CODING INTERVIEW PREPARATION

Day 2: Two Pointers & Sliding Window

Complete Study Guide with Multiple Solutions

TABLE OF CONTENTS

Page

DAY 2: TWO POINTERS & SLIDING WINDOW

Key Concepts for Today:

- **Two Pointers** technique for efficient array traversal
- **Sliding Window** for substring and subarray problems
- **String manipulation** and character validation
- **Optimization patterns** for reducing time complexity
- **Window expansion and contraction** strategies

Problems Index:

1. **Valid Palindrome** (Easy)
2. **Best Time to Buy and Sell Stock** (Easy)
3. **Longest Substring Without Repeating Characters** (Medium)
4. **Longest Repeating Character Replacement** (Medium)
5. **Minimum Window Substring** (Hard)
6. **Container With Most Water** (Medium)
7. **3Sum** (Medium)
8. **Trapping Rain Water** (Hard)

CORE CONCEPTS EXPLANATION

Two Pointers Technique

Definition: A technique using two index variables to traverse data structures, typically from opposite ends or at different speeds.[1] [2]

Common Patterns:

- **Opposite Ends**: Start from beginning and end, move towards center
- **Fast/Slow**: Two pointers moving at different speeds
- **Same Direction**: Both pointers move in same direction with different conditions

Key Advantages:

- Reduces **O(n²)** brute force to **O(n)** time complexity[1]
- **O(1)** space complexity in most cases
- Eliminates need for nested loops in many scenarios

When to Use:

- Sorted arrays with target sum problems
- Palindrome validation
- Finding pairs or triplets
- Container/water trapping problems

Sliding Window Technique

Definition: A method to process arrays/strings by maintaining a subset (window) of elements and dynamically adjusting window size.[3] [4]

Window Types:

- **Fixed Size**: Window size remains constant

- **Variable Size**: Window expands and contracts based on conditions
- **Shrinking Window**: Window only contracts when condition is met

Core Operations:

- **Expand**: Add elements to right side of window
- **Contract**: Remove elements from left side of window
- **Slide**: Move entire window while maintaining size

Time Complexity Benefit:

- Each element enters and exits window at most once
- Achieves **O(n)** time instead of **O(n²)** for many substring problems[3]

String Manipulation Patterns

Character Validation:

- `char.IsLetterOrDigit()` for alphanumeric checking
- `char.ToLower()` for case-insensitive comparison
- ASCII arithmetic for character operations

Frequency Counting:

- Use arrays for limited character sets (a-z: size 26)
- Use HashMap for unlimited character sets
- Track character counts in sliding windows

Why These Concepts Matter in Interviews

Optimization Thinking:

- Demonstrates ability to reduce time complexity systematically
- Shows understanding of space-time trade-offs
- Reveals pattern recognition skills

Real-world Applications:

- Network packet processing (sliding window protocols)
- Database query optimization
- Real-time data stream analysis
- Memory management algorithms

PROBLEM 1: VALID PALINDROME

Problem Statement:

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

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

Test Cases:

```
Example 1:
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.

Example 2:
Input: s = "race a car"
Output: false
Explanation: "raceacar" is not a palindrome.

Example 3:
Input: s = " "
Output: true
Explanation: After removing non-alphanumeric characters, s becomes empty "", which is a p
```

Interview Questions & Answers:

1. **Q:** "How do you handle non-alphanumeric characters?"
   A: Skip them by advancing pointers until we find alphanumeric characters, without modifying the original string.[5]
2. **Q:** "What about case sensitivity?"
   A: Convert characters to lowercase during comparison using `char.ToLower()` to make comparison case-insensitive.[6]
3. **Q:** "Can you solve this without extra space?"
   A: Yes, using two pointers eliminates need for creating a cleaned string, achieving **O(1)** space complexity.[5]
4. **Q:** "How would you handle Unicode characters?"
   A: Use `char.IsLetterOrDigit()` which handles Unicode properly, unlike manual ASCII range checking.
5. **Q:** "What if we need to return the cleaned palindrome string?"
   A: We'd need **O(n)** space to build the cleaned string, but the two-pointer validation approach remains the same.

Approach 1: Brute Force - Clean and Compare

Idea: Remove all non-alphanumeric characters, convert to lowercase, then compare string with its reverse.

Time Complexity: O(n) - string operations and comparison
Space Complexity: O(n) - for cleaned string

```
using System;
using System.Text;

public class Solution
{
    public bool IsPalindrome(string s)
    {
        // Clean the string - remove non-alphanumeric and convert to lowercase
        var cleaned = new StringBuilder();
        foreach (char c in s)
        {
            if (char.IsLetterOrDigit(c))
            {
                cleaned.Append(char.ToLower(c));
            }
        }

        string cleanedStr = cleaned.ToString();

        // Compare string with its reverse
        for (int i = 0; i < cleanedStr.Length / 2; i++)
        {
            if (cleanedStr[i] != cleanedStr[cleanedStr.Length - 1 - i])
            {
                return false;
            }
        }

        return true;
    }
}
```

Approach 2: Two Pointers with Helper Method

Explanation: Use two pointers but extract character cleaning logic into helper methods for better code organization.

Time Complexity: O(n) - single pass through string
Space Complexity: O(1) - no extra space needed

```
using System;

public class Solution
{
    public bool IsPalindrome(string s)
    {
        int left = 0, right = s.Length - 1;

        while (left < right)
        {
            // Skip non-alphanumeric characters from left
            while (left < right && !IsAlphaNumeric(s[left]))
            {
                left++;
            }

            // Skip non-alphanumeric characters from right
            while (left < right && !IsAlphaNumeric(s[right]))
            {
                right--;
            }

            // Compare characters (case-insensitive)
            if (ToLowerCase(s[left]) != ToLowerCase(s[right]))
            {
                return false;
            }

            left++;
            right--;
        }

        return true;
    }

    private bool IsAlphaNumeric(char c)
    {
        return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9');
    }

    private char ToLowerCase(char c)
    {
        if (c >= 'A' && c <= 'Z')
        {
            return (char)(c + 32);
        }
        return c;
    }
}
```

Approach 3: Optimal - Two Pointers

Detailed Reasoning: Use two pointers starting from opposite ends, skip non-alphanumeric characters, and compare valid characters case-insensitively. This is optimal because it processes the string in a single pass without using extra space.[5]

Time Complexity: O(n) - each character visited at most once
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    /// <summary>
    /// Validates palindrome using two pointers technique
    /// Time: O(n), Space: O(1)
    /// </summary>
    public bool IsPalindrome(string s)
    {
        // Initialize pointers at opposite ends
        int left = 0;
        int right = s.Length - 1;

        while (left < right)
        {
            // Skip non-alphanumeric characters from left
            if (!char.IsLetterOrDigit(s[left]))
            {
                left++;
                continue;
            }

            // Skip non-alphanumeric characters from right
            if (!char.IsLetterOrDigit(s[right]))
            {
                right--;
                continue;
            }

            // Compare characters (case-insensitive)
            if (char.ToLower(s[left]) != char.ToLower(s[right]))
            {
                return false;
            }

            // Move both pointers inward
            left++;
            right--;
        }

        return true;
    }
}
```

Key Takeaways:

- **Two pointers pattern**: Move from opposite ends toward center
- **Skip invalid characters**: Use continue to handle non-alphanumeric efficiently
- **Case-insensitive comparison**: Use built-in `char.ToLower()` for reliability

PROBLEM 2: BEST TIME TO BUY AND SELL STOCK

Problem Statement:

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`.

Test Cases:

```
Example 1:
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Example 2:
Input: prices = [7,6,4,3,2,1]
Output: 0
Explanation: Prices only decrease, so no profit can be made.

Example 3:
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
```

Interview Questions & Answers:

1. **Q:** "Can you buy and sell on the same day?"
   A: No, you must buy before you sell. The selling day must be after the buying day.[7]
2. **Q:** "What if all prices are decreasing?"
   A: Return 0 since no profit can be made. Never sell at a loss.[8]
3. **Q:** "Can you make multiple transactions?"
   A: This problem allows only one transaction (one buy + one sell). Multiple transactions would be a different variant.
4. **Q:** "How do you handle empty or single-element arrays?"
   A: Return 0 since you need at least two days to buy and sell.
5. **Q:** "What's the key insight for optimization?"
   A: Track the minimum price seen so far, and for each day, calculate profit if we sell that day.[7]

Approach 1: Brute Force - Check All Pairs

Idea: For each day, check all future days to find maximum profit from that buying day.

Time Complexity: O(n²) - nested loops to check all pairs
Space Complexity: O(1) - no extra space needed

```
using System;

public class Solution
{
    public int MaxProfit(int[] prices)
    {
        if (prices.Length < 2) return 0;

        int maxProfit = 0;

        // Try buying on each day
        for (int buyDay = 0; buyDay < prices.Length - 1; buyDay++)
        {
            // Try selling on each future day
            for (int sellDay = buyDay + 1; sellDay < prices.Length; sellDay++)
            {
                int profit = prices[sellDay] - prices[buyDay];
                maxProfit = Math.Max(maxProfit, profit);
            }
        }

        return maxProfit;
    }
}
```

Approach 2: Track Min Price and Max Profit

Explanation: Keep track of minimum price seen so far and maximum profit achievable. For each price, calculate profit if we sell today.

Time Complexity: O(n) - single pass through array
Space Complexity: O(1) - only using two variables

```
using System;

public class Solution
{
    public int MaxProfit(int[] prices)
    {
        if (prices.Length < 2) return 0;

        int minPrice = prices[^0];
        int maxProfit = 0;

        for (int i = 1; i < prices.Length; i++)
        {
            // Calculate profit if we sell today
            int profit = prices[i] - minPrice;
            maxProfit = Math.Max(maxProfit, profit);

            // Update minimum price for future calculations
            minPrice = Math.Min(minPrice, prices[i]);
        }

        return maxProfit;
    }
}
```

Approach 3: Optimal - Single Pass

Detailed Reasoning: This problem can be viewed as finding the maximum difference between two elements where the larger element comes after the smaller one. We maintain the minimum price seen so far and calculate profit for each potential selling day.[7]

Time Complexity: O(n) - single pass through array
Space Complexity: O(1) - constant extra space

```
using System;

public class Solution
{
    /// <summary>
    /// Finds maximum profit from single stock transaction
    /// Time: O(n), Space: O(1)
    /// </summary>
    public int MaxProfit(int[] prices)
    {
        // Edge case: need at least 2 days to trade
        if (prices == null || prices.Length < 2) return 0;

        int minPrice = int.MaxValue;
        int maxProfit = 0;

        foreach (int price in prices)
        {
            // If current price is lower than our minimum, update minimum
            if (price < minPrice)
            {
                minPrice = price;
            }
            // Otherwise, calculate profit if we sell at current price
            else if (price - minPrice > maxProfit)
            {
                maxProfit = price - minPrice;
            }
        }

        return maxProfit;
```

```
    }
}
```

Key Takeaways:

- **Greedy approach**: Always track the minimum buying price
- **Running maximum**: Update maximum profit as we process each day
- **Single pass optimization**: Combine finding minimum and calculating profit

PROBLEM 3: LONGEST SUBSTRING WITHOUT REPEATING CHARACTERS

Problem Statement:

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

Test Cases:

```
Example 1:
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

Example 2:
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.

Example 3:
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
```

Interview Questions & Answers:

1. **Q:** "What characters can the string contain?"
   A: The string can contain any ASCII characters including letters, digits, symbols, and spaces.
2. **Q:** "How do you detect repeating characters efficiently?"
   A: Use a HashSet to track characters in current window. When we encounter a character already in the set, we know we have a repeat.[9]
3. **Q:** "What's the sliding window approach?"
   A: Expand window by moving right pointer. When duplicate found, contract window by moving left pointer until duplicate is removed.[10] [9]
4. **Q:** "Can you optimize using character indices?"
   A: Yes, instead of moving left pointer one by one, jump directly to the position after the previous occurrence of the repeating character.
5. **Q:** "How do you handle empty strings?"
   A: Return 0 for empty strings since there are no substrings to consider.

Approach 1: Brute Force - Check All Substrings

Idea: Generate all possible substrings and check each one for uniqueness of characters.

Time Complexity: $O(n^3)$ - $O(n^2)$ substrings × $O(n)$ uniqueness check
Space Complexity: O(n) - for character set in worst case

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int LengthOfLongestSubstring(string s)
    {
        int maxLength = 0;

        // Check all possible substrings
        for (int i = 0; i < s.Length; i++)
        {
            for (int j = i; j < s.Length; j++)
            {
                // Check if substring from i to j has unique characters
                if (HasUniqueCharacters(s, i, j))
                {
                    maxLength = Math.Max(maxLength, j - i + 1);
                }
            }
        }

        return maxLength;
    }

    private bool HasUniqueCharacters(string s, int start, int end)
    {
        var charSet = new HashSet<char>();

        for (int i = start; i <= end; i++)
        {
            if (charSet.Contains(s[i]))
            {
                return false;
            }
            charSet.Add(s[i]);
        }

        return true;
    }
}
```

Approach 2: Sliding Window with HashSet

Explanation: Use sliding window with HashSet to track characters. Expand window until duplicate found, then contract from left until duplicate removed.[9]

Time Complexity: O(n) - each character visited at most twice
Space Complexity: O(min(m,n)) where m is character set size

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int LengthOfLongestSubstring(string s)
    {
        var charSet = new HashSet<char>();
        int left = 0;
        int maxLength = 0;

        for (int right = 0; right < s.Length; right++)
        {
            // If character is already in window, shrink from left
            while (charSet.Contains(s[right]))
            {
                charSet.Remove(s[left]);
                left++;
            }

            // Add current character and update max length
            charSet.Add(s[right]);
            maxLength = Math.Max(maxLength, right - left + 1);
        }

        return maxLength;
    }
}
```

Approach 3: Optimal - Sliding Window with HashMap

Detailed Reasoning: Instead of moving left pointer one by one, use HashMap to store last seen index of each character. When duplicate found, jump left pointer directly to position after previous occurrence.[9]

Time Complexity: O(n) - each character visited exactly once
Space Complexity: O(min(m,n)) - HashMap size bounded by character set

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Finds longest substring without repeating characters using optimized sliding wind
    /// Time: O(n), Space: O(min(m,n))
    /// </summary>
    public int LengthOfLongestSubstring(string s)
    {
        // Map to store last seen index of each character
        var charToIndex = new Dictionary<char, int>();
        int left = 0;
        int maxLength = 0;

        for (int right = 0; right < s.Length; right++)
        {
            char currentChar = s[right];

            // If character was seen before and is within current window
            if (charToIndex.ContainsKey(currentChar) && charToIndex[currentChar] >= left)
            {
                // Jump left pointer to position after previous occurrence
                left = charToIndex[currentChar] + 1;
            }

            // Update last seen index of current character
            charToIndex[currentChar] = right;

            // Update maximum length found so far
            maxLength = Math.Max(maxLength, right - left + 1);
        }

        return maxLength;
    }
}
```

Key Takeaways:

- **Sliding window pattern**: Expand right, contract left when needed
- **HashMap optimization**: Jump left pointer instead of incremental movement
- **Index tracking**: Store positions to enable efficient window adjustment

PROBLEM 4: LONGEST REPEATING CHARACTER REPLACEMENT

Problem Statement:

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

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

Test Cases:

```
Example 1:
Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with two 'B's or vice versa.

Example 2:
Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBBA". The substring

Example 3:
Input: s = "AAAA", k = 0
Output: 4
Explanation: All characters are already the same.
```

Interview Questions & Answers:

1. **Q:** "How do you determine if a window is valid?"
   A: A window is valid if `window_length - max_frequency <= k`, meaning we can change at most k characters to make all characters the same.[11] [12]
2. **Q:** "Which character should we choose to repeat?"
   A: Always choose the character with maximum frequency in current window, as it minimizes the number of changes needed.[11]
3. **Q:** "What happens when window becomes invalid?"
   A: Shrink window from left until it becomes valid again, updating character frequencies accordingly.
4. **Q:** "Can you solve this without shrinking the window?"
   A: Yes, using "lazy window" approach where window size only increases or stays same, making implementation simpler.[12]
5. **Q:** "How do you handle the character frequency tracking?"
   A: Use an array of size 26 for uppercase English letters, or HashMap for general characters.

Approach 1: Brute Force - Check All Substrings

Idea: For each possible substring, calculate minimum changes needed and check if it's within k limit.

Time Complexity: O(n² × m) where m is character set size
Space Complexity: O(m) for frequency counting

```
using System;

public class Solution
{
    public int CharacterReplacement(string s, int k)
    {
        int maxLength = 0;

        // Check all possible substrings
        for (int i = 0; i < s.Length; i++)
        {
            int[] freq = new int[^26];
            int maxFreq = 0;

            for (int j = i; j < s.Length; j++)
            {
                // Update frequency of current character
                freq[s[j] - 'A']++;
                maxFreq = Math.Max(maxFreq, freq[s[j] - 'A']);

                // Calculate changes needed for this window
                int windowLength = j - i + 1;
                int changesNeeded = windowLength - maxFreq;

                if (changesNeeded <= k)
                {
                    maxLength = Math.Max(maxLength, windowLength);
                }
            }
        }

        return maxLength;
    }
}
```

Approach 2: Sliding Window with Window Shrinking

Explanation: Use sliding window that expands and contracts. When window becomes invalid (needs more than k changes), shrink from left until valid.[12]

Time Complexity: O(n) - each character enters and exits window at most once
Space Complexity: O(1) - fixed size frequency array

```
using System;

public class Solution
{
    public int CharacterReplacement(string s, int k)
    {
        int[] freq = new int[^26];
        int left = 0;
        int maxFreq = 0;
        int maxLength = 0;

        for (int right = 0; right < s.Length; right++)
        {
            // Expand window - add right character
            freq[s[right] - 'A']++;
            maxFreq = Math.Max(maxFreq, freq[s[right] - 'A']);
```

```
            // Check if window is valid
            int windowLength = right - left + 1;
            int changesNeeded = windowLength - maxFreq;

            // Shrink window if invalid
            while (changesNeeded > k)
            {
                freq[s[left] - 'A']--;
                left++;

                // Recalculate max frequency after shrinking
                maxFreq = 0;
                for (int i = 0; i < 26; i++)
                {
                    maxFreq = Math.Max(maxFreq, freq[i]);
                }

                windowLength = right - left + 1;
                changesNeeded = windowLength - maxFreq;
            }

            maxLength = Math.Max(maxLength, windowLength);
        }

        return maxLength;
    }
}
```

Approach 3: Optimal - Lazy Sliding Window

Detailed Reasoning: Use "lazy window" approach where window size never decreases. When window becomes invalid, move both pointers right. This works because we only care about the maximum length, not all valid windows.[12]

Time Complexity: O(n) - single pass with constant operations
Space Complexity: O(1) - fixed size frequency array

```
using System;

public class Solution
{
    /// <summary>
    /// Finds longest repeating character replacement using lazy sliding window
    /// Time: O(n), Space: O(1)
    /// </summary>
    public int CharacterReplacement(string s, int k)
    {
        int[] freq = new int[^26];
        int left = 0;
        int maxFreq = 0;
        int maxLength = 0;

        for (int right = 0; right < s.Length; right++)
        {
            // Expand window - add right character
            freq[s[right] - 'A']++;
            maxFreq = Math.Max(maxFreq, freq[s[right] - 'A']);

            // Calculate current window length
            int windowLength = right - left + 1;

            // If window is invalid, slide it (don't shrink)
            if (windowLength - maxFreq > k)
            {
                freq[s[left] - 'A']--;
                left++;
            }

            // Update maximum length found
            // Note: window length is now right - left + 1
            maxLength = Math.Max(maxLength, right - left + 1);
        }

        return maxLength;
    }
}
```

Key Takeaways:

- **Window validity condition**: `length - max_frequency <= k`
- **Lazy window technique**: Window size never decreases, simplifying logic
- **Frequency tracking**: Maintain running count of characters in current window

PROBLEM 5: 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 window in `s` that covers all characters in `t`, return the empty string `""`.

The testcases will be generated such that the answer is unique.

Test Cases:

```
Example 1:
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t

Example 2:
Input: s = "a", t = "a"
Output: "a"
Explanation: The entire string s is the minimum window.
```

```
Example 3:
Input: s = "a", t = "aa"
Output: ""
Explanation: Both 'a's from t must be included in the window. Since the largest window of
```

Interview Questions & Answers:

1. **Q:** "How do you handle duplicate characters in t?"
   A: Count frequency of each character in t. Window must contain at least that many of each character.[13]
2. **Q:** "What's the expansion and contraction strategy?"
   A: Expand window until all characters from t are covered, then contract from left while maintaining coverage to find minimum window.[13]
3. **Q:** "How do you efficiently check if window covers t?"
   A: Track count of satisfied characters. When count equals length of t, we have a valid window.
4. **Q:** "What if no valid window exists?"
   A: Return empty string. This happens when s doesn't contain all characters needed for t.
5. **Q:** "How do you optimize the window search?"
   A: Use two pointers and character frequency maps. Only expand when needed, contract greedily when possible.

Approach 1: Brute Force - Check All Substrings

Idea: Generate all possible substrings of s and check which ones contain all characters from t, then find the shortest.

Time Complexity: O(n² × m) where n = length of s, m = length of t
Space Complexity: O(m) for character frequency maps

```
using System;
using System.Collections.Generic;

public class Solution
{
    public string MinWindow(string s, string t)
    {
        if (s.Length < t.Length) return "";

        // Count characters in t
        var need = new Dictionary<char, int>();
        foreach (char c in t)
        {
            need[c] = need.GetValueOrDefault(c, 0) + 1;
        }

        string minWindow = "";
        int minLength = int.MaxValue;

        // Check all substrings
        for (int i = 0; i < s.Length; i++)
        {
            for (int j = i; j < s.Length; j++)
            {
                if (ContainsAllChars(s.Substring(i, j - i + 1), need))
                {
                    if (j - i + 1 < minLength)
                    {
                        minLength = j - i + 1;
                        minWindow = s.Substring(i, j - i + 1);
                    }
                }
            }
        }

        return minWindow;
    }

    private bool ContainsAllChars(string window, Dictionary<char, int> need)
    {
        var have = new Dictionary<char, int>();
        foreach (char c in window)
        {
            have[c] = have.GetValueOrDefault(c, 0) + 1;
        }

        foreach (var pair in need)
        {
            if (!have.ContainsKey(pair.Key) || have[pair.Key] < pair.Value)
            {
                return false;
            }
        }
        return true;
    }
}
```

Approach 2: Sliding Window with HashMap

Explanation: Use sliding window with two HashMaps to track required characters and current window characters.[13]

Time Complexity: O(n + m) - each character in s visited at most twice
Space Complexity: O(m) - for character frequency maps

```
using System;
using System.Collections.Generic;

public class Solution
{
    public string MinWindow(string s, string t)
    {
        if (s.Length < t.Length) return "";
```

```
        // Count characters needed from t
        var need = new Dictionary<char, int>();
        foreach (char c in t)
        {
            need[c] = need.GetValueOrDefault(c, 0) + 1;
        }

        var window = new Dictionary<char, int>();
        int left = 0, right = 0;
        int formed = 0; // Number of unique chars in window with desired frequency
        int required = need.Count;

        // Result tracking
        int minLength = int.MaxValue;
        int minLeft = 0;

        while (right < s.Length)
        {
            // Expand window
            char c = s[right];
            window[c] = window.GetValueOrDefault(c, 0) + 1;

            if (need.ContainsKey(c) && window[c] == need[c])
            {
                formed++;
            }

            // Contract window
            while (left <= right && formed == required)
            {
                // Update result if current window is smaller
                if (right - left + 1 < minLength)
                {
                    minLength = right - left + 1;
                    minLeft = left;
                }

                // Remove left character
                char leftChar = s[left];
                window[leftChar]--;
                if (need.ContainsKey(leftChar) && window[leftChar] < need[leftChar])
                {
                    formed--;
                }
                left++;
            }

            right++;
        }

        return minLength == int.MaxValue ? "" : s.Substring(minLeft, minLength);
    }
}
```

Approach 3: Optimal - Optimized Sliding Window

Detailed Reasoning: Further optimize by only considering characters that appear in t, reducing the effective length of s for processing.[13]

Time Complexity: O(n + m) - linear time complexity
Space Complexity: O(m) - space for frequency maps

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Finds minimum window substring using optimized sliding window
    /// Time: O(n + m), Space: O(m)
    /// </summary>
    public string MinWindow(string s, string t)
    {
        if (s.Length == 0 || t.Length == 0 || s.Length < t.Length) return "";

        // Character frequency map for t
        var need = new Dictionary<char, int>();
        foreach (char c in t)
        {
            need[c] = need.GetValueOrDefault(c, 0) + 1;
        }

        int required = need.Count;
        int left = 0, right = 0;
        int formed = 0;

        // Character frequency map for current window
        var windowCounts = new Dictionary<char, int>();

        // Result: [window length, left, right]
        int[] result = {-1, 0, 0};

        while (right < s.Length)
        {
            // Add character from right to window
            char rightChar = s[right];
            windowCounts[rightChar] = windowCounts.GetValueOrDefault(rightChar, 0) + 1;

            // Check if current character's frequency matches requirement
            if (need.ContainsKey(rightChar) &&
                windowCounts[rightChar] == need[rightChar])
            {
                formed++;
            }

            // Try to contract window until it's no longer valid
            while (left <= right && formed == required)
            {
                rightChar = s[right];
```

```
                // Update result if this window is smaller
                if (result[^0] == -1 || right - left + 1 < result[^0])
                {
                    result[^0] = right - left + 1;
                    result[^1] = left;
                    result[^2] = right;
                }

                // Character at left pointer
                char leftChar = s[left];
                windowCounts[leftChar]--;

                if (need.ContainsKey(leftChar) &&
                    windowCounts[leftChar] < need[leftChar])
                {
                    formed--;
                }

                left++;
            }

            right++;
        }

        return result[^0] == -1 ? "" : s.Substring(result[^1], result[^0]);
    }
}
```

Key Takeaways:

- **Character frequency matching**: Track when window satisfies all requirements
- **Greedy contraction**: Once valid window found, contract greedily to minimize
- **Efficient validation**: Use counter to avoid recomputing window validity

PROBLEM 6: CONTAINER WITH MOST WATER

Problem Statement:

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

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

Return the maximum amount of water a container can store.

Note: You may not slant the container.

Test Cases:

```
Example 1:
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: Heights at indices 1 and 8 (values 8 and 7) form container with area = min(8

Example 2:
Input: height = [1,1]
Output: 1
Explanation: Two lines of height 1 form container with area = min(1,1) * (1-0) = 1.

Example 3:
Input: height = [4,3,2,1,4]
Output: 16
Explanation: Heights at indices 0 and 4 (values 4 and 4) form container with area = min(4
```

Interview Questions & Answers:

1. **Q:** "How do you calculate the area of water between two lines?"
   A: Area = `min(height[left], height[right]) * (right - left)`. Height is limited by the shorter line.[14]
2. **Q:** "Why does the two-pointer approach work?"
   A: Moving the pointer with smaller height gives us the only chance to find a larger area, since keeping it would only decrease width without increasing height.[14]
3. **Q:** "What if there are multiple optimal solutions?"
   A: Problem asks for maximum area value, not the indices, so any pair giving maximum area is correct.
4. **Q:** "How do you handle edge cases?"
   A: Need at least 2 lines to form container. For arrays with length < 2, return 0.
5. **Q:** "Can you prove the two-pointer approach doesn't miss the optimal solution?"
   A: When we move the shorter pointer, we eliminate all possible containers with that pointer and smaller width, none of which could be better than current.

Approach 1: Brute Force - Check All Pairs

Idea: Check every possible pair of lines and calculate water area for each pair.

Time Complexity: O(n^2) - nested loops to check all pairs
Space Complexity: O(1) - no extra space needed

```
using System;

public class Solution
{
    public int MaxArea(int[] height)
    {
        int maxWater = 0;
```

```
        int n = height.Length;

        // Check all possible pairs of lines
        for (int i = 0; i < n - 1; i++)
        {
            for (int j = i + 1; j < n; j++)
            {
                // Calculate area between lines i and j
                int width = j - i;
                int minHeight = Math.Min(height[i], height[j]);
                int area = width * minHeight;

                maxWater = Math.Max(maxWater, area);
            }
        }

        return maxWater;
    }
}
```

Approach 2: Two Pointers with Area Tracking

Explanation: Use two pointers from opposite ends. Always move the pointer with smaller height inward, since that's the only way to potentially increase area.

Time Complexity: O(n) - single pass with two pointers
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    public int MaxArea(int[] height)
    {
        int left = 0;
        int right = height.Length - 1;
        int maxWater = 0;

        while (left < right)
        {
            // Calculate current area
            int width = right - left;
            int minHeight = Math.Min(height[left], height[right]);
            int currentArea = width * minHeight;

            maxWater = Math.Max(maxWater, currentArea);

            // Move pointer with smaller height
            if (height[left] < height[right])
            {
                left++;
            }
            else
            {
                right--;
            }
        }

        return maxWater;
    }
}
```

Approach 3: Optimal - Two Pointers

Detailed Reasoning: The two-pointer approach is optimal because moving the pointer with larger height inward can never lead to a larger area (width decreases, height can't increase). Only moving the smaller height pointer gives us a chance to find larger area.[14]

Time Complexity: O(n) - each element visited exactly once
Space Complexity: O(1) - constant extra space

```
using System;

public class Solution
{
    /// <summary>
    /// Finds maximum water container area using two pointers
    /// Time: O(n), Space: O(1)
    /// </summary>
    public int MaxArea(int[] height)
    {
        // Edge case: need at least 2 lines
        if (height == null || height.Length < 2) return 0;

        int left = 0;
        int right = height.Length - 1;
        int maxArea = 0;

        while (left < right)
        {
            // Calculate area with current left and right pointers
            int width = right - left;
            int currentHeight = Math.Min(height[left], height[right]);
            int currentArea = width * currentHeight;

            // Update maximum area found so far
            maxArea = Math.Max(maxArea, currentArea);

            // Move the pointer with smaller height
            // This is the key insight: moving the larger height pointer
            // can never give us a better result (width decreases, height can't increase)
            if (height[left] < height[right])
            {
                left++;
            }
            else
            {
                right--;
```

```
            }
        }

        return maxArea;
    }
}
```

Key Takeaways:

- **Area calculation**: `width × min(height1, height2)`
- **Greedy pointer movement**: Always move pointer with smaller height
- **Proof of optimality**: Moving larger height pointer never improves solution

PROBLEM 7: 3SUM

Problem Statement:

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

Notice that the solution set must not contain duplicate triplets.

Test Cases:

```
Example 1:
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[^0] + nums[^1] + nums[^2] = (-1) + 0 + 1 = 0.
nums[^1] + nums[^2] + nums[^4] = 0 + 1 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].

Example 2:
Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.

Example 3:
Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: The only possible triplet sums up to 0.
```

Interview Questions & Answers:

1. **Q:** "How do you avoid duplicate triplets?"
   A: Sort array first, then skip duplicate values when iterating through potential first elements and when using two pointers.
2. **Q:** "Why sort the array first?"
   A: Sorting enables two-pointer technique for the remaining two elements and makes duplicate skipping easier with simple index advancement.
3. **Q:** "What's the relationship to 2Sum?"
   A: 3Sum reduces to 2Sum: fix first element, then find two elements in remaining array that sum to `-first_element`.
4. **Q:** "How do you handle the three-way duplicate skipping?"
   A: Skip duplicates at three levels: first element (i), second element (left pointer), and third element (right pointer).
5. **Q:** "Can you optimize further than O(n²)?"
   A: No, there's no known algorithm better than O(n²) for 3Sum in the general case.

Approach 1: Brute Force - Three Nested Loops

Idea: Use three nested loops to check all possible triplets and collect those that sum to zero.

Time Complexity: O(n³) - three nested loops
Space Complexity: O(1) excluding output space

```
using System;
using System.Collections.Generic;

public class Solution
{
    public IList<IList<int>> ThreeSum(int[] nums)
    {
        var result = new List<IList<int>>();
        var seen = new HashSet<string>();

        // Check all possible triplets
        for (int i = 0; i < nums.Length - 2; i++)
        {
            for (int j = i + 1; j < nums.Length - 1; j++)
            {
                for (int k = j + 1; k < nums.Length; k++)
                {
                    if (nums[i] + nums[j] + nums[k] == 0)
                    {
                        // Create sorted triplet to avoid duplicates
                        var triplet = new List<int> { nums[i], nums[j], nums[k] };
                        triplet.Sort();

                        string key = $"{triplet[^0]},{triplet[^1]},{triplet[^2]}";
                        if (!seen.Contains(key))
                        {
                            seen.Add(key);
                            result.Add(new List<int>(triplet));
                        }
                    }
                }
            }
        }

        return result;
```

```
    }
}
```

Approach 2: Sorting + HashMap

Explanation: Sort array, fix first element, then use HashMap for 2Sum on remaining elements.

Time Complexity: O(n²) - sorting + O(n²) for nested loops
Space Complexity: O(n) - for HashMap

```
using System;
using System.Collections.Generic;

public class Solution
{
    public IList<IList<int>> ThreeSum(int[] nums)
    {
        var result = new List<IList<int>>();
        Array.Sort(nums);

        for (int i = 0; i < nums.Length - 2; i++)
        {
            // Skip duplicates for first element
            if (i > 0 && nums[i] == nums[i - 1]) continue;

            int target = -nums[i];
            var seen = new HashSet<int>();

            for (int j = i + 1; j < nums.Length; j++)
            {
                int complement = target - nums[j];

                if (seen.Contains(complement))
                {
                    result.Add(new List<int> { nums[i], complement, nums[j] });

                    // Skip duplicates for second element
                    while (j + 1 < nums.Length && nums[j] == nums[j + 1]) j++;
                }

                seen.Add(nums[j]);
            }
        }

        return result;
    }
}
```

Approach 3: Optimal - Sorting + Two Pointers

Detailed Reasoning: Sort array, then for each potential first element, use two pointers to find pairs that complete the triplet. This is optimal because it achieves O(n²) time with O(1) extra space.

Time Complexity: O(n²) - O(n log n) sorting + O(n²) for main algorithm
Space Complexity: O(1) excluding output space

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Finds all unique triplets that sum to zero using sorting and two pointers
    /// Time: O(n²), Space: O(1)
    /// </summary>
    public IList<IList<int>> ThreeSum(int[] nums)
    {
        var result = new List<IList<int>>();

        // Edge case: need at least 3 elements
        if (nums == null || nums.Length < 3) return result;

        // Sort array to enable two-pointer technique
        Array.Sort(nums);

        for (int i = 0; i < nums.Length - 2; i++)
        {
            // Skip positive numbers as first element (impossible to sum to 0)
            if (nums[i] > 0) break;

            // Skip duplicates for first element
            if (i > 0 && nums[i] == nums[i - 1]) continue;

            // Use two pointers for remaining elements
            int left = i + 1;
            int right = nums.Length - 1;
            int target = -nums[i];

            while (left < right)
            {
                int currentSum = nums[left] + nums[right];

                if (currentSum == target)
                {
                    // Found valid triplet
                    result.Add(new List<int> { nums[i], nums[left], nums[right] });

                    // Skip duplicates for second element
                    while (left < right && nums[left] == nums[left + 1]) left++;

                    // Skip duplicates for third element
                    while (left < right && nums[right] == nums[right - 1]) right--;
```

```
                    // Move both pointers
                    left++;
                    right--;
                }
                else if (currentSum < target)
                {
                    left++;  // Need larger sum
                }
                else
                {
                    right--; // Need smaller sum
                }
            }
        }

        return result;
    }
}
```

Key Takeaways:

- **Reduce to 2Sum**: Fix first element, solve 2Sum for remaining elements
- **Sorting enables optimization**: Two pointers work on sorted arrays
- **Triple duplicate handling**: Skip duplicates at all three levels

PROBLEM 8: TRAPPING RAIN WATER

Problem Statement:

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

Test Cases:

```
Example 1:
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The elevation map (black) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1].

Example 2:
Input: height = [4,2,0,3,2,5]
Output: 9
Explanation: Water trapped above each position sums to 9 units.

Example 3:
Input: height = [1,0,2]
Output: 1
Explanation: 1 unit of water can be trapped between heights 1 and 2.
```

Interview Questions & Answers:

1. **Q:** "How do you calculate water at each position?"
   A: Water at position i = `min(max_left, max_right) - height[i]`, but only if this value is positive.
2. **Q:** "Why does the two-pointer approach work?"
   A: We can determine water level at current position when we know the smaller of max_left and max_right, since that's the limiting factor.
3. **Q:** "What's the key insight for optimization?"
   A: Instead of precomputing all left and right maximums, use two pointers and track maximums dynamically.
4. **Q:** "How do you handle edge cases?"
   A: Arrays with length < 3 cannot trap water. Also, water amount is never negative.
5. **Q:** "Can you solve this with constant space?"
   A: Yes, two-pointer approach uses O(1) space compared to O(n) for the precomputation approach.

Approach 1: Brute Force - For Each Position

Idea: For each position, find maximum height to its left and right, then calculate trapped water.

Time Complexity: O(n²) - for each position, scan left and right
Space Complexity: O(1) - no extra space needed

```
using System;

public class Solution
{
    public int Trap(int[] height)
    {
        if (height.Length < 3) return 0;

        int totalWater = 0;

        // For each position, calculate trapped water
        for (int i = 1; i < height.Length - 1; i++)
        {
            // Find maximum height to the left
            int leftMax = 0;
            for (int j = 0; j < i; j++)
            {
                leftMax = Math.Max(leftMax, height[j]);
            }

            // Find maximum height to the right
            int rightMax = 0;
            for (int j = i + 1; j < height.Length; j++)
            {
                rightMax = Math.Max(rightMax, height[j]);
            }
```

```
            // Calculate water at current position
            int waterLevel = Math.Min(leftMax, rightMax);
            if (waterLevel > height[i])
            {
                totalWater += waterLevel - height[i];
            }
        }

        return totalWater;
    }
}
```

Approach 2: Precompute Left and Right Maximums

Explanation: Precompute maximum heights to left and right of each position, then calculate trapped water in single pass.

Time Complexity: O(n) - three separate passes through array
Space Complexity: O(n) - for left and right maximum arrays

```
using System;

public class Solution
{
    public int Trap(int[] height)
    {
        if (height.Length < 3) return 0;

        int n = height.Length;
        int[] leftMax = new int[n];
        int[] rightMax = new int[n];

        // Precompute left maximums
        leftMax[^0] = height[^0];
        for (int i = 1; i < n; i++)
        {
            leftMax[i] = Math.Max(leftMax[i - 1], height[i]);
        }

        // Precompute right maximums
        rightMax[n - 1] = height[n - 1];
        for (int i = n - 2; i >= 0; i--)
        {
            rightMax[i] = Math.Max(rightMax[i + 1], height[i]);
        }

        // Calculate trapped water
        int totalWater = 0;
        for (int i = 0; i < n; i++)
        {
            int waterLevel = Math.Min(leftMax[i], rightMax[i]);
            if (waterLevel > height[i])
            {
                totalWater += waterLevel - height[i];
            }
        }

        return totalWater;
    }
}
```

Approach 3: Optimal - Two Pointers

Detailed Reasoning: Use two pointers moving from opposite ends. At each step, we know the water level is determined by the smaller of the two maximum heights encountered so far. This eliminates need for precomputation while maintaining linear time.

Time Complexity: O(n) - single pass through array
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    /// <summary>
    /// Calculates trapped rainwater using two pointers technique
    /// Time: O(n), Space: O(1)
    /// </summary>
    public int Trap(int[] height)
    {
        // Edge case: need at least 3 bars to trap water
        if (height == null || height.Length < 3) return 0;

        int left = 0;
        int right = height.Length - 1;
        int leftMax = 0;
        int rightMax = 0;
        int totalWater = 0;

        while (left < right)
        {
            if (height[left] < height[right])
            {
                // Process left side
                if (height[left] >= leftMax)
                {
                    leftMax = height[left];  // Update left maximum
                }
                else
                {
                    // Water can be trapped here
                    totalWater += leftMax - height[left];
                }
                left++;
            }
            else
            {
```

```
                // Process right side
                if (height[right] >= rightMax)
                {
                    rightMax = height[right];  // Update right maximum
                }
                else
                {
                    // Water can be trapped here
                    totalWater += rightMax - height[right];
                }
                right--;
            }
        }

        return totalWater;
    }
}
```

Key Takeaways:

- **Water level calculation**: `min(left_max, right_max) - current_height`
- **Two-pointer optimization**: Process side with smaller current maximum
- **Dynamic maximum tracking**: Update maximums as we traverse

DAY 2 SUMMARY

Summary of Concepts Covered

Two Pointers Technique:

- **Opposite ends**: Start from array boundaries, move toward center
- **Fast/slow pointers**: Different speeds for cycle detection
- **Same direction**: Both pointers move forward with different conditions
- **Key advantage**: Reduces O(n²) to O(n) time complexity[1]

Sliding Window Technique:

- **Fixed window**: Constant size window for moving averages
- **Variable window**: Expand/contract based on conditions
- **Window operations**: Expand right, contract left, maintain state[3]
- **Optimization**: Each element enters/exits window at most once

String Processing Patterns:

- **Character validation**: Use built-in methods for robustness
- **Case handling**: Normalize case for comparisons
- **Frequency counting**: Arrays for limited sets, HashMap for general use

Time & Space Complexity Summary

Problem Type	Brute Force	Optimized	Key Technique
Palindrome Validation	O(n) space	**O(1)** space	Two pointers from ends
Stock Problem	**O(n²)** time	**O(n)** time	Track minimum price
Substring Problems	**O(n³)** time	**O(n)** time	Sliding window
Container Water	**O(n²)** time	**O(n)** time	Two pointers greedy
3Sum	**O(n³)** time	**O(n²)** time	Sort + two pointers
Rain Water	**O(n²)** time	**O(n)** time	Two pointers dynamic

Pattern Recognition Guide

Use Two Pointers When:

- Sorted array with target sum
- Palindrome or symmetry checking
- Finding pairs/triplets with conditions
- Container or area maximization problems

Use Sliding Window When:

- Substring or subarray problems
- Finding optimal window (min/max length)
- Character or element frequency tracking
- Problems involving "consecutive" elements

Review Tips

- **Practice pointer movement logic**: Understand when to move which pointer
- **Master window validity**: Learn to determine when window satisfies conditions
- **Optimize incrementally**: Start with brute force, then identify optimization opportunities
- **Handle edge cases**: Empty arrays, single elements, all same elements

How to Practice Mock Interviews

1. **Problem Analysis** (5 min): Identify if it's two pointers or sliding window
2. **Approach Discussion** (5 min): Explain brute force, then optimal approach
3. **Implementation** (20 min): Code while explaining pointer movement logic
4. **Complexity Analysis** (5 min): Analyze time/space complexity
5. **Testing** (10 min): Walk through examples and edge cases

Common Mistakes to Avoid

- **Infinite loops**: Ensure pointers always make progress toward termination
- **Boundary errors**: Check pointer bounds before accessing array elements
- **Duplicate handling**: Remember to skip duplicates in problems requiring unique results
- **Window state management**: Properly maintain frequency counts or sums in sliding window
- **Greedy assumptions**: Verify that greedy pointer movement doesn't miss optimal solutions

Advanced Variations to Explore

- **4Sum, k-Sum**: Extend 3Sum pattern to more elements
- **Sliding window maximum**: Use deque for O(1) maximum queries
- **Variable window with multiple conditions**: More complex validity functions
- **Two pointers on two arrays**: Merge sorted arrays, intersection problems

This completes Day 2: Two Pointers & Sliding Window with comprehensive coverage of all 8 problems, following the exact structure and educational depth you requested.

⁂

1. https://www.geeksforgeeks.org/dsa/two-pointers-technique/
2. https://takeuforward.org/data-structure/what-is-two-pointer-approach/
3. https://www.youtube.com/watch?v=dOonV4byDEg
4. https://www.linkedin.com/pulse/mastering-sliding-window-technique-practical-mohideen-risvi-y-zzgec
5. https://algomap.io/problems/valid-palindrome
6. https://github.com/zqfang/LeetCode/blob/master/Python/valid-palindrome.py
7. https://algo.monster/liteproblems/121
8. https://www.youtube.com/watch?v=kJZrMGpyWpk
9. https://algo.monster/liteproblems/3
10. https://www.geeksforgeeks.org/dsa/length-of-the-longest-substring-without-repeating-characters/
11. https://www.youtube.com/watch?v=tkNWKvxI3mU
12. https://walkccc.me/LeetCode/problems/424/
13. https://algo.monster/liteproblems/76
14. https://www.youtube.com/watch?v=Y_4_or0Sc7I