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`.

```
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
```

```
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:
```

```
                # 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.

```
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`.

```
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 an
            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
```

```
    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]]`

```
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]
```

```
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

Example 1

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)

```
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]`

```
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`.

```python
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`).

```
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]
```

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # A set to store unique characters in the current window
        unique_set = set()
        # Left pointer of the window
        left_pointer = 0
        # Variable to store the maximum length of substring found
        max_length = 0

        # Iterate through the string with right_pointer as the moving index
        for right_pointer, char in enumerate(s):
            # If the character is already in the set, it means we have a duplicate
            while char in unique_set:
                # Remove the character at left_pointer from the set to shrink the window
                unique_set.remove(s[left_pointer])
                # Move the left_pointer to the right
                left_pointer += 1

            # Add the current character to the set
            unique_set.add(char)
            # Adjust the maximum length if the current window is larger
            max_length = max(max_length, right_pointer - left_pointer + 1)

        # Return the max length of substring with all unique characters found
        return max_length

# Test case: Let's take the string "abcabcbb" as an example to see how the solution works
solution = Solution()
result = solution.lengthOfLongestSubstring("abcabcbb")
```

```
print(result)  # Expected output is 3, that is for the substring "abc"
```

```
3
```

```
def is_palindrome(s):
    # Convert to lowercase and remove non-alphanumeric characters for an accurate check
    clean_s = ''.join(char.lower() for char in s if char.isalnum())

    # Initialize two pointers
    left, right = 0, len(clean_s) - 1

    # Move the pointers towards each other
    while left < right:
        if clean_s[left] != clean_s[right]:
            return False
        left += 1
        right -= 1

    return True

# Example usage
string = "A man, a plan, a canal, Panama"
if is_palindrome(string):
    print(f'"{string}" is a palindrome.')
else:
    print(f'"{string}" is not a palindrome.')
```

```
"A man, a plan, a canal, Panama" is a palindrome.
```

```
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        # If the lengths of the strings are not equal, they cannot be anagrams.
        if len(s) != len(t):
            return False

        # Initialize two dictionaries to count the frequency of each character
        # in both strings s and t.
        count_s, count_t = {}, {}

        # Iterate over the indices of the strings
```

```
        for i in range(len(s)):
            # For string s, increment the count of the character s[i] in count_s.
            # The get method retrieves the current count of s[i], defaulting to 0 if it doesn
            count_s[s[i]] = 1 + count_s.get(s[i], 0)

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

        # After processing both strings, compare the two dictionaries.
        # If the dictionaries are equal, it means both strings have identical character count
        # and thus they are anagrams of each other.
        return count_s == count_t
```

```
# Test case
s = "listen"
t = "silent"

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

# Call the isAnagram method and print the result
result = solution.isAnagram(s, t)
print(result)  # Expected output: True
```

```
True
```

```
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 an
            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
```

```
from collections import Counter

class Solution:
    def minWindow(self, s: str, t: str) -> str:
```

```
        # Create a counter for characters in t
        target_counter = Counter(t) # {'A': 1, 'B': 1, 'C': 1}

        # Initialize the window counter
        window_counter = Counter()

        left = 0  # Initialize left pointer
        min_left = -1  # Variable to store starting index of the minimum window
        val_char_count = 0  # Count of valid characters based on target comparison
        min_size = float('inf')  # Size of the minimum valid window

        # Loop over each character in the string 's' using a right pointer
        for right, char in enumerate(s):
            # Add current character to the window counter
            window_counter[char] += 1

            # If the character frequency in window does not exceed target, increment valid co
            if window_counter[char] <= target_counter[char]:
                val_char_count += 1

            # When the valid character count matches the length of 't', a valid window is fou
            while val_char_count == len(t):
                # Check if this window is the smallest found so far
                if right - left + 1 < min_size:
                    min_size = right - left + 1
                    min_left = left

                # At this point, the window is valid; try to shrink it by moving left pointer
                # If removing the left character might affect the valid count
                if window_counter[s[left]] <= target_counter[s[left]]:
                    val_char_count -= 1

                # Remove the character at the left pointer from the window
                window_counter[s[left]] -= 1

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

        # Return the minimum window if found, otherwise return an empty string
        return "" if min_left < 0 else s[min_left:min_left + min_size]

# Testing the function with the example test case
```

```
solution = Solution()
result = solution.minWindow("ADOBECODEBANC", "ABC")
print(result)  # Output should be "BANC"
```

```
BANC
```

```
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        # This array will track the frequency of each character in the current window.
        count = [0] * 26
        left = 0  # The starting index of our current window.
        max_count = 0  # The highest frequency of any single character in our current window
        result = 0  # The length of the longest valid window we've found so far.

        for right in range(len(s)):
            # Update the count for the current character at index 'right'.
            count[ord(s[right]) - ord('A')] += 1

            # Update the max_count to reflect the highest frequency character in the window.
            max_count = max(max_count, count[ord(s[right]) - ord('A')])

            # If the current window size (right - left + 1) minus the highest frequency
            # character is greater than k, reduce the window size from the left.
            while (right - left + 1) - max_count > k:
                count[ord(s[left]) - ord('A')] -= 1
                left += 1

            # Check if the current window is the largest valid window we've seen.
            result = max(result, right - left + 1)

        return result

# Let's walk through a test case:
# s = "AABABBA", k = 1
# The goal is to find the longest substring where we can replace up to 1 character
# to make all the characters in the substring the same.

solution = Solution()
test_result = solution.characterReplacement("AABABBA", 1)
print(test_result)  # Expected output is 4, for the substring "ABBA".
```

```
4
```

```python
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        # If the list of strings is empty, return an empty string as there's no common prefi
        if not strs:
            return ""

        # Iterate over each character index of the first string
        for i in range(len(strs[0])):
            # Get the character at the current index of the first string
            char = strs[0][i]

            # Compare the character with the corresponding character in all other strings
            for j in range(1, len(strs)):
                # If the current string length is less than i or the character doesn't match
                # return the substring from the start to the current index (i) as the result
                if i == len(strs[j]) or strs[j][i] != char:
                    return strs[0][:i]

        # If no mismatch is found, return the entire first string as all strings contain it a
        return strs[0]


# Test case
# Input: A list of strings to find the longest common prefix
input_strs = ["flower", "flow", "flight"]
# Expected Output: "fl"
# Explanation: The longest common prefix among the input strings ("flower", "flow", "flight")

solution = Solution()
result = solution.longestCommonPrefix(input_strs)
print(f"The longest common prefix is: '{result}'")
```

```
The longest common prefix is: 'fl'
```