1. Two Sum Problem

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

Example

- **Input**:
  - `nums = [2, 7, 11, 15]`
  - `target = 9`
- **Output**:
  - `[0, 1]`
- **Explanation**:
  - `nums[0] + nums[1] = 2 + 7 = 9`, so return `[0, 1]`.

Note

- Each input would have exactly one solution.
- You may not use the same element twice.

```
def two_sum_sorted(nums, target):
    # Start with two pointers: one at the beginning
    # and one at the end of the array
    left, right = 0, len(nums) - 1

    # Continue looping until the two pointers cross each other
    while left < right:
        # Calculate the sum of the elements pointed by the left
        # and right pointers
        current_sum = nums[left] + nums[right]
        # If the current sum is equal to the target, we have
        # found the solution
        if current_sum == target:
            # Returning indices, or you could
            # return [nums[left], nums[right]]
            # to get the actual numbers
            return [left, right]

        # If the current sum is less than the target, we need a larger sum
        # Increment the left pointer to move to a bigger number
        elif current_sum < target:
            left += 1

        # If the current sum is more than the target, we need a smaller sum
        # Decrement the right pointer to move to a smaller number
        else:
            right -= 1

    # If no such pair is found that adds up to the target,
    # return an empty list
    return []

# Example usage
# Consider a sorted array where you want two numbers
# to add up to a specific target
nums = [1, 2, 4, 5, 6, 10]  # This is a sorted array
target = 8
result = two_sum_sorted(nums, target)
print(result)
# This should print indices like [1, 4], corresponding to numbers 2 and 6
```

```
[1, 4]
```

- **Time Complexity**: (O(n))
  - We use a two-pointer approach, which requires a single pass through the array. Each move (either `left += 1` or `right -= 1`) brings us closer to the solution, making this a linear-time algorithm.
- **Space Complexity**: (O(1))
  - The algorithm uses only a constant amount of space for the pointers `left` and `right`, so no additional space grows with input size.

```python
def two_sum(nums, target):
    # Dictionary to store the complement and its index
    num_to_index = {}

    # Iterate over the list to find the two numbers
    for index, num in enumerate(nums):
        # Calculate the complement
        complement = target - num

        # If the complement exists in the dictionary, we found a solution
        if complement in num_to_index:
            return [num_to_index[complement], index]

        # Otherwise, store the number with its index
        num_to_index[num] = index

    # Return an empty list if no solution is found -
    # though per the problem statement,
    # there should always be one solution.
    return []

# Example usage:
nums = [2, 7, 11, 15]
target = 9
print(two_sum(nums, target))  # Output: [0, 1]
```

```
[0, 1]
```

1. **Iteration 1**: Index 0, Number 2 → Complement 7 (not in `num_to_index`), store `{2: 0}`.
2. **Iteration 2**: Index 1, Number 7 → Complement 2 (found in `num_to_index`), return `[0, 1]`.

- **Time Complexity**: (O(n))
  - We iterate through the list once, checking and updating the hash map with each element. Each lookup and insertion in a hash map is (O(1)) on average, making the total time complexity linear.
- **Space Complexity**: (O(n))
  - In the worst case, we store all elements of the input array in the hash map, so space complexity is linear.

2. Contains Duplicate

The problem is to determine if a given list of integers, `nums`, contains any duplicates. A duplicate value means that there is at least one integer that appears more than once in the list. The function should return `True` if there are any duplicates and `False` otherwise.

Sample Input and Output

Example:

- **Input**: `[1, 2, 3, 1]`
- **Output**: `True`

Explanation: The integer `1` appears twice in the list, thus the output is `True`.

```python
from typing import List
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        # Initialize an empty set to keep track of unique elements encountered
        unique_set = set()

        # Iterate over each element in the input list
        for i in nums:
            # Check if the element is already in the set
            if i in unique_set:
                # If found in the set, it's a duplicate, return True
                return True
            # Add the element to the set since it's unique so far
            unique_set.add(i)

        # If loop completes without returning True, all elements are unique
        return False
```

```python
solution = Solution()
result = solution.containsDuplicate([1, 2, 3, 4])
print(result)  # Expected output: False
```

```
False
```

Complexity Analysis

- **Time Complexity**: O(n), where n is the number of elements in the list `nums`. This is because we iterate over each element of the list once.
- **Space Complexity**: O(n) in the worst case, where n is the total number of unique elements, which depends on how many unique elements can exist in the given list.

3. Majority Element

The problem requires finding the majority element in an array, which is defined as the element that appears more than `n/2` times, where `n` is the length of the array. One viable algorithm to solve this problem efficiently is the Boyer-Moore Voting Algorithm. This algorithm aims to find a candidate for the majority element with linear time complexity and constant space complexity by progressively canceling out the counts of different elements.

Sample Input and Output

Input: `[3, 2, 3]`
Output: `3`

Input: `[2, 2, 1, 1, 1, 2, 2]`
Output: `2`

Here, in the first test case, the number `3` appears `2` times out of `3`, which is more than half the size of the array. In the second test case, `2` appears `4` times out of `7`.

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        # Initialize a counter to zero
        count = 0
        # This variable will hold the candidate for the majority element
        majority_element = None

        # Iterate through each number in the array
        for num in nums:
            # When count is zero, choose the current element
            # as a new candidate
            if count == 0:
                majority_element = num
                count += 1
            elif num == majority_element:
```

```python
                # If the current element is the same as the candidate,
                # increment the count
                count += 1
            else:
                # If the current element is different, decrement the count
                count -= 1

        # Return the candidate as it will be the majority element
        return majority_element

# Example test case
solution = Solution()
test_case = [3, 2, 3]
print(solution.majorityElement(test_case))  # Output: 3
```

```
3
```

Time and Space Complexity

- **Time Complexity:** O(n), where n is the number of elements in the list. We traverse through the list only once.
- **Space Complexity:** O(1), as we are using only a few additional variables, independent of the input size.

4. Valid anagram

Problem Statement:
Given two strings `s` and `t`, determine if `t` is an anagram of `s`. Two strings are anagrams if one string can be rearranged to form the other string. Both strings consist of lowercase Latin letters.

Sample Input:
- `s = "anagram"` - `t = "nagaram"`

Sample Output:
- `True`

Explanation:
The string "nagaram" is a rearrangement of the string "anagram". Both strings have the same character counts, hence they are anagrams.

```python
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        # Check if the lengths of the strings are the same.
        # If not, they can't be anagrams and we can immediately return False.
        if len(s) != len(t):
            return False

        # Create two dictionaries to store the frequency of each character.
        count_s, count_t = {}, {}

        # Loop through both strings simultaneously by their indices.
        for i in range(len(s)):
            # Increment the character count for the
            # current character in string s
            # Retrieve the current count from count_s using get,
            # which defaults
            # to 0 if the character is not found.
            count_s[s[i]] = 1 + count_s.get(s[i], 0)

            # Similarly, increment the character count for the current character
            # in string t
            count_t[t[i]] = 1 + count_t.get(t[i], 0)

        # Compare the dictionaries after processing both strings.
        # If they're identical, it means both strings have the same character
        # counts and are anagrams.
        return count_s == count_t
```

```python
# Create an instance of the Solution class
solution = Solution()

# Test case: Check if "cinema" is an anagram of "iceman"
result = solution.isAnagram("cinema", "iceman")
print(result)  # Output: True
```

```
True
```

Time and Space Complexity

- **Time Complexity:** O(n), where n is the length of the strings. We loop over the strings only once.

- **Space Complexity:** O(1), because the primary data structures used (the dictionaries) utilize a constant amount of space relative to the problem size given the fixed character set constraint (26 lowercase letters).

5. Group Anagrams

Given an array of strings, group the anagrams together. An anagram is a word formed by rearranging the letters of another word.

You can return the answer in any order— the main requirement is that all anagrams are grouped together in sublists.

Example:

Input:

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

```
from collections import defaultdict

def groupAnagrams(strs):
    anagrams = defaultdict(list)

    for s in strs:
        # Initialize count array
        count = [0] * 26  # There are 26 possible lowercase characters

        # Count the frequency of each character in the string
        for char in s:
            count[ord(char) - ord('a')] += 1

        # Use the tuple of counts as the key in the hashmap
        key = tuple(count)
        anagrams[key].append(s)
    # Return all values in the dictionary as a list of lists
    return list(anagrams.values())

# Example usage:
strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
print(groupAnagrams(strs))
```

```
[['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']]
```

- **Time Complexity:** (O(NK)), where (N) is the number of strings, and (K) is the maximum length of a string, as we iterate through each character of every string.
- **Space Complexity:** (O(NK)) for storing the grouped anagrams in the dictionary.

6. Longest Substring Without Repeating Characters

The objective is to find the length of the longest substring without repeating characters in a given string `s`. A substring is a contiguous sequence of characters within the string. The challenge is to efficiently manage and track these characters to determine the maximum possible length of such substrings without repetition.

Sample Input and Output

- **Input:** `s = "abcabcbb"`
- **Output:** `3`
- **Explanation:** The longest substring without repeating characters is "abc", with a length of 3.

```python
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        unique_set = set()  # A set to store unique characters of the current substring
        left_pointer = 0    # The starting index of the current substring
        max_length = 0      # Variable to keep track of the maximum length found

        # Iterate over each character in the string 's' using 'right_pointer' as the index
        for right_pointer, char in enumerate(s):
            # If the character is already in the set, move the left_pointer to the right
            # until the character is removed from the current substring
            while char in unique_set:
                unique_set.remove(s[left_pointer])  # Remove the character at left_pointer fr
                left_pointer += 1                   # Increment left_pointer to narrow the wi

            # Add the new unique character to the set
            unique_set.add(char)

            # Calculate the length of the current substring and update max_length if it's lar
            max_length = max(max_length, right_pointer - left_pointer + 1)

        return max_length
```

```
# Example Test Case
solution = Solution()
s = "abcabcbb"
print(solution.lengthOfLongestSubstring(s))  # Output: 3
```

```
3
```

- **Time Complexity:** O(n), where n is the length of the string `s`, as each character is processed at most twice (once when added to the set and once when removed).
- **Space Complexity:** O(min(m, n)), where m is the size of the character set (unique characters that can be in `s`) and n is the length of the string. This is due to the space used by the set to store the current substring's characters.

7. Subarray Sum Equals K

Description

Given an integer array `nums` and an integer `k`, you need to find the total number of continuous subarrays whose sum equals to `k`.

Example

Input

- `nums = [1, 1, 1]`
- `k = 2`

Output

- 2

Explanation

The array has the following subarrays whose sum equals `2`: 1. The subarray `[1, 1]` starting from index `0` to `1`. 2. The subarray `[1, 1]` starting from index `1` to `2`.

Constraints

- The length of the array `nums` will be between `1` and `20,000`.
- Elements of `nums` will be integers ranging from `-1000` to `1000`.
- The integer `k` will be in the range of `-1e7` to `1e7`.

```python
def subarraySum(nums, k):
    # Initialize count of subarrays found
    count = 0
    # This variable keeps track of the cumulative sum up to the current position in the array
    current_sum = 0
    # Using a dictionary to map cumulative sums to their counts
    # Start with the base case: the cumulative sum of 0 has occurred once
    prefix_sum_count = {0: 1}

    # Iterate over each number in the input array
    for num in nums:
        # Update the cumulative sum by adding the current number
        current_sum += num

        # Calculate the needed sum which, when subtracted from current_sum, would equal k
        needed_sum = current_sum - k

        # Check if needed_sum is already in the prefix_sum_count map
        # If it is, increment the count by the number of times needed_sum has occurred
        if needed_sum in prefix_sum_count:
            count += prefix_sum_count[needed_sum]

        # Update the hashmap with the current cumulative sum
        # If it exists already, increment its count, otherwise add it with a count of 1
        prefix_sum_count[current_sum] = prefix_sum_count.get(current_sum, 0) + 1

    # Return the total count of subarrays found that sum to k
    return count

# Example usage
nums = [1, 2, 3]
k = 3
print(subarraySum(nums, k))  # Output will be 2 (subarrays are [1, 2] and [3])
```

```
2
```

Example: nums = [1, 2, 3], k = 3

Iteration Details:

1. **When `num = 1`:**
   - `current_sum = 1`, `needed_sum = -2`.
   - `-2` is not in `prefix_sum_count`; `count` remains 0.
   - Update `prefix_sum_count` to `{0: 1, 1: 1}`.
2. **When `num = 2`:**
   - `current_sum = 3`, `needed_sum = 0`.
   - `0` is in `prefix_sum_count` (once), indicating one subarray ([1, 2]) sums to `k`.
   - Increment `count` to 1.
   - Update `prefix_sum_count` to `{0: 1, 1: 1, 3: 1}`.
3. **When `num = 3`:**
   - `current_sum = 6`, `needed_sum = 3`.
   - `3` is in `prefix_sum_count` (once), indicating another subarray ([3]) sums to `k`.
   - Increment `count` to 2.
   - Update `prefix_sum_count` to `{0: 1, 1: 1, 3: 1, 6: 1}`.

Conclusion:

There are two subarrays that sum to `k`. The function returns `2`.

Complexity Analysis

- **Time Complexity:** `O(n)`, where `n` is the number of elements in the array. This is because we traverse the array only once.
- **Space Complexity:** `O(n)`, due to the potential storage needed for the cumulative sums in the HashMap.

8. Find All Anagrams in a String

Given two strings `s` and `p`, return an array of all the start indices of `p`'s anagrams in `s`. You may return the answer in any order.

Example 1:

- **Input:** `s = "cbaebabacd"`, `p = "abc"`
- **Output:** `[0,6]`
- **Explanation:**
  - The substring with start index = 0 is `"cba"`, which is an anagram of `"abc"`.
  - The substring with start index = 6 is `"bac"`, which is an anagram of `"abc"`.

Constraints:

- The length of both input strings may vary based on specific problem requirements (not specified here).

```
from collections import Counter
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        # If the length of p is greater than the length of s, it's impossible to have an anag
        if len(p) > len(s):
            return []

        # Initialize a Counter to track the current window's character counts in s.
        s_count = Counter()
        # Initialize a Counter with the character counts of string p.
        p_count = Counter(p)
        # This will store the starting indices of the anagrams of p in s.
        result = []

        # Length of the string p to define the sliding window's length.
        p_len = len(p)

        # Iterate over each character in the string s.
        for i in range(len(s)):
            # Add the current character to the sliding window counter.
            s_count[s[i]] += 1
            # If the window size exceeds p's length, we need to remove the oldest character.
            if i >= p_len:
                # Identify the character that is sliding out of the window.
                out_char = s[i-p_len]
                # If the count of that character is 1, we remove it from the counter.
                if s_count[out_char] == 1:
                    del s_count[out_char]
                else:
                    # Otherwise, just decrement its count.
                    s_count[out_char] -= 1

            # If current window's character count matches p's character count, we found an ar
            if s_count == p_count:
                # Append the starting index of the anagram.
                result.append(i - p_len + 1)

        # Return the list of starting indices of anagrams found.
        return result
```

```
s = "cbaebabacd"
p = "abc"
expected_output = [0, 6]

solution = Solution()
result = solution.findAnagrams(s, p)
print("Output:", result)  # Should output [0, 6]
print("Test Passed:", result == expected_output)  # Should output True
```

```
Output: [0, 6]
Test Passed: True
```

- **Time Complexity:** O(n), where n is the length of the string `s`, since each character is processed at most twice.
- **Space Complexity:** O(1), considering the counter size is constant due to the fixed alphabet size.

9. Squares of a Sorted Array

Problem

Given an integer array `nums` sorted in non-decreasing order, return an array of the squares of each number sorted in non-decreasing order.

Example 1:

Input: `nums = [-4,-1,0,3,10]`
Output: `[0,1,9,16,100]`
Explanation: After squaring, the array becomes `[16,1,0,9,100]`. After sorting, it becomes `[0,1,9,16,100]`.

Example 2:

Input: `nums = [-7,-3,2,3,11]`
Output: `[4,9,9,49,121]`

Constraints:

- `1 <= nums.length <= 10^4`
- `-10^4 <= nums[i] <= 10^4`
- `nums` is sorted in non-decreasing order.

```
def sortedSquares(nums):
    # Initialize two pointers: left at the start, right at the end
    left = 0
    right = len(nums) - 1

    # Prepare an output array of the same length as nums initialized with zeros
    result = [0] * len(nums)

    # Start filling the result array from the last position
    position = len(nums) - 1

    # Loop until the left pointer exceeds the right pointer
    while left <= right:
        # Calculate the square of the elements at both pointers
        left_square = nums[left] ** 2
        right_square = nums[right] ** 2

        # Compare squared values: move the larger one to the result[position]
        if left_square > right_square:
            # If left square is larger, place it at the current position
            result[position] = left_square
            # Move the left pointer to the right
            left += 1
        else:
            # If right square is larger or equal, place it at the current position
            result[position] = right_square
            # Move the right pointer to the left
            right -= 1

        # Move the position backward
        position -= 1
```

```python
    return result

# Example usage:
nums = [-4, -1, 0, 3, 10]
print(sortedSquares(nums))  # Output: [0, 1, 9, 16, 100]
```

```
[0, 1, 9, 16, 100]
```

- **Time Complexity**: (O(n)), where (n) is the length of the input array, because we iterate through the array at most once with both pointers.
- **Space Complexity**: (O(n)), for the output array that stores the squared values.

10. 3Sum

Given an array of `n` integers, find all unique triplets `(a, b, c)` in the array such that `a + b + c = 0`.

Example:

- Input: `[-1, 0, 1, 2, -1, -4]`
- Output: `[[-1, 0, 1], [-1, -1, 2]]`

```python
from typing import List

class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # Sort the input array to facilitate two-pointer approach
        nums.sort()
        n = len(nums)
        triplets = []  # To store the resulting triplets

        # Iterate through the array, treating each number as a
        # potential start of a triplet
        for i in range(n - 2):
            # Since the list is sorted, if the current number is
            # greater than zero,
            # all further numbers will also be greater than zero,
            # making it impossible
            # to sum to zero
            if nums[i] > 0:
                break
            # Skip the number if it's the same as the previous
            # one to avoid duplicates
            if i > 0 and nums[i] == nums[i - 1]:
                continue

            # Use two pointers to find the other two numbers of the triplet
            left, right = i + 1, n - 1

            # Continue while the left pointer is less than the right pointer
            while left < right:
                current_sum = nums[i] + nums[left] + nums[right]

                # If the current sum is less than zero,
                # move the left pointer to the right
                if current_sum < 0:
                    left += 1
                # If the current sum is greater than zero,
                # move the right pointer to the left
                elif current_sum > 0:
                    right -= 1
                # If the current sum is zero, we found a valid triplet
                else:
                    triplets.append([nums[i], nums[left], nums[right]])

                    # Move the left pointer to the right and
                    # the right pointer to the left
                    left += 1
                    right -= 1

                    # Skip the same elements to avoid duplicate triplets
                    while left < right and nums[left] == nums[left - 1]:
                        left += 1
                    while left < right and nums[right] == nums[right + 1]:
                        right -= 1

        return triplets

# Example Test Cases
solution = Solution()

# Test Case 1: Basic Test
nums1 = [-1, 0, 1, 2, -1, -4]
```

```python
print(solution.threeSum(nums1))
# Possible output: [[-1, -1, 2], [-1, 0, 1]]
```

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

Time complexity: O(n^2), where n is the number of elements in the input list, due to the sorting step (O(n log n)) and the nested two-pointer approach (O(n^2)).
Space complexity: O(1) if we disregard the space used for the output, as no extra space proportional to input size is used beyond the input array itself.

11. 3Sum Closest

Given an integer array `nums` and an integer `target`, find three integers in `nums` such that the sum is closest to `target`. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Constraints

- `3 <= nums.length <= 500`
- `-1000 <= nums[i] <= 1000`
- `-10^4 <= target <= 10^4`

Examples

Input: nums = [-1, 2, 1, -4], target = 1

Output:: 2

Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2)

```python
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        # Sort the list to use the two-pointer technique effectively
        nums.sort()

        # Initialize the closest sum to infinity for comparison
        closest_sum = float("inf")

        # Length of the list
        n = len(nums)

        # Loop through each number, treating it as the first number of the triplet
        for i in range(n - 2):
            # Initialize two pointers, starting after the current number i
            left = i + 1
            right = n - 1

            # Use the two-pointer technique to find the closest sum
            while left < right:
                # Calculate the current sum of the triplet
                current_sum = nums[i] + nums[left] + nums[right]

                # If the current sum is exactly the target, return it immediately
                if current_sum == target:
                    return current_sum

                # Check if the current sum is closer to the target than the previously recor
                if abs(current_sum - target) < abs(closest_sum - target):
                    closest_sum = current_sum

                # Adjust pointers based on how the current sum compares to the target
                if current_sum > target:
                    # If current sum is greater than target, move the right pointer left to
                    right -= 1
                else:
                    # If current sum is less than target, move the left pointer right to inc
                    left += 1

        # Return the closest sum found
        return closest_sum
```

```
nums = [-1, 2, 1, -4]
solution = Solution()
target = 1
result = solution.threeSumClosest(nums, target)
print(f"Test Case 1: Expected: 2, Got: {result}")
```

```
Test Case 1: Expected: 2, Got: 2
```

- **Time Complexity**: (O(n^2)), where (n) is the number of elements in the input list `nums`. This is due to the nested loop created by the two-pointer technique.

- **Space Complexity**: (O(1)), as the algorithm uses a constant amount of extra space regardless of the input size.

12. Sort Colors

Given an array `nums` with `n` objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1

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

```python
def sortColors(nums):
    # Initialize the pointers.
    # 'low' will track the position where the next 0 should be placed.
    # 'mid' will scan through the list to decide elements' positions.
    # 'high' will track the position where the next 2 should be placed.
    low, mid, high = 0, 0, len(nums) - 1

    # Iterate through the list using 'mid' pointer.
    while mid <= high:
        if nums[mid] == 0:
            # If the current element is 0, we need to swap it with the element at 'low'
            # position because 'low' marks the boundary for 0's.
            nums[low], nums[mid] = nums[mid], nums[low]
            # Move 'low' and 'mid' pointers to the right, as we've correctly placed
            # a 0 at 'low'.
            low += 1
            mid += 1
        elif nums[mid] == 1:
            # If the current element is 1, it's already in the correct place,
            # because 1's are in the middle. Just move the 'mid' pointer.
            mid += 1
        else:  # nums[mid] == 2
            # If the current element is 2, we need to swap it with the element at 'high'
            # position because 'high' marks the boundary for 2's.
            # Note: We do not increment 'mid' here because the element swapped
            # from 'high' to 'mid' needs to be evaluated.
            nums[mid], nums[high] = nums[high], nums[mid]
            # Move 'high' pointer to the left, as we've correctly placed
            # a 2 at 'high'.
            high -= 1

# Example usage:
nums = [2, 0, 2, 1, 1, 0]
sortColors(nums)
print(nums)  # Output: [0, 0, 1, 1, 2, 2]
```

```
[0, 0, 1, 1, 2, 2]
```

- **Time Complexity**: O(n)
  The algorithm makes a single pass over the array with `n` elements, ensuring that each element is processed a constant number of times. Therefore, the time complexity is linear, O(n).
- **Space Complexity**: O(1)
  The algorithm utilizes a constant amount of extra space for the pointers (`low`, `mid`, `high`), irrespective of the input size. Hence, the space complexity is O(1).

13. Container With Most Water

Problem Description:

You are given an array `height` of length `n`. Each element in the array represents the height of a vertical line drawn at that index. The width between each pair of lines is 1. You need to find two lines, which together with the x-axis form a container, such that the container holds the most water.

Example:

Given `height = [1,8,6,2,5,4,8,3,7]`, the function should return `49`, which corresponds to the area between the indices `1` and `8`.

```
from typing import List

class Solution:
    def maxArea(self, height: List[int]) -> int:
        # Initialize two pointers, one starting from the beginning (left)
        # and the other from the end (right) of the list.
        left, right = 0, len(height) - 1

        # Variable to store the maximum area found so far.
        max_area = 0

        # Continue iterating until the two pointers meet.
        while left < right:
            # Calculate the width between the two pointers: (right - left)
            width = right - left

            # Determine the height as the minimum of the
            # two heights at the pointers.
            current_height = min(height[left], height[right])

            # Compute the current area by multiplying width and height.
            current_area = width * current_height

            # Update the maximum area if the current area is greater.
            max_area = max(current_area, max_area)

            # Move the pointer pointing to the shorter line to try
            # and find a taller container.
            # This is because moving the shorter line could potentially
            # increase the area.
            if height[left] < height[right]:
                # Move the 'left' pointer to the right to attempt
                # a larger area.
                left += 1
            else:
                # Move the 'right' pointer to the left.
                right -= 1

        # After the loop, return the maximum area found during all iterations.
        return max_area

# Define the list of heights representing the vertical lines
# on the container walls.
height = [1, 8, 6, 2, 5, 4, 8, 3, 7]

# Create an instance of the Solution class to access the maxArea function.
```

```
solution = Solution()

# Use the instance to invoke the maxArea function with the list of heights.
max_area_result = solution.maxArea(height)

# Print the result which is the maximum area that can be contained.
print("The maximum area is:", max_area_result)
```

```
The maximum area is: 49
```

- **Time Complexity:** (O(n)) due to a single pass through the list.
- **Space Complexity:** (O(1)) as it uses a constant amount of extra space.

14. 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 `""`. The test cases will be generated such that the answer is unique.

Example:

- **Input:** `s = "ADOBECODEBANC"`, `t = "ABC"`
- **Output:** `"BANC"`
- **Explanation:** The minimum window substring `"BANC"` includes 'A', 'B', and 'C' from string `t`.

```
from collections import Counter

class Solution:
    def minWindow(self, s: str, t: str) -> str:
        # Count each character in string T to know what is needed
        target_counter = Counter(t)

        # A counter to keep track of characters in the current window
        window_counter = Counter()

        # Pointers for the left and right boundaries of the window
        left = 0
        # Variables to track the minimum window
        min_left = -1
        min_size = float('inf')  # Infinity for comparison to find minimum

        # Variables to track the number of unique target characters met
        formed = 0
        required = len(target_counter)  # Total unique characters to be matched

        # Expand the window with the right pointer
        for right, char in enumerate(s):
            # Add one character from the right to the window
            window_counter[char] += 1

            # Check if the current character's frequency in the window matches the target
            if char in target_counter and window_counter[char] == target_counter[char]:
                formed += 1

            # Contract the window from the left as long as all target characters are matched
            while formed == required:
                # Update the minimum window if the current one is smaller
                if right - left + 1 < min_size:
                    min_size = right - left + 1
                    min_left = left  # Store left boundary of the smallest window

                # The character at the current left position will be "removed" from the windo
                window_counter[s[left]] -= 1

                # If a character is less than needed, decrement formed
                if s[left] in target_counter and window_counter[s[left]] < target_counter[s[
                    formed -= 1

                # Move the left pointer right to try and find a smaller window
                left += 1

        # If no valid window is found, return an empty string
        return "" if min_left == -1 else s[min_left:min_left + min_size]

# Example usage:
s = "ADOBECODEBANC"
t = "ABC"
solution = Solution()
print(solution.minWindow(s, t))  # Output: "BANC"
```

```
BANC
```

- **Time Complexity**: O(m + n), where `m` is the length of string `s` and `n` is the length of string `t`. This results from scanning through `s` with two pointers, effectively making one pass through `s` and handling character frequencies.
- **Space Complexity**: O(n + k), where `n` is the number of unique characters in `t` and `k` is the number of unique characters in `s`. This space is used by the frequency counters (`target_counter` and `window_counter`).

15. Sliding Window Maximum

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Examples

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output:

[3,3,5,5,6,7]

Explanation:

Window position	Max
`[1  3  -1] -3  5  3  6  7`	3
`1 [3  -1  -3] 5  3  6  7`	3
`1  3 [-1  -3  5] 3  6  7`	5
`1  3  -1 [-3  5  3] 6  7`	5
`1  3  -1  -3 [5  3  6] 7`	6
`1  3  -1  -3  5 [3  6  7]`	7

```python
from collections import deque
from typing import List

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        # Initialize a deque to store indices of the array elements.
        # It will help us keep track of the maximum in the current window.
        index_queue = deque()

        # List to store the maximum of each sliding window.
        max_val = []

        # Iterate through each element in the array with both index and value.
        for index, value in enumerate(nums):
            # Remove elements from the front of the deque if they are outside
            # the current sliding window's range (i.e., older than index - k + 1).
            if index_queue and index - k + 1 > index_queue[0]:
                index_queue.popleft()

            # Remove elements from the back of the deque if the current element
            # is greater than the elements at those indices. This ensures that
            # the deque stores indices of elements in decreasing order by value.
            while index_queue and nums[index_queue[-1]] <= value:
                index_queue.pop()

            # Add the current element's index to the deque. At this point,
            # all elements in the deque are greater than or equal to the current element.
            index_queue.append(index)

            # If we've processed at least `k` elements (the first complete window),
            # append the maximum for the current window to `max_val`.
            # The maximum is the element at the index stored at the front of the deque.
            if index >= k - 1:
                max_val.append(nums[index_queue[0]])

        # Return the list containing the maximum of each sliding window.
        return max_val

# Example usage
sol = Solution()
print(sol.maxSlidingWindow([1,3,-1,-3,5,3,6,7], 3)) # Output: [3,3,5,5,6,7]
```

```
[3, 3, 5, 5, 6, 7]
```

- **Time Complexity**: O(n), where n is the number of elements in the input list `nums`. Each element is processed at most twice (once added and once removed from the deque).
- **Space Complexity**: O(k), where k is the size of the sliding window. This space is used by the deque to store indices of elements within the current window.

16. Longest Substring Without Repeating Characters

Description:
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.

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # Set to store unique characters in the current window
        unique_set = set()

        # Left pointer of the sliding window
        left_pointer = 0

        # Variable to store the maximum length of substring found
        max_length = 0

        # Iterate over the string using the right pointer
        for right_pointer, char in enumerate(s):
            # If character is already in the set, slide the window
            # from the left until the character can be added
            while char in unique_set:
                unique_set.remove(s[left_pointer])  # Remove the leftmost character
                left_pointer += 1  # Move the left pointer to the right

            # Add the current character to the set
            unique_set.add(char)

            # Calculate the length of the current window and update max length if needed
            max_length = max(max_length, right_pointer - left_pointer + 1)

        # Return the maximum length found
        return max_length

# Test case
s = "abcabcbb"
# Explanation: The answer is "abc", with the length of 3.
print(Solution().lengthOfLongestSubstring(s))  # Output: 3
```

```
3
```

- **Time Complexity:** (O(n))
    - The algorithm processes each character in the string once, moving the left and right pointers at most (n) times, where (n) is the length of the string.
- **Space Complexity:** (O(min(n, m)))
    - Using a set to store characters in the current window, space is proportional to the size of the window, bounded by the smaller of (n) (total characters in the string) and (m) (number of unique characters in the character set).

17. 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:

- **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.

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # Initialize variables to track the maximum profit and the minimum price seen so far
        max_price = 0
        min_price = float('inf')

        # Iterate over each price in the list of prices.
        for price in prices:
            # Update the maximum profit by checking if selling at the current price
            # and buying at the minimum price seen so far gives a higher profit.
            max_price = max(max_price, price - min_price)

            # Update the minimum price to the lesser of the current price or the
            # minimum price seen so far.
            min_price = min(price, min_price)

        # Return the maximum profit found.
        return max_price

# Test Case
prices = [7, 1, 5, 3, 6, 4]
solution = Solution()
print(solution.maxProfit(prices))  # Expected output: 5

# Explanation of Test Case:
# The maximum profit can be achieved by buying on day 2 (price = 1) and selling on day 5 (pri
# The profit is 6 - 1 = 5, which is the maximum possible profit for this sequence of prices.
```

```
5
```

- **Time Complexity:** (O(n))
  The algorithm iterates through each element of the `prices` list exactly once, where (n) is the number of days (or elements in the list).
- **Space Complexity:** (O(1))
  The algorithm uses a constant amount of extra space regardless of the size of the input, as it only requires a few extra variables to hold the state.

18. Meeting Rooms

Given an array of meeting time intervals consisting of start and end times `[[s1, e1], [s2, e2], ...]` (inclusive of the start time and exclusive of the end time), determine if a person can attend all meetings without any scheduling conflicts.

Example 1

- **Input:** `intervals = [[0, 30], [5, 10], [15, 20]]`
- **Output:** `False`
- **Explanation:**
  - The meeting `[0, 30]` overlaps with `[5, 10]`, making it impossible to attend all without conflict.

Example 2

- **Input:** `intervals = [[7, 10], [2, 4]]`
- **Output:** `True`
- **Explanation:**
  - The meetings `[7, 10]` and `[2, 4]` do not overlap, allowing attendance at all meetings.

```python
def canAttendMeetings(intervals):
    # Sort the intervals based on their start times
    intervals.sort(key=lambda x: x[0])

    # Iterate through the sorted intervals and check for overlaps
    for i in range(1, len(intervals)):
        # If there is an overlap, return False
        if intervals[i][0] < intervals[i-1][1]:
            return False

    # If no overlaps are found, return True
    return True

# Example usage:
intervals = [[0, 30], [5, 10], [15, 20]]
print(canAttendMeetings(intervals))  # Output: False
```

```
False
```

- **Time Complexity:** O(n log n) - This is due to the sorting of the intervals.
- **Space Complexity:** O(1) - Assuming the sort is done in place, aside from the input storage.

19. Insert Interval

Problem Description

You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [starti, endi]` represent the start and the end of the ith interval and `intervals` is sorted in ascending order by `starti`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `starti` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` after the insertion.

Note that you don't need to modify `intervals` in-place. You can make a new array and return it.

Example

Example 1:

- **Input:**
  - `intervals = [[1,3],[6,9]]`
  - `newInterval = [2,5]`
- **Output:** `[[1,5],[6,9]]`

```python
from typing import List

class Solution:
    def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List[int]]:
        # Edge case: if the list of intervals is empty, just return the new interval wrapped
        if not intervals:
            return [newInterval]

        n = len(intervals)
        target = newInterval[0]  # The start of the new interval that we'll compare against

        left, right = 0, n - 1  # Set initial binary search boundaries

        # Perform binary search to find the insertion position based on interval starts
        while left <= right:
            mid = left + (right - left) // 2  # Avoid potential overflow with this computatio
            if intervals[mid][0] < target:
                left = mid + 1  # Move to the right half if mid interval starts before the n
            else:
                right = mid - 1  # Move to the left half otherwise

        # Insert the new interval at the found index 'left'
        intervals.insert(left, newInterval)

        # This list will hold the merged, non-overlapping intervals
        res = []

        # Iterate through the intervals to merge them as needed
        for i in intervals:
            # If the result list is empty or the last interval in result does not overlap wi
            if not res or res[-1][1] < i[0]:
                res.append(i)
            else:
                # Merge the current interval with the last interval in result by updating the
                res[-1][1] = max(res[-1][1], i[1])

        # Return the merged list of intervals
        return res
```

- **Time Complexity:** (O(n)) - The complexity is dominated by the merging step, which involves iterating over all intervals. The binary search for insertion takes (O(log n)), but is outweighed by the (O(n)) for merging.
- **Space Complexity:** (O(n)) - In the worst case, the extra space usage results from the `res` list, which, in the worst case, could store all intervals if none are merged.

20. Merge Intervals

Problem Statement

Given an array of intervals where `intervals[i] = [start_i, end_i]`, merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example

- **Input:** `intervals = [[1,3],[2,6],[8,10],[15,18]]`
- **Output:** `[[1,6],[8,10],[15,18]]`
- **Explanation:** Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

```python
from typing import List

class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        # First, sort the intervals list based on the starting times
        intervals.sort(key=lambda x: x[0])

        # Initialize an empty list to store the merged intervals
        merged = []

        # Iterate over each interval in the sorted list
        for interval in intervals:
            # Check if the merged list is empty or there is no overlap with the last interval
            if not merged or merged[-1][1] < interval[0]:
                # If no overlap, append the current interval to merged
                merged.append(interval)
            else:
                # If there is overlap, merge the current interval with the last interval in m
                merged[-1][1] = max(merged[-1][1], interval[1])

        # Return the merged list of intervals
        return merged

# Test case
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
solution = Solution()
result = solution.merge(intervals)
print(result)  # Output: [[1, 6], [8, 10], [15, 18]]
```

```
[[1, 6], [8, 10], [15, 18]]
```

- **Time Complexity:** O(n log n), where n is the number of intervals. The complexity arises from sorting the `intervals` list. The traversal through the list to merge intervals takes O(n) time.
- **Space Complexity:** O(n), where n is the number of intervals. This is due to the storage required for the `merged` list in the worst-case scenario where no intervals are overlapping.

21. Non-overlapping Intervals

Given an array of intervals `intervals` where `intervals[i] = [starti, endi]`, return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping. Note that intervals which only touch at a point are considered non-overlapping. For example, `[1, 2]` and `[2, 3]` are non-overlapping.

Example

- **Input:** `intervals = [[1,2],[2,3],[3,4],[1,3]]`
- **Output:** `1`
- **Explanation:** `[1,3]` can be removed and the rest of the intervals are non-overlapping.

```
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        # Sort intervals based on their end times to maximize the number of non-overlapping i
        intervals.sort(key=lambda x: x[1])

        # Initialize the count of intervals to be removed
        remove_interval_count = 0

        # Initialize the end time of the first interval
        end_time = intervals[0][1]

        # Iterate over the rest of the intervals
        for start, end in intervals[1:]:
            # If the start of the current interval is greater than or equal to the end of the
            if start >= end_time:
                # Update the end time to the end of the current interval
                end_time = end
            else:
                # If the current interval overlaps, increment the removal count
                remove_interval_count += 1

        # Return the total number of intervals that need to be removed to make the rest non-o
        return remove_interval_count
```

```
intervals = [[1, 2], [2, 3], [3, 4], [1, 3]]
solution = Solution()
print(solution.eraseOverlapIntervals(intervals))  # Output should be 1
```

```
1
```

Time Complexity: (O(n log n)) due to the sorting of intervals.

Space Complexity: (O(1)) since the algorithm uses a constant amount of extra space.

22. Meeting Rooms II

Problem Statement

Given an array of intervals where each interval represents a meeting's start and end time, determine the minimum number of conference rooms required to accommodate all the meetings without overlapping. Each meeting is represented as a pair of integers `[start, end]` where `start < end`.

Example

Input:

[[0, 30], [5, 10], [15, 20]]

Output:

2

Explanation: - The first meeting is from 0 to 30. - The second meeting is from 5 to 10.m Since it overlaps with the first meeting, a new room is needed. - The third meeting is from 15 to 20. At this time, the second meeting has ended, but the first is still ongoing, so a new room is needed. - Thus, a total of 2 rooms are required to host all meetings without any overlap.

Another Example

Input:

[[7, 10], [2, 4]]

Output:

1

Explanation: - The first meeting is from 7 to 10. - The second meeting is from 2 to 4. There is no overlap, so both can share the same room. - Thus, only 1 room is required.

```python
import heapq

def minMeetingRooms(intervals):
    if not intervals:
        return 0

    # Sort the intervals based on start time
    intervals.sort(key=lambda x: x[0])

    # Min-heap to track end times of meetings
    min_heap = []

    # Add the first meeting's end time to the heap
    heapq.heappush(min_heap, intervals[0][1])

    # Iterate over remaining intervals
    for i in range(1, len(intervals)):
        # If the room is free (i.e., the earliest meeting has ended), remove it
        if intervals[i][0] >= min_heap[0]:
            heapq.heappop(min_heap)

        # Add the current meeting's end time to the heap
        heapq.heappush(min_heap, intervals[i][1])

    # The size of the heap is the number of rooms required
    return len(min_heap)

# Example usage:
meetings = [[0, 30], [5, 10], [15, 20]]
print(minMeetingRooms(meetings))  # Output: 2
```

```
2
```

- **Time Complexity**: (O(N log N)) due to sorting the intervals and using a min-heap for managing meeting end times.
- **Space Complexity**: (O(N)) to store the end times of meetings in the min-heap.

23. Gas Station

Problem

There are n gas stations along a circular route, where the amount of gas at the i-th station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the i-th station to its next (i + 1)-th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1. If there exists a solution, it is guaranteed to be unique.

Example

Example 1:

Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2] Output: 3

Explanation:

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

```
def canCompleteCircuit(gas, cost):
    # Initialize total and current tank balances
    total_tank, current_tank = 0, 0
    # Start checking from the first gas station
    starting_station = 0

    # Iterate over each gas station
    for i in range(len(gas)):
        # Calculate the net gain/loss of gas at station i
        balance = gas[i] - cost[i]
        # Update the total tank balance after visiting station i
        total_tank += balance
        # Update the current tank balance after visiting station i
        current_tank += balance

        # If at any station the current tank balance is negative,
        # it indicates that we cannot start from our current starting station
        if current_tank < 0:
            # Set the next station as the new starting station
            starting_station = i + 1
            # Reset the current tank balance because we'll start fresh from the next station
            current_tank = 0

    # After checking all the stations, if total_tank is non-negative,
    # it means the circuit can be completed, return the starting station
    if total_tank >= 0:
        return starting_station
    else:
        # Otherwise, it is impossible to complete the circuit with any starting point
        return -1

# Example usage
gas = [1, 2, 3, 4, 5]   # Gas available at each station
cost = [3, 4, 5, 1, 2]  # Cost to travel to the next station
print(canCompleteCircuit(gas, cost))  # Output: 3
```

```
3
```

- **Time Complexity**: O(n) - We iterate over the list of gas stations once, making the complexity linear relative to the number of stations.
- **Space Complexity**: O(1) - The algorithm uses a constant amount of extra space, irrespective of the input size.

24. 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

Input:

nums = [1,2,3,4]

Output:

[24,12,8,6]

```
from typing import List

class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        # Initialize the result array with 1s. This array will store the final product resul
        result = [1] * len(nums)

        # Compute the prefix product for each element.
        # The prefix product for `result[i]` is the product of all elements to the left of `
        for i in range(1, len(nums)):
            result[i] = result[i-1] * nums[i-1]

        # Initialize a variable `postfix` to accumulate the postfix product.
        postfix = 1
        # Compute the postfix product for each element and multiply it with the current pref
        # The postfix product is the product of all elements to the right of `i`.
        for i in range(len(nums)-1, -1, -1):
            # Multiply the current value in result[i] (which is the prefix product) by the c
            result[i] = result[i] * postfix
            # Update the postfix product for the next iteration to include the current eleme
            postfix = postfix * nums[i]

        return result
```

```
# Create an instance of the Solution class
solution = Solution()

# Define the input
nums = [1, 2, 3, 4]

# Get the output using the productExceptSelf method
output = solution.productExceptSelf(nums)
```

```
# Print the output
print(output)  # Expected output: [24, 12, 8, 6]
```

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

- **Time Complexity**: (O(n)) - The function iterates through the input list twice (once for prefix products and once for postfix products), where (n) is the number of elements in the input list.
- **Space Complexity**: (O(1)) (excluding the output list) - The function uses a constant amount of additional space for the `postfix` variable; the `result` list is not considered extra space since it is part of the output.‘

25. Rotate Array

Problem Description

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

Example:

text = [1,2,3,4,5,6,7], , k = 3

Output: [5,6,7,1,2,3,4]

Explanation:

- Rotate 1 step to the right: ([7,1,2,3,4,5,6])
- Rotate 2 steps to the right: ([6,7,1,2,3,4,5])
- Rotate 3 steps to the right: ([5,6,7,1,2,3,4])

```
class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        """
        Rotates the array `nums` to the right by `k` steps in place.
        """
        # Calculate the effective rotations needed, as rotating by the length of the array re
        # This handles cases where k is greater than the length of the array.
        k %= len(nums)

        # Slice the array into two parts:
        # 1. The last k elements (which will move to the front after rotation).
        # 2. The rest of the array (which moves to the back after rotation).
        # Then concatenate these two parts in reverse order.
        nums[:] = nums[-k:] + nums[:-k]

        # nums[-k:] provides the last k elements of the array.
        # nums[:-k] provides all elements except the last k ones.
        # The result of the concatenation assigns the rotated version back to the original a
```

- **Time Complexity**: (O(n)) - The solution involves slicing the array and concatenating two lists, both of which require linear time proportional to the size of the array.
- **Space Complexity**: (O(n)) - While the operation modifies the array in place, the slicing operation creates temporary sublists that occupy additional space.

26. 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`.

```
from typing import List

class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        # Convert the list to a set to eliminate duplicates and allow O(1) lookups.
        nums = set(nums)
        # Initialize the variable to track the length of the longest consecutive sequence fou
        long_seq = 0

        # Iterate over each number in the set.
        for i in nums:
            # Only consider starting a new sequence if `i-1` is not in the set,
            # which means `i` is the beginning of a potential sequence.
            if i - 1 not in nums:
                current_num = i
                current_seq = 1

                # Check for the rest of the sequence incrementally.
                while current_num + 1 in nums:
                    current_num += 1
                    current_seq += 1

                # Update the longest sequence found so far.
                long_seq = max(current_seq, long_seq)

        # Return the length of the longest consecutive sequence.
        return long_seq

# Test case
nums = [100, 4, 200, 1, 3, 2]
solution = Solution()
print(solution.longestConsecutive(nums))  # Output: 4

# Explanation of the test case:
# The longest consecutive sequence in the list [100, 4, 200, 1, 3, 2] is [1, 2, 3, 4],
# which has a length of 4.
```

```
4
```

- **Time Complexity:** O(n), where n is the number of elements in the input list. Hash set operations (insertions and lookups) are O(1) on average, and iterating through the list requires O(n) time.
- **Space Complexity:** O(n), primarily due to storing all elements of the input list in a hash set to enable O(1) lookups.

27. Combination Sum

Description

Given an array of distinct integers `candidates` and a target integer `target`, return a list of all unique combinations of `candidates` where the chosen numbers sum to `target`. You may return the combinations in any order.

The same number may be chosen from `candidates` an unlimited number of times. Two combinations are unique if the **frequency** of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to `target` is less than 150 combinations for the given input.

Example

- **Input:** `candidates = [2,3,6,7], target = 7`
- **Output:** `[[2,2,3],[7]]`
- **Explanation:**
    - 2 and 3 are candidates, and $2 + 2 + 3 = 7$. Note that 2 can be used multiple times.
    - 7 is a candidate, and $7 = 7$.
    - These are the only two combinations.

```python
from typing import List

class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        # The backtracking function to explore combinations.
        def backtrack(remaining: int, start: int, path: List[int], res: List[List[int]]):
            # If remaining is negative, the current combination exceeds the target.
            if remaining < 0:
                return  # Stop exploring this path as it's not valid.

            # If remaining is zero, we have found a valid combination.
            if remaining == 0:
                # Append a copy of the current path to the results
                res.append(list(path))
                return  # Return to explore other possibilities.

            # Iterate over the candidates starting from the current position.
            for i in range(start, len(candidates)):
                # Include the candidate number in the current path.
                path.append(candidates[i])
                # Recursively call backtrack with the updated target (remaining - current ca
                # Crucially, we pass `i` again to allow the same element to be reused.
                backtrack(remaining - candidates[i], i, path, res)
                # Backtrack: Remove the last added number to try another possibility.
                path.pop()

        # This will hold all the valid combinations we find.
        result = []
        # Start the backtracking process with an empty path and the full target.
        backtrack(target, 0, [], result)
        return result  # Return the list of all valid combinations.

# Example usage:
sol = Solution()
print(sol.combinationSum([2, 3, 6, 7], 7))  # Output: [[2, 2, 3], [7]]
```

```
[[2, 2, 3], [7]]
```

- **Time Complexity:** (O(N^{T/M + 1}))
  Where (N) is the number of candidates, (T) is the target value, and (M) is the minimal value among the candidates. The complexity arises because, in the worst case, we explore all possible combinations to reach the target.
- **Space Complexity:** (O(T/M))
  The space complexity is primarily due to the recursion stack, where (T/M) represents the maximum depth of the recursive calls. Each recursive call adds a new candidate to the current combination, potentially up to a maximum depth determined by the smallest candidate value needed to sum to the target.

28. Contiguous Array

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

Problem Overview:

The task is to identify the maximum length of any contiguous subarray that contains the same number of 0s and 1s in the given binary array `nums`.

Additional Information:

Example:

- **Input:** `nums = [0, 1]`
- **Output:** `2`
- **Explanation:** `[0, 1]` is the longest contiguous subarray with an equal number of `0` and `1`.
- **Constraints:**

  - The array `nums` contains only binary values (0 and 1).

```
class Solution:
    def findMaxLength(self, nums: List[int]) -> int:
        # Initialize variables to track the balance of 1s and 0s, and the maximum length of b
        balance = 0
        max_length = 0

        # A dictionary to store the first occurrence of each balance
        # Starting with a balance of 0 at index -1 (to handle the entire array being balanced
        balance_map = {0: -1}

        # Iterate through the array
        for index, value in enumerate(nums):
            # Update the balance: +1 for 1, -1 for 0
            balance += 1 if value == 1 else -1

            # Check if this balance was seen before
            if balance in balance_map:
                # Calculate the length of the balanced subarray
                # It is the difference between the current index and the index of the first o
                max_length = max(max_length, index - balance_map[balance])
            else:
                # If this balance hasn't been seen, record its first occurrence
                balance_map[balance] = index

        # Return the maximum length of the balanced subarray found
        return max_length
```

```
# Test case: A sample input to test the functionality
nums = [0, 1, 0, 1, 1, 0]

# Instantiate the Solution class
solution = Solution()

# Call the method and print the result
print(solution.findMaxLength(nums))  # Expected output: 4
```

6

- **Time Complexity**: O(n), where n is the length of the input list `nums`. The function makes exactly one pass through the list.

- **Space Complexity**: O(n), due to the use of a hash map (dictionary) to store the first occurrences of each balance, which could potentially store up to n different keys.