Idea:

The key idea is to perform binary search to find the target element in the rotated sorted array. We compare the middle element with the target and the endpoints of the subarray to determine which part of the array is sorted. Depending on the comparison, we narrow down the search to the sorted part of the array.

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                return mid

            if nums[left] <= nums[mid]:
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1

        return -1
```

Problem: LeetCode 981 - Time Based Key-Value Store

Design a time-based key-value data structure that can store multiple values for the same key at different time stamps and retrieve the key's value at a certain timestamp.

Implement the TimeMap class:

- TimeMap() Initializes the object of the data structure.
- void set(String key, String value, int timestamp) Stores the key key with the value value at the given time timestamp.

- String get(String key, int timestamp) Returns a value such that set was called previously, with timestamp_prev <= timestamp. If there are multiple such values, it returns the value associated with the largest timestamp_prev. If there are no values, it returns "".

Example 1:

Input

```
["TimeMap", "set", "get", "get", "set", "get", "get"]
```

```
[[["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["foo", 5]]
```

Output

```
[null, null, "bar", "bar", null, "bar2", "bar2"]
```

Explanation

```
TimeMap timeMap = new TimeMap();
```

```
timeMap.set("foo", "bar", 1); // store the key "foo" and value "bar" along with timestamp = 1.
```

```
timeMap.get("foo", 1);      // return "bar"
```

```
timeMap.get("foo", 3);      // return "bar", since there is no value corresponding to foo at timestamp 3 and
                             // timestamp 2, then the only value is at timestamp 1 is "bar".
```

```
timeMap.set("foo", "bar2", 4); // store the key "foo" and value "bar2" along with timestamp = 4.
```

```
timeMap.get("foo", 4);      // return "bar2"
```

```
timeMap.get("foo", 5);      // return "bar2"
```

Key Idea:

To implement a time-based key-value store, we can use a dictionary to store the values associated with each key. For each key, we store a list of tuples representing the timestamp and the corresponding value. When querying a key at a specific timestamp, we perform binary search on the list of timestamps associated with that key to find the largest timestamp less than or equal to the given timestamp.

```
class TimeMap:
    def __init__(self):
        self.data = defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        self.data[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
```



```
values = self.data[key]
left, right = 0, len(values) - 1

while left <= right:
    mid = left + (right - left) // 2
    if values[mid][0] == timestamp:
        return values[mid][1]
    elif values[mid][0] < timestamp:
        left = mid + 1
    else:
        right = mid - 1

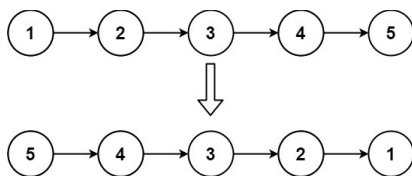
if right >= 0:
    return values[right][1]
return ""
```

Linked List

Problem: LeetCode 206 - Reverse Linked List

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

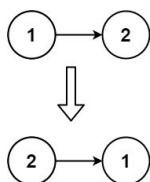
Example 1:



Input: head = [1,2,3,4,5]

Output: [5,4,3,2,1]

Example 2:



Input: head = [1,2]

Output: [2,1]

Example 3:

Input: head = []

Output: []

Key Idea:

To reverse a singly linked list, we need to reverse the direction of the pointers while traversing the list. We maintain three pointers: 'prev' (to keep track of the previous node), 'current' (to keep track of the current node), and 'next_node' (to keep track of the next node in the original list). In each iteration, we update the 'current.next' pointer to point to the 'prev' node, and then move 'prev' and 'current' pointers one step forward. We repeat this process until we reach the end of the original list, and the 'prev' pointer will be pointing to the new head of the reversed list.

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        prev = None
        current = head
```

```
while current:
    next_node = current.next
    current.next = prev
    prev = current
    current = next_node

return prev
```

Problem: LeetCode 21 - Merge Two Sorted Lists

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:

Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

Key Idea:

To merge two sorted linked lists 'l1' and 'l2', we can create a new linked list 'dummy' to hold the merged result. We maintain two pointers, 'current' and 'prev', to traverse through the two input lists. At each step, we compare the values at the 'current' pointers of 'l1' and 'l2', and add the smaller value to the 'dummy' list. We then move the 'current' pointer of the list with the smaller value one step forward. After iterating through both lists, if any list still has remaining elements, we append them to the 'dummy' list.

```
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode()
        current = dummy
```

```

while l1 and l2:
    if l1.val < l2.val:
        current.next = l1
        l1 = l1.next
    else:
        current.next = l2
        l2 = l2.next
    current = current.next

if l1:
    current.next = l1
elif l2:
    current.next = l2

return dummy.next

```

Problem: LeetCode 143 - Reorder List

You are given the head of a singly linked-list. The list can be represented as:

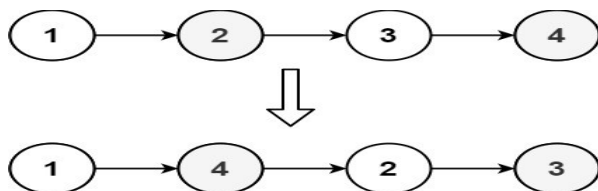
$$L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$$

Reorder the list to be on the following form:

$$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$$

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

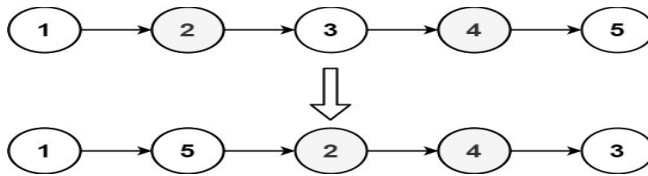
Example 1:



Input: head = [1,2,3,4]

Output: [1,4,2,3]

Example 2:



Input: head = [1,2,3,4,5]

Output: [1,5,2,4,3]

Key Idea:

To reorder a singly linked list, we can break the list into two halves, reverse the second half, and then merge the two halves alternately. First, we find the middle of the list using the slow and fast pointer technique. We reverse the second half of the list in place. Finally, we merge the two halves by alternating nodes from each half.

```

class Solution:
    def reorderList(self, head: ListNode) -> None:
        if not head or not head.next or not head.next.next:
            return

        # Find the middle of the list
        slow, fast = head, head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next

        # Reverse the second half of the list
        prev, current = None, slow.next
        slow.next = None
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node

        # Merge the two halves alternately
        p1, p2 = head, prev
        while p2:

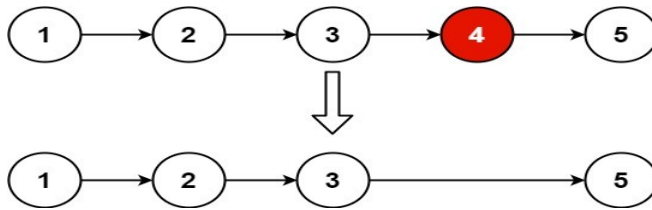
```

```
next_p1, next_p2 = p1.next, p2.next
p1.next = p2
p2.next = next_p1
p1, p2 = next_p1, next_p2
```

Problem: LeetCode 19 - Remove Nth Node From End of List

Given the head of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:



Input: head = [1,2,3,4,5], n = 2

Output: [1,2,3,5]

Example 2:

Input: head = [1], n = 1

Output: []

Example 3:

Input: head = [1,2], n = 1

Output: [1]

Key Idea:

To remove the n^{th} node from the end of a singly linked list, we can use the two-pointer approach. We maintain two pointers, 'fast' and 'slow', where 'fast' moves n nodes ahead of 'slow'. Then we move both pointers simultaneously until 'fast' reaches the end of the list. At this point, 'slow' will be pointing to the node just before the node to be removed. We update the 'slow.next' pointer to skip the node to be removed.

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        dummy = ListNode(0)
        dummy.next = head
        fast = slow = dummy

        # Move 'fast' n nodes ahead
        for _ in range(n):
            fast = fast.next
```

```

# Move both pointers until 'fast' reaches the end
while fast.next:
    fast = fast.next
    slow = slow.next

# Remove the nth node from the end
slow.next = slow.next.next

return dummy.next

```

Problem: LeetCode 138 - Copy List with Random Pointer

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a [deep copy](#) of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list.**

For example, if there are two nodes X and Y in the original list, where $X.random \rightarrow Y$, then for the corresponding two nodes x and y in the copied list, $x.random \rightarrow y$.

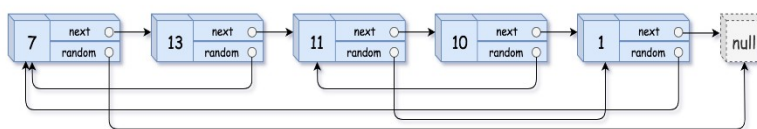
Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of $[val, random_index]$ where:

- **val**: an integer representing `Node.val`
- **random_index**: the index of the node (range from 0 to $n-1$) that the random pointer points to, or null if it does not point to any node.

Your code will **only** be given the head of the original linked list.

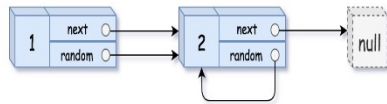
Example 1:



Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]

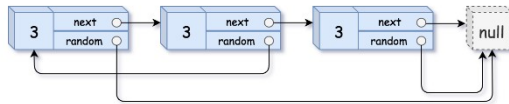
Example 2:



Input: head = [[1,1],[2,1]]

Output: [[1,1],[2,1]]

Example 3:



Input: head = [[3,null],[3,0],[3,null]]

Output: [[3,null],[3,0],[3,null]]

Key Idea:

To create a deep copy of a linked list with random pointers, we can follow a three-step approach. First, we duplicate each node in the original list and insert the duplicates right after their corresponding original nodes. Second, we update the random pointers of the duplicate nodes to point to the correct nodes. Finally, we split the combined list into two separate lists: the original list and the duplicated list.

class Solution:

```
def copyRandomList(self, head: "Node") -> "Node":
```

```
    if not head:
```

```
        return None
```

```
    # Step 1: Duplicate nodes and insert them in between the original nodes
```

```
    current = head
```

```
    while current:
```

```
        duplicate = Node(current.val)
```

```
        duplicate.next = current.next
```

```
        current.next = duplicate
```

```
        current = duplicate.next
```

```
    # Step 2: Update random pointers for the duplicate nodes
```

```
    current = head
```

```
    while current:
```

```
        if current.random:
```

```

        current.next.random = current.random.next

        current = current.next.next

# Step 3: Split the combined list into two separate lists
original = head
duplicate_head = head.next
current = duplicate_head
while original:
    original.next = original.next.next
    if current.next:
        current.next = current.next.next
    original = original.next
    current = current.next

return duplicate_head

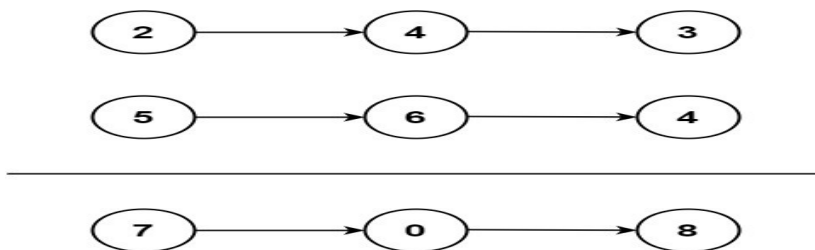
```

Problem: LeetCode 2 - Add Two Numbers

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.

Example 1:



Input: l1 = [2,4,3], l2 = [5,6,4]

Output: [7,0,8]

Explanation: 342 + 465 = 807.

Example 2:

Input: l1 = [0], l2 = [0]

Output: [0]

Example 3:

Input: l1 = [9,9,9,9,9,9], l2 = [9,9,9,9]

Output: [8,9,9,9,0,0,1]

Key Idea:

To add two numbers represented by linked lists, we can simulate the addition digit by digit while considering carry. We maintain a dummy node to build the resulting linked list. We iterate through the input lists, summing the corresponding digits along with any carry from the previous digit. We update the carry and create a new node with the sum digit. After processing both lists, if there is a carry remaining, we add a new node with the carry.

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode()
        current = dummy
        carry = 0

        while l1 or l2:
            val1 = l1.val if l1 else 0
            val2 = l2.val if l2 else 0
            total = val1 + val2 + carry

            carry = total // 10
            digit = total % 10

            current.next = ListNode(digit)
            current = current.next

            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next

        if carry:
            current.next = ListNode(carry)
```

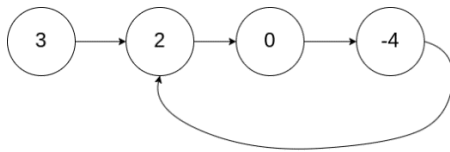
```
return dummy.next
```

Problem: LeetCode 141 - Linked List Cycle

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter.**

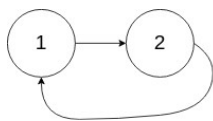
Return true *if there is a cycle in the linked list*. Otherwise, return false.

Example 1:

Input: head = [3,2,0,-4], pos = 1

Output: true

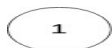
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:

Input: head = [1,2], pos = 0

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:

Input: head = [1], pos = -1

Output: false

Explanation: There is no cycle in the linked list.

Key Idea:

To detect a cycle in a linked list, we can use the Floyd's Tortoise and Hare algorithm. We maintain two pointers, 'slow' and 'fast', where 'slow' moves one step at a time and 'fast' moves two steps at a time. If there is a cycle in the linked list, the two pointers will eventually meet at some point. If there is no cycle, the 'fast' pointer will reach the end of the list.

```
class Solution:

    def hasCycle(self, head: ListNode) -> bool:

        if not head or not head.next:

            return False

        slow = head
        fast = head.next

        while slow != fast:

            if not fast or not fast.next:

                return False

            slow = slow.next
            fast = fast.next.next

        return True
```

Problem: LeetCode 287 - Find the Duplicate Number

Given an array of integers `nums` containing $n + 1$ integers where each integer is in the range $[1, n]$ inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

Example 1:

Input: `nums = [1,3,4,2,2]`

Output: 2

Example 2:

Input: `nums = [3,1,3,4,2]`

Output: 3

Example 3:

Input: `nums = [3,3,3,3,3]`

Output: 3

Key Idea:

To find the duplicate number in an array, we can treat the array as a linked list where each value points to the next value in the array. This problem is then reduced to finding the cycle in the linked list. We use the Floyd's

Tortoise and Hare algorithm to detect the cycle. Once the cycle is detected, we find the entrance of the cycle, which represents the duplicate number.

```
class Solution:

    def findDuplicate(self, nums: List[int]) -> int:

        slow = nums[0]
        fast = nums[0]

        # Move slow and fast pointers
        while True:

            slow = nums[slow]
            fast = nums[nums[fast]]

            if slow == fast:
                break

        # Find the entrance of the cycle
        slow = nums[0]
        while slow != fast:

            slow = nums[slow]
            fast = nums[fast]

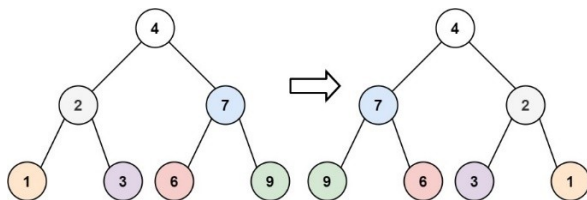
        return slow
```

Binary Tree

Problem: LeetCode 226 - Invert Binary Tree

Given the root of a binary tree, invert the tree, and return *its root*.

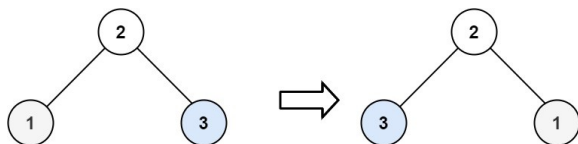
Example 1:



Input: root = [4,2,7,1,3,6,9]

Output: [4,7,2,9,6,3,1]

Example 2:



Input: root = [2,1,3]

Output: [2,3,1]

Example 3:

Input: root = []

Output: []

Key Idea:

To invert a binary tree, we can use a recursive approach. For each node, we swap its left and right subtrees, and then recursively invert the left and right subtrees.

```
class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
```



```
if not root:
    return None

# Swap left and right subtrees
root.left, root.right = root.right, root.left

# Recursively invert left and right subtrees
self.invertTree(root.left)
self.invertTree(root.right)

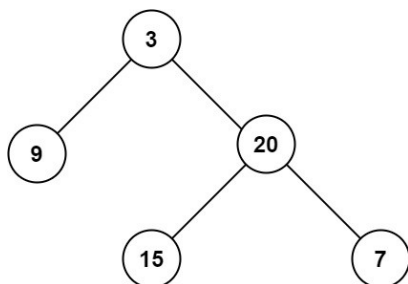
return root
```

Problem: LeetCode 104 - Maximum Depth of Binary Tree

Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: 3

Example 2:

Input: root = [1,null,2]

Output: 2

Key Idea:

To find the maximum depth of a binary tree, we can use a recursive approach. For each node, the maximum depth is the maximum of the depths of its left and right subtrees, plus one. We start from the root and recursively calculate the maximum depth for each subtree.

```
class Solution:
```

```
def maxDepth(self, root: TreeNode) -> int:
    if not root:
        return 0

    left_depth = self.maxDepth(root.left)
    right_depth = self.maxDepth(root.right)

    return max(left_depth, right_depth) + 1
```

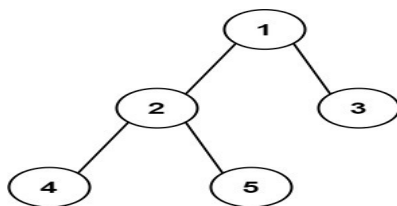
Problem: LeetCode 543 - Diameter of Binary Tree

Given the root of a binary tree, return *the length of the **diameter** of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.

Example 1:



Input: root = [1,2,3,4,5]

Output: 3

Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].

Example 2:

Input: root = [1,2]

Output: 1

Key Idea:

To find the diameter of a binary tree (the length of the longest path between any two nodes), we can use a recursive approach. For each node, the longest path passes either through the node or doesn't. The diameter is the maximum of three values: the diameter of the left subtree, the diameter of the right subtree, and the sum of the heights of the left and right subtrees (if the path passes through the node).

```
class Solution:
```

```

def diameterOfBinaryTree(self, root: TreeNode) -> int:

    def height(node):
        if not node:
            return 0

        left_height = height(node.left)
        right_height = height(node.right)

        self.diameter = max(self.diameter, left_height + right_height)

        return max(left_height, right_height) + 1

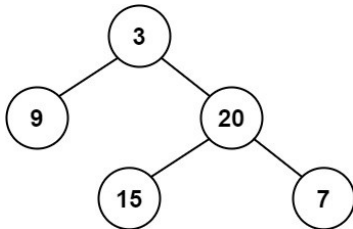
    self.diameter = 0
    height(root)
    return self.diameter

```

Problem: LeetCode 110 - Balanced Binary Tree

Given a binary tree, determine if it is **height-balanced**

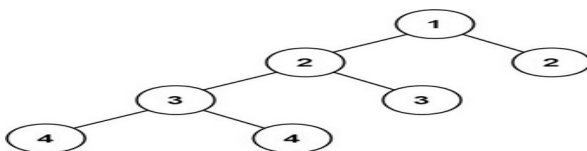
Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: true

Example 2:



Input: root = [1,2,2,3,3,null,null,4,4]

Output: false

Example 3:

Input: root = []

Output: true

Key Idea:

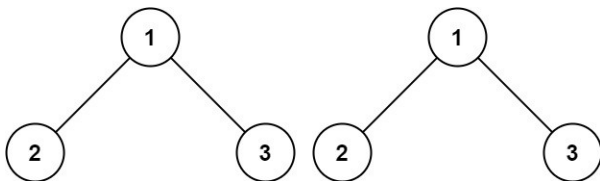
To check if a binary tree is balanced, we can use a recursive approach. For each node, we calculate the height of its left and right subtrees. If the difference in heights is greater than 1, the tree is not balanced. We continue this process for all nodes, recursively checking each subtree.

```
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        def height(node):
            if not node:
                return 0
            left_height = height(node.left)
            right_height = height(node.right)
            if abs(left_height - right_height) > 1:
                return float("inf") # Indicate imbalance
            return max(left_height, right_height) + 1
        return height(root) != float("inf")
```

Problem: LeetCode 100 - Same Tree

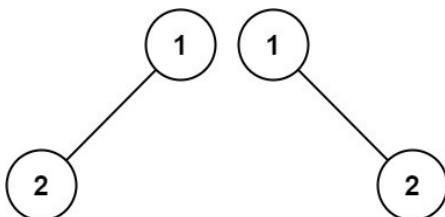
Given the roots of two binary trees p and q, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:

Input: p = [1,2,3], q = [1,2,3]

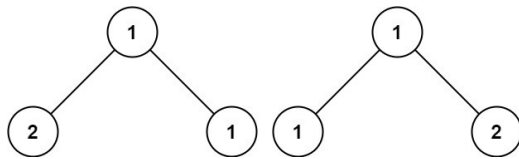
Output: true

Example 2:

Input: p = [1,2], q = [1,null,2]

Output: false

Example 3:



Input: p = [1,2,1], q = [1,1,2]

Output: false

Key Idea:

To determine if two binary trees are the same, we can use a recursive approach. For each pair of corresponding nodes, we compare their values and recursively check the left and right subtrees. If the values are equal and the left and right subtrees are also equal, then the trees are the same.

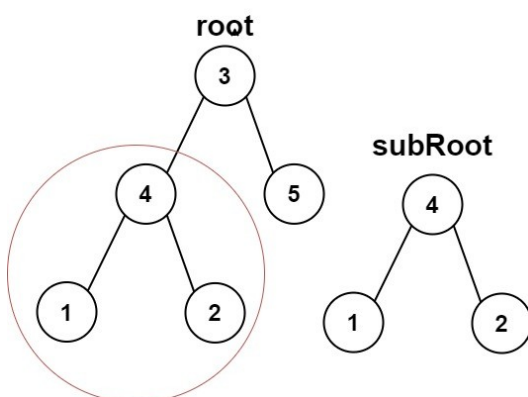
```
class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if not p and not q:
            return True
        if not p or not q or p.val != q.val:
            return False
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

Problem: LeetCode 57 - Subtree of Another Tree

Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

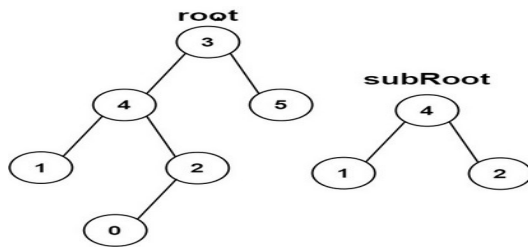
Example 1:



Input: root = [3,4,5,1,2], subRoot = [4,1,2]

Output: true

Example 2:



Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]

Output: false

Key Idea:

To check if one binary tree is a subtree of another, we can use a recursive approach. For each node in the main tree, we check if the current subtree rooted at that node is equal to the given subtree. If not, we recursively check the left and right subtrees.

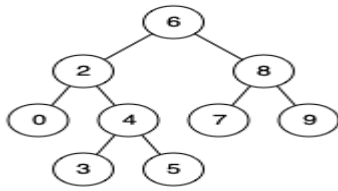
```
class Solution:
    def isSubtree(self, s: TreeNode, t: TreeNode) -> bool:
        if not s:
            return False
        if self.isSameTree(s, t):
            return True
        return self.isSubtree(s.left, t) or self.isSubtree(s.right, t)

    def isSameTree(self, p, q):
        if not p and not q:
            return True
        if not p or not q or p.val != q.val:
            return False
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

Problem: LeetCode 235 - Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

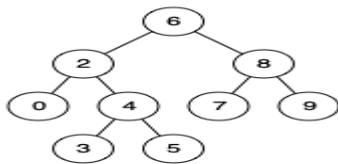
According to the [definition of LCA on Wikipedia](#): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Example 1:

Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

Output: 6

Explanation: The LCA of nodes 2 and 8 is 6.

Example 2:

Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

Output: 2

Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

Example 3:

Input: root = [2,1], p = 2, q = 1

Output: 2

Key Idea:

To find the lowest common ancestor (LCA) of two nodes in a binary search tree (BST), we can use a recursive approach. We compare the values of the two nodes with the current node's value. If both nodes are in the left subtree, we move to the left child. If both nodes are in the right subtree, we move to the right child. If one node is in the left subtree and the other is in the right subtree, we've found the LCA.

class Solution:

```
def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
```

```
    if root.val > p.val and root.val > q.val:
```

```
        return self.lowestCommonAncestor(root.left, p, q)
```

```
    elif root.val < p.val and root.val < q.val:
```

```
        return self.lowestCommonAncestor(root.right, p, q)
```

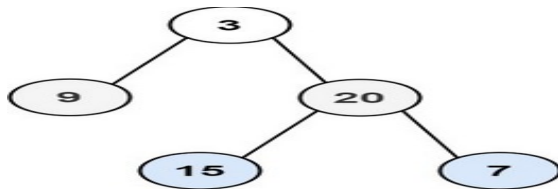
```
    else:
```

```
        return root
```

Problem: LeetCode 102 - Binary Tree Level Order Traversal

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

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: [[3],[9,20],[15,7]]

Example 2:

Input: root = [1]

Output: [[1]]

Example 3:

Input: root = []

Output: []

Key Idea:

To perform level order traversal of a binary tree, we can use a breadth-first search (BFS) approach. We start with the root node, and in each iteration, we process all nodes at the current level before moving to the next level. We use a queue to keep track of nodes at each level.

```
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []

        result = []
        queue = [root]

        while queue:
            level = []
            next_level = []
```



```

for node in queue:
    level.append(node.val)
    if node.left:
        next_level.append(node.left)
    if node.right:
        next_level.append(node.right)

result.append(level)
queue = next_level

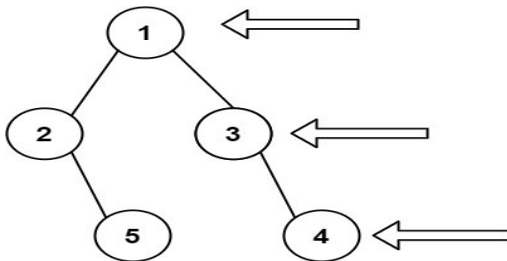
return result

```

Problem: LeetCode 199 - Binary Tree Right Side View

Given the root of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.

Example 1:



Input: root = [1,2,3,null,5,null,4]

Output: [1,3,4]

Example 2:

Input: root = [1,null,3]

Output: [1,3]

Example 3:

Input: root = []

Output: []

Key Idea:

To obtain the right side view of a binary tree, we can perform a level order traversal using a breadth-first search (BFS) approach. For each level, we add the last node's value to the result list. This way, we capture the rightmost nodes at each level.

```
class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        if not root:
            return []

        result = []
        queue = [root]

        while queue:
            level_size = len(queue)

            for i in range(level_size):
                node = queue.pop(0)
                if i == level_size - 1:
                    result.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

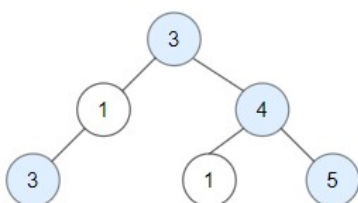
            return result
```

Problem: LeetCode 1448 - Count Good Nodes in Binary Tree

Given a binary tree root, a node X in the tree is named **good** if in the path from root to X there are no nodes with a value *greater than* X .

Return the number of **good** nodes in the binary tree.

Example 1:



Input: root = [3,1,4,3,null,1,5]

Output: 4

Explanation: Nodes in blue are **good**.

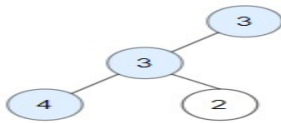
Root Node (3) is always a good node.

Node 4 -> (3,4) is the maximum value in the path starting from the root.

Node 5 -> (3,4,5) is the maximum value in the path

Node 3 -> (3,1,3) is the maximum value in the path.

Example 2:



Input: root = [3,3,null,4,2]

Output: 3

Explanation: Node 2 -> (3, 3, 2) is not good, because "3" is higher than it.

Example 3:

Input: root = [1]

Output: 1

Explanation: Root is considered as **good**.

Key Idea:

To count the number of good nodes in a binary tree, we can use a recursive depth-first search (DFS) approach. For each node, we keep track of the maximum value encountered on the path from the root to the current node. If the value of the current node is greater than or equal to the maximum value on the path, it is a good node. We increment the count and continue the DFS for the left and right subtrees.

```
class Solution:
    def goodNodes(self, root: TreeNode) -> int:
        def dfs(node, max_val):
            if not node:
                return 0

            if node.val >= max_val:
                max_val = node.val
                count = 1
            else:
                count = 0
```

```
count = 0

count += dfs(node.left, max_val)
count += dfs(node.right, max_val)

return count

return dfs(root, float("-inf"))
```

Problem: LeetCode 98 - Validate Binary Search Tree

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

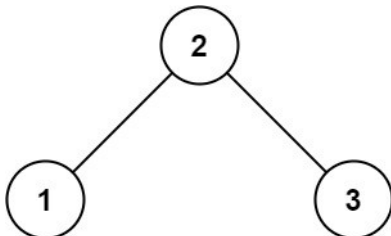
- The left

subtree

of a node contains only nodes with keys **less than** the node's key.

- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

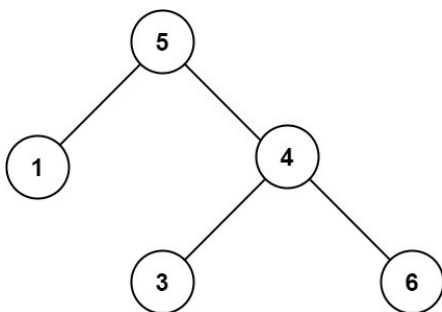
Example 1:



Input: root = [2,1,3]

Output: true

Example 2:



Input: root = [5,1,4,null,null,3,6]

Output: false

Explanation: The root node's value is 5 but its right child's value is 4.

Key Idea:

To validate if a binary tree is a valid binary search tree (BST), we can perform an in-order traversal and check if the values are in ascending order. During the in-order traversal, we keep track of the previously visited node's value and compare it with the current node's value.

```
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        def inorder_traversal(node, prev):
            if not node:
                return True

            if not inorder_traversal(node.left, prev):
                return False

            if prev[0] is not None and node.val <= prev[0]:
                return False
            prev[0] = node.val

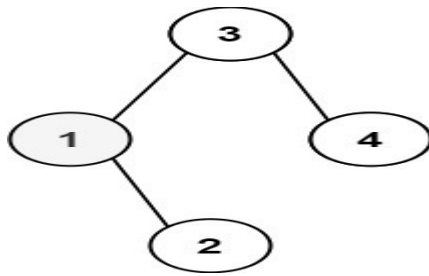
            return inorder_traversal(node.right, prev)

        prev = [None]
        return inorder_traversal(root, prev)
```

Problem: LeetCode 230 - Kth Smallest Element in a BST

Given the root of a binary search tree, and an integer k, return *the kth smallest value (1-indexed) of all the values of the nodes in the tree.*

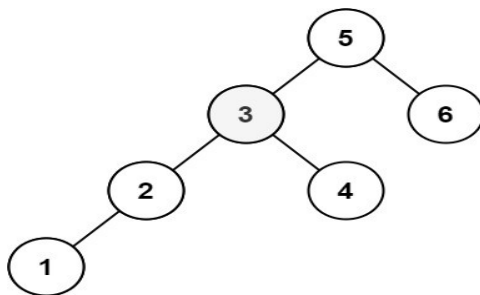
Example 1:



Input: root = [3,1,4,null,2], k = 1

Output: 1

Example 2:



Input: root = [5,3,6,2,4,null,null,1], k = 3

Output: 3

Constraints:

- The number of nodes in the tree is n.
- $1 \leq k \leq n \leq 10^4$
- $0 \leq \text{Node.val} \leq 10^4$

Follow up: If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

Key Idea:

To find the kth smallest element in a binary search tree (BST), we can perform an in-order traversal and keep track of the count of visited nodes. When the count reaches k, we've found the kth smallest element.

```

class Solution:
    def kthSmallest(self, root: TreeNode, k: int) -> int:
        def inorder_traversal(node):
            if not node:
                return []
  
```

```
left = inorder_traversal(node.left)

right = inorder_traversal(node.right)

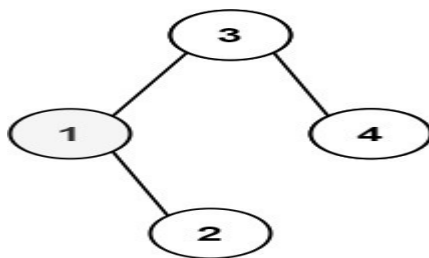
return left + [node.val] + right

inorder_values = inorder_traversal(root)
return inorder_values[k - 1]
```

Problem: LeetCode 105 - Construct Binary Tree from Preorder and Inorder Traversal

Given the root of a binary search tree, and an integer k , return the k^{th} smallest value (**1-indexed**) of all the values of the nodes in the tree.

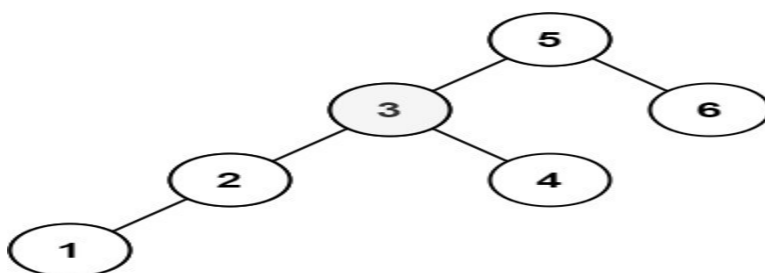
Example 1:



Input: root = [3,1,4,null,2], k = 1

Output: 1

Example 2:



Input: root = [5,3,6,2,4,null,null,1], k = 3

Output: 3

Key Idea:

To construct a binary tree from its preorder and inorder traversals, we can use a recursive approach. The first element in the preorder list is the root of the current subtree. We locate its position in the inorder list to determine the left and right subtrees. We recursively construct the left and right subtrees for each subtree.

```
class Solution:
```

```
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
```

```
        if not preorder or not inorder:
```

```
            return None
```

```
        root_val = preorder.pop(0)
```

```
        root = TreeNode(root_val)
```

```
        root_index = inorder.index(root_val)
```

```
        root.left = self.buildTree(preorder, inorder[:root_index])
```

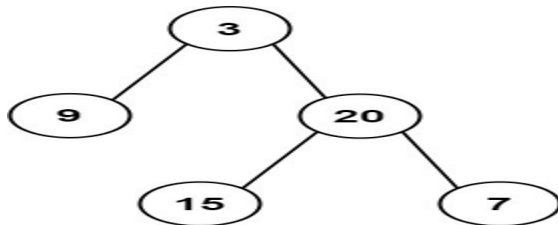
```
        root.right = self.buildTree(preorder, inorder[root_index + 1 :])
```

```
        return root
```

Problem: LeetCode 105 - Construct Binary Tree from Preorder and Inorder Traversal

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return *the binary tree*.

Example 1:



Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]

Output: [3,9,20,null,null,15,7]

Example 2:

Input: preorder = [-1], inorder = [-1]

Output: [-1]

Constraints:

- $1 \leq \text{preorder.length} \leq 3000$
- $\text{inorder.length} = \text{preorder.length}$
- $-3000 \leq \text{preorder}[i], \text{inorder}[i] \leq 3000$
- preorder and inorder consist of **unique** values.
- Each value of inorder also appears in preorder.

- preorder is **guaranteed** to be the preorder traversal of the tree.
- inorder is **guaranteed** to be the inorder traversal of the tree.

Key Idea:

To construct a binary tree from its preorder and inorder traversals, we can use a recursive approach. The first element in the preorder list is the root of the current subtree. We locate its position in the inorder list to determine the left and right subtrees. We recursively construct the left and right subtrees for each subtree.

```
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        if not preorder or not inorder:
            return None

        root_val = preorder.pop(0)
        root = TreeNode(root_val)
        root_index = inorder.index(root_val)

        root.left = self.buildTree(preorder, inorder[:root_index])
        root.right = self.buildTree(preorder, inorder[root_index + 1 :])

        return root
```

Backtracking