



## Generate for the rest of the missing problems similarly as a continuation

I'll continue generating the remaining problems (4-8) following the exact same detailed structure as the previous problems:

### PROBLEM 4: GROUP ANAGRAMS

#### Problem Statement:

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

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

#### Test Cases:

Example 1:

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

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

Explanation: "eat", "tea", "ate" are anagrams. "tan", "nat" are anagrams. "bat" stands alone.

Example 2:

Input: `strs = [""]`

Output: `[[""]]`

Explanation: Empty string forms its own group.

Example 3:

Input: `strs = ["a"]`

Output: `[["a"]]`

Explanation: Single character forms its own group.

#### Interview Questions & Answers:

- Q:** "How do you determine if two strings are anagrams?"  
**A:** Two strings are anagrams if they contain the same characters with the same frequencies. We can sort both strings or use character frequency counting.
- Q:** "What's the most efficient way to group them?"  
**A:** Use a HashMap where the key represents the anagram group (sorted string or frequency signature) and the value is a list of strings belonging to that group.

3. **Q:** "How would you handle very long strings?"

**A:** Sorting approach becomes expensive for long strings. Character frequency approach with array (for limited charset) or HashMap is better.

4. **Q:** "What if the input contains Unicode characters?"

**A:** Use HashMap for character frequency instead of fixed-size array, as Unicode has many more possible characters.

5. **Q:** "Can you optimize space complexity?"

**A:** We need to return all groups, so space complexity is inherently  $O(n)$  where  $n$  is total number of characters across all strings.

## Approach 1: Brute Force - Compare Every Pair

**Idea:** For each string, compare it with all other strings to find anagrams using character frequency comparison.

**Time Complexity:**  $O(n^2 \times m)$  where  $n$  = number of strings,  $m$  = average string length

**Space Complexity:**  $O(n \times m)$  for storing results

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

public class Solution
{
    public IList<IList<string>> GroupAnagrams(string[] strs)
    {
        var result = new List<IList<string>>();
        bool[] used = new bool[strs.Length];

        // Check each string against all others
        for (int i = 0; i < strs.Length; i++)
        {
            if (used[i]) continue;

            var group = new List<string> { strs[i] };
            used[i] = true;

            // Find all anagrams of current string
            for (int j = i + 1; j < strs.Length; j++)
            {
                if (!used[j] && AreAnagrams(strs[i], strs[j]))
                {
                    group.Add(strs[j]);
                    used[j] = true;
                }
            }

            result.Add(group);
        }

        return result;
    }
}
```

```

private bool AreAnagrams(string s1, string s2)
{
    if (s1.Length != s2.Length) return false;

    int[] count = new int[26];
    for (int i = 0; i < s1.Length; i++)
    {
        count[s1[i] - 'a']++;
        count[s2[i] - 'a']--;
    }

    foreach (int c in count)
    {
        if (c != 0) return false;
    }
    return true;
}
}

```

## Approach 2: Sorting as Key

**Explanation:** Use sorted version of each string as HashMap key. All anagrams will have the same sorted string.

**Time Complexity:**  $O(n \times m \log m)$  where  $n$  = number of strings,  $m$  = average string length

**Space Complexity:**  $O(n \times m)$  for HashMap and results

```

using System;
using System.Collections.Generic;
using System.Linq;

public class Solution
{
    public IList<IList<string>> GroupAnagrams(string[] strs)
    {
        var groups = new Dictionary<string, List<string>>();

        foreach (string str in strs)
        {
            // Create sorted key for this string
            char[] chars = str.ToCharArray();
            Array.Sort(chars);
            string key = new string(chars);

            // Add to appropriate group
            if (!groups.ContainsKey(key))
            {
                groups[key] = new List<string>();
            }
            groups[key].Add(str);
        }

        return groups.Values.ToList<IList<string>>();
    }
}

```

```
}  
}
```

### Approach 3: Optimal - Character Frequency as Key

**Detailed Reasoning:** Instead of sorting strings (expensive for long strings), use character frequency signature as key. This avoids the  $O(m \log m)$  sorting cost per string, achieving better time complexity especially for longer strings.

**Time Complexity:**  $O(n \times m)$  where  $n$  = number of strings,  $m$  = average string length

**Space Complexity:**  $O(n \times m)$  for HashMap and results

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
public class Solution  
{  
    /// <summary>  
    /// Groups anagrams using character frequency signature  
    /// Time:  $O(n \times m)$ , Space:  $O(n \times m)$   
    /// </summary>  
    public IList<IList<string>> GroupAnagrams(string[] strs)  
    {  
        var groups = new Dictionary<string, List<string>>();  
  
        foreach (string str in strs)  
        {  
            // Create frequency signature as key  
            string key = GetFrequencyKey(str);  
  
            // Group strings with same frequency signature  
            if (!groups.ContainsKey(key))  
            {  
                groups[key] = new List<string>();  
            }  
            groups[key].Add(str);  
        }  
  
        return groups.Values.ToList<IList<string>>();  
    }  
  
    private string GetFrequencyKey(string str)  
    {  
        // Count character frequencies  
        int[] count = new int[26];  
        foreach (char c in str)  
        {  
            count[c - 'a']++;  
        }  
  
        // Build frequency signature  
        var key = new StringBuilder();
```

```

        for (int i = 0; i < 26; i++)
        {
            if (count[i] > 0)
            {
                key.Append((char)('a' + i));
                key.Append(count[i]);
            }
        }

        return key.ToString();
    }
}

```

### Key Takeaways:

- **Signature-based grouping:** Create unique identifier for each anagram group
- **Avoid unnecessary sorting:** Character frequency is more efficient for long strings
- **HashMap for grouping:** Natural pattern for categorization problems

## PROBLEM 5: TOP K FREQUENT ELEMENTS

### Problem Statement:

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

### Test Cases:

Example 1:

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

Output: `[1,2]`

Explanation: 1 appears 3 times, 2 appears 2 times, 3 appears 1 time. Top 2 are `[1,2]`.

Example 2:

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

Output: `[1]`

Explanation: Only one element, so it's the most frequent.

Example 3:

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

Output: `[3,2]`

Explanation: 3 appears 3 times, 2 appears 2 times, 1 appears 1 time. Top 2 are `[3,2]`.

### Interview Questions & Answers:

1. **Q:** "What if there are ties in frequency?"

**A:** Problem allows returning answer in any order, so any elements with tied frequency can be included arbitrarily.

2. **Q:** "What's the difference between this and finding the most frequent element?"  
**A:** This asks for top K elements, requiring us to maintain K elements instead of just tracking the single maximum.
3. **Q:** "Can we use sorting?"  
**A:** Yes, but it's not optimal. We can sort by frequency and take first K elements, but heap-based approach is more efficient.
4. **Q:** "What if k is larger than the number of unique elements?"  
**A:** Return all unique elements since we can't return more elements than exist.
5. **Q:** "How would you optimize for very large arrays?"  
**A:** Use bucket sort or quickselect algorithm to avoid full sorting, achieving better average time complexity.

### Approach 1: Brute Force - Count and Sort All

**Idea:** Count frequency of all elements, sort by frequency, and take first k elements.

**Time Complexity:**  $O(n \log n)$  - dominated by sorting

**Space Complexity:**  $O(n)$  - for frequency map and result

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

public class Solution
{
    public int[] TopKFrequent(int[] nums, int k)
    {
        // Count frequencies
        var freqMap = new Dictionary<int, int>();
        foreach (int num in nums)
        {
            freqMap[num] = freqMap.GetValueOrDefault(num, 0) + 1;
        }

        // Sort by frequency and take top k
        var sorted = freqMap.OrderByDescending(x => x.Value)
            .Take(k)
            .Select(x => x.Key)
            .ToArray();

        return sorted;
    }
}
```

## Approach 2: Min Heap

**Explanation:** Use min heap of size k to keep track of k most frequent elements. This avoids sorting all elements.

**Time Complexity:**  $O(n \log k)$  - heap operations for each element

**Space Complexity:**  $O(n + k)$  - frequency map plus heap

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

public class Solution
{
    public int[] TopKFrequent(int[] nums, int k)
    {
        // Count frequencies
        var freqMap = new Dictionary<int, int>();
        foreach (int num in nums)
        {
            freqMap[num] = freqMap.GetValueOrDefault(num, 0) + 1;
        }

        // Use min heap to maintain top k elements
        var heap = new SortedSet<(int freq, int num)>();

        foreach (var pair in freqMap)
        {
            heap.Add((pair.Value, pair.Key));

            // Keep only k elements in heap
            if (heap.Count > k)
            {
                heap.Remove(heap.Min);
            }
        }

        // Extract elements from heap
        int[] result = new int[k];
        int i = 0;
        foreach (var item in heap)
        {
            result[i++] = item.num;
        }

        return result;
    }
}
```

### Approach 3: Optimal - Bucket Sort

**Detailed Reasoning:** Since frequency is bounded by array length, we can use bucket sort. Create buckets indexed by frequency, then collect elements from highest frequency buckets. This achieves linear time complexity.

**Time Complexity:  $O(n)$**  - linear time for counting and bucketing

**Space Complexity:  $O(n)$**  - for buckets and frequency map

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

public class Solution
{
    /// <summary>
    /// Finds top K frequent elements using bucket sort approach
    /// Time:  $O(n)$ , Space:  $O(n)$ 
    /// </summary>
    public int[] TopKFrequent(int[] nums, int k)
    {
        // Step 1: Count frequencies
        var freqMap = new Dictionary<int, int>();
        foreach (int num in nums)
        {
            freqMap[num] = freqMap.GetValueOrDefault(num, 0) + 1;
        }

        // Step 2: Create buckets indexed by frequency
        // buckets[i] contains all numbers with frequency i
        List<int>[] buckets = new List<int>[nums.Length + 1];

        foreach (var pair in freqMap)
        {
            int freq = pair.Value;
            int num = pair.Key;

            if (buckets[freq] == null)
            {
                buckets[freq] = new List<int>();
            }
            buckets[freq].Add(num);
        }

        // Step 3: Collect top k elements from highest frequency buckets
        var result = new List<int>();

        // Start from highest possible frequency and work down
        for (int freq = nums.Length; freq >= 0 && result.Count < k; freq--)
        {
            if (buckets[freq] != null)
            {
                foreach (int num in buckets[freq])
                {
                    result.Add(num);
                    if (result.Count == k) break;
                }
            }
        }
    }
}
```



```

        }
    }
}

return result.ToArray();
}
}

```

## Key Takeaways:

- **Bucket sort optimization:** When range is limited, bucket sort achieves linear time
- **Frequency-based bucketing:** Index buckets by frequency for efficient collection
- **Early termination:** Stop collecting once we have k elements

## PROBLEM 6: PRODUCT OF ARRAY EXCEPT SELF

### Problem Statement:

Given an integer array `nums`, return an array `answer` such that `answer[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

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

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

### Test Cases:

Example 1:

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

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

Explanation:  $24=2*3*4$ ,  $12=1*3*4$ ,  $8=1*2*4$ ,  $6=1*2*3$

Example 2:

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

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

Explanation: Product except -1 is 0 (due to 0), except 1 is 0, except 0 is  $(-1)*1*(-3)*3=9$

Example 3:

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

Output: `[60,40,30,24]`

Explanation:  $60=3*4*5$ ,  $40=2*4*5$ ,  $30=2*3*5$ ,  $24=2*3*4$

## Interview Questions & Answers:

1. **Q:** "Why can't we use division?"

**A:** Division fails with zeros in the array and problem explicitly forbids it. Also, division can cause precision issues with floating-point arithmetic.

2. **Q:** "How do you handle zeros in the array?"

**A:** If there's one zero, all products except the zero position will be 0. If there are multiple

zeros, all products will be 0.

3. **Q:** "Can you solve this with constant extra space?"

**A:** Yes, by using the output array itself to store prefix products, then computing suffix products on the fly.

4. **Q:** "What if the array contains very large numbers?"

**A:** Problem states products fit in 32-bit integer, but in practice, we'd need to consider overflow and possibly use long or BigInteger.

5. **Q:** "How would you modify this to handle the division approach safely?"

**A:** Count zeros: if no zeros, divide total product by each element. If one zero, only that position gets the product of other elements. If multiple zeros, all results are 0.

## Approach 1: Brute Force - Calculate Each Product

**Idea:** For each position, calculate the product of all other elements by multiplying them explicitly.

**Time Complexity:**  $O(n^2)$  - for each element, iterate through all others

**Space Complexity:**  $O(1)$  - only output array space

```
using System;

public class Solution
{
    public int[] ProductExceptSelf(int[] nums)
    {
        int n = nums.Length;
        int[] result = new int[n];

        // For each position, calculate product of all other elements
        for (int i = 0; i < n; i++)
        {
            int product = 1;

            // Multiply all elements except nums[i]
            for (int j = 0; j < n; j++)
            {
                if (i != j)
                {
                    product *= nums[j];
                }
            }

            result[i] = product;
        }

        return result;
    }
}
```

## Approach 2: Left and Right Product Arrays

**Explanation:** Create separate arrays for left products (product of all elements to the left) and right products (product of all elements to the right). Final result is  $\text{left}[i] * \text{right}[i]$ .

**Time Complexity:  $O(n)$**  - three passes through array

**Space Complexity:  $O(n)$**  - for left and right arrays

```
using System;

public class Solution
{
    public int[] ProductExceptSelf(int[] nums)
    {
        int n = nums.Length;
        int[] left = new int[n];
        int[] right = new int[n];
        int[] result = new int[n];

        // Calculate left products
        left[0] = 1;
        for (int i = 1; i < n; i++)
        {
            left[i] = left[i - 1] * nums[i - 1];
        }

        // Calculate right products
        right[n - 1] = 1;
        for (int i = n - 2; i >= 0; i--)
        {
            right[i] = right[i + 1] * nums[i + 1];
        }

        // Combine left and right products
        for (int i = 0; i < n; i++)
        {
            result[i] = left[i] * right[i];
        }

        return result;
    }
}
```

## Approach 3: Optimal - Constant Space

**Detailed Reasoning:** Use output array to store left products, then traverse right-to-left while maintaining running right product. This eliminates the need for separate left/right arrays while maintaining  $O(n)$  time complexity.

**Time Complexity:  $O(n)$**  - two passes through array

**Space Complexity:  $O(1)$**  - only using output array space (not counting output array as extra space)

```

using System;

public class Solution
{
    /// <summary>
    /// Calculates product of array except self using constant extra space
    /// Time: O(n), Space: O(1)
    /// </summary>
    public int[] ProductExceptSelf(int[] nums)
    {
        int n = nums.Length;
        int[] result = new int[n];

        // First pass: fill result array with left products
        // result[i] will contain product of all elements to left of i
        result[0] = 1; // No elements to left of first element
        for (int i = 1; i < n; i++)
        {
            result[i] = result[i - 1] * nums[i - 1];
        }

        // Second pass: multiply by right products
        // Use a variable to track running product of elements to the right
        int rightProduct = 1;
        for (int i = n - 1; i >= 0; i--)
        {
            // result[i] already has left product, multiply by right product
            result[i] *= rightProduct;

            // Update right product for next iteration
            rightProduct *= nums[i];
        }

        return result;
    }
}

```

## Key Takeaways:

- **Prefix and suffix products:** Decompose problem into left and right components
- **Space optimization:** Reuse output array to eliminate extra space
- **Running product technique:** Maintain state while traversing to avoid multiple arrays

## PROBLEM 7: VALID SUDOKU

### Problem Statement:

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

1. Each row must contain the digits 1-9 without repetition.

2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine  $3 \times 3$  sub-boxes of the grid must contain the digits 1-9 without repetition.

Note: A Sudoku board (partially filled) could be valid but is not necessarily solvable. Only the filled cells need to be validated.

## Test Cases:

Example 1:

Input: board =

```
[["5","3",".",".","7",".",".",".","."],
 ["6",".",".","1","9","5",".",".","."],
 [".","9","8",".",".",".","6","."],
 ["8",".",".","6",".",".","3"],
 ["4",".","8",".","3",".","1"],
 ["7",".","2",".","6"],
 [".","6",".","2","8"],
 [".","4","1","9",".","5"],
 [".","8",".","7","9"]]
```

Output: true

Example 2:

Input: board =

```
[["8","3",".",".","7",".",".",".","."],
 ["6",".",".","1","9","5",".",".","."],
 [".","9","8",".",".",".","6","."],
 ["8",".",".","6",".",".","3"],
 ["4",".","8",".","3",".","1"],
 ["7",".","2",".","6"],
 [".","6",".","2","8"],
 [".","4","1","9",".","5"],
 [".","8",".","7","9"]]
```

Output: false

Explanation: Same as Example 1, except with the 5 in the top left corner changed to 8. Since this is no longer a valid Sudoku board, the function returns false.

Example 3:

Input: board =

```
[[".",".","5",".","1","."],
 [".","4","3",".",".","."],
 [".",".","3",".","1"],
 ["8",".","2","7","."],
 [".","2","7","."],
 [".","1","5","."],
 [".","2","9","."],
 [".","2","9","."],
 [".","4",".","."]]
```

Output: true

## Interview Questions & Answers:

1. **Q:** "How do you identify which 3×3 box a cell belongs to?"  
**A:** For cell (row, col), the 3×3 box index is  $(\text{row}/3) * 3 + (\text{col}/3)$ . This gives each box a unique identifier from 0-8.
2. **Q:** "What's the most efficient way to track duplicates?"  
**A:** Use HashSet for each row, column, and 3×3 box. As we process each cell, check if the digit already exists in corresponding sets.
3. **Q:** "How do you handle empty cells ('.' characters)?"  
**A:** Skip empty cells since they don't violate any rules. Only validate filled cells.
4. **Q:** "Can you validate incrementally as you fill the board?"  
**A:** Yes, this approach works well for Sudoku solvers where you validate each placement before trying the next.
5. **Q:** "What if the board size was different (not 9×9)?"  
**A:** The algorithm generalizes to any n×n board with  $\sqrt{n} \times \sqrt{n}$  boxes, just change the constants and box calculation formula.

## Approach 1: Brute Force - Check Each Rule Separately

**Idea:** For each filled cell, check if it violates any of the three rules by scanning its entire row, column, and 3×3 box.

**Time Complexity:**  $O(n^2)$  where  $n=9$ , but with high constant factor due to repeated scanning

**Space Complexity:**  $O(1)$  - no extra data structures

```
using System;

public class Solution
{
    public bool IsValidSudoku(char[][] board)
    {
        // Check each filled cell against all three rules
        for (int row = 0; row < 9; row++)
        {
            for (int col = 0; col < 9; col++)
            {
                if (board[row][col] != '.')
                {
                    char digit = board[row][col];

                    // Check row for duplicates
                    if (!IsValidInRow(board, row, col, digit)) return false;

                    // Check column for duplicates
                    if (!IsValidInColumn(board, row, col, digit)) return false;

                    // Check 3x3 box for duplicates
                    if (!IsValidInBox(board, row, col, digit)) return false;
                }
            }
        }
    }
}
```

```

    }
    return true;
}

private bool IsValidInRow(char[][] board, int row, int col, char digit)
{
    for (int c = 0; c < 9; c++)
    {
        if (c != col && board[row][c] == digit) return false;
    }
    return true;
}

private bool IsValidInColumn(char[][] board, int row, int col, char digit)
{
    for (int r = 0; r < 9; r++)
    {
        if (r != row && board[r][col] == digit) return false;
    }
    return true;
}

private bool IsValidInBox(char[][] board, int row, int col, char digit)
{
    int startRow = (row / 3) * 3;
    int startCol = (col / 3) * 3;

    for (int r = startRow; r < startRow + 3; r++)
    {
        for (int c = startCol; c < startCol + 3; c++)
        {
            if ((r != row || c != col) && board[r][c] == digit) return false;
        }
    }
    return true;
}
}

```

## Approach 2: HashSet for Each Row, Column, Box

**Explanation:** Create separate HashSets to track digits in each row, column, and 3×3 box. Single pass through board with constant-time duplicate checking.

**Time Complexity:**  $O(n^2)$  where  $n=9$  - single pass with  $O(1)$  operations

**Space Complexity:**  $O(n^2)$  - for HashSets

```

using System;
using System.Collections.Generic;

public class Solution
{
    public bool IsValidSudoku(char[][] board)
    {
        // Create HashSets for tracking digits

```

```

HashSet<char>[] rows = new HashSet<char>[9];
HashSet<char>[] cols = new HashSet<char>[9];
HashSet<char>[] boxes = new HashSet<char>[9];

// Initialize HashSets
for (int i = 0; i < 9; i++)
{
    rows[i] = new HashSet<char>();
    cols[i] = new HashSet<char>();
    boxes[i] = new HashSet<char>();
}

// Single pass through the board
for (int row = 0; row < 9; row++)
{
    for (int col = 0; col < 9; col++)
    {
        char digit = board[row][col];
        if (digit != '.')
        {
            // Calculate which 3x3 box this cell belongs to
            int boxIndex = (row / 3) * 3 + col / 3;

            // Check if digit already exists in row, column, or box
            if (rows[row].Contains(digit) ||
                cols[col].Contains(digit) ||
                boxes[boxIndex].Contains(digit))
            {
                return false;
            }

            // Add digit to corresponding sets
            rows[row].Add(digit);
            cols[col].Add(digit);
            boxes[boxIndex].Add(digit);
        }
    }
}

return true;
}

```

### Approach 3: Optimal - Single HashSet with Encoded Keys

**Detailed Reasoning:** Instead of maintaining separate HashSets, use a single HashSet with encoded strings to represent "digit in row X", "digit in column Y", "digit in box Z". This is more memory efficient and equally fast.

**Time Complexity:**  $O(n^2)$  where  $n=9$  - single pass

**Space Complexity:**  $O(n^2)$  - single HashSet with encoded entries

```

using System;
using System.Collections.Generic;

```



```

public class Solution
{
    /// <summary>
    /// Validates Sudoku board using single HashSet with encoded keys
    /// Time: O(1) since board is fixed 9x9, Space: O(1)
    /// </summary>
    public bool IsValidSudoku(char[][] board)
    {
        // Single HashSet to track all constraints
        var seen = new HashSet<string>();

        // Single pass through the board
        for (int row = 0; row < 9; row++)
        {
            for (int col = 0; col < 9; col++)
            {
                char digit = board[row][col];

                if (digit != '.')
                {
                    // Create encoded strings for each constraint
                    string inRow = $"row{row}-{digit}";
                    string inCol = $"col{col}-{digit}";
                    string inBox = $"box{row/3}{col/3}-{digit}";

                    // Check if any constraint is violated
                    if (seen.Contains(inRow) ||
                        seen.Contains(inCol) ||
                        seen.Contains(inBox))
                    {
                        return false;
                    }

                    // Add all constraints for this digit
                    seen.Add(inRow);
                    seen.Add(inCol);
                    seen.Add(inBox);
                }
            }
        }

        return true;
    }
}

```

## Key Takeaways:

- **Constraint encoding:** Represent multiple constraint types in single data structure
- **3×3 box calculation:**  $(row/3) * 3 + col/3$  maps cells to box indices
- **Single-pass validation:** Check all constraints simultaneously for efficiency

## PROBLEM 8: ENCODE AND DECODE STRINGS

### Problem Statement:

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Note: The string may contain any possible characters out of 256 valid ASCII characters. Your algorithm should be generalized enough to work on any possible characters.

### Test Cases:

```
Example 1:
Input: strs = ["hello","world"]
Output: ["hello","world"]
Explanation:
Encoded: "5#hello5#world"
Decoded back: ["hello","world"]
```

```
Example 2:
Input: strs = [""]
Output: [""]
Explanation:
Encoded: "0#"
Decoded back: [""]
```

```
Example 3:
Input: strs = ["a","bc","def"]
Output: ["a","bc","def"]
Explanation:
Encoded: "1#a2#bc3#def"
Decoded back: ["a","bc","def"]
```

### Interview Questions & Answers:

- Q:** "What if strings contain the delimiter character?"  
**A:** Use length-prefixed encoding like "length#string" so delimiters inside strings don't interfere with parsing.
- Q:** "How do you handle empty strings?"  
**A:** Empty string becomes "0#" - zero length followed by empty content.
- Q:** "What about very long strings?"  
**A:** Length-prefixed approach handles any length. For extremely large strings, might need streaming approach.
- Q:** "Can you use a different encoding scheme?"  
**A:** Yes - could use escape sequences, base64 encoding, or binary length prefixes, but length-prefix is simplest and most efficient.
- Q:** "How would you make this more efficient for network transmission?"  
**A:** Add compression (gzip), use binary encoding instead of ASCII, or implement streaming

for very large datasets.

## Approach 1: Delimiter-Based (Problematic)

**Idea:** Use a special delimiter between strings. This approach has issues when strings contain the delimiter.

**Time Complexity:**  $O(n)$  where  $n$  is total length of all strings

**Space Complexity:**  $O(n)$  for encoded string

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

public class Codec
{
    // This approach is flawed but shown for educational purposes
    public string encode(IList<string> strs)
    {
        if (strs.Count == 0) return "";

        var sb = new StringBuilder();
        foreach (string s in strs)
        {
            sb.Append(s);
            sb.Append("|||"); // Delimiter that might appear in strings!
        }

        // Remove last delimiter
        return sb.ToString().TrimEnd('|||');
    }

    public IList<string> decode(string s)
    {
        if (string.IsNullOrEmpty(s)) return new List<string>();

        return new List<string>(s.Split("|||")); // Fails if strings contain |||
    }
}
```

## Approach 2: Escape Sequences

**Explanation:** Use escape sequences for special characters, similar to how strings handle quotes with backslashes.

**Time Complexity:**  $O(n)$  for encoding/decoding

**Space Complexity:**  $O(n)$  for encoded string

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

```

public class Codec
{
    public string encode(IList<string> strs)
    {
        var sb = new StringBuilder();

        foreach (string str in strs)
        {
            // Escape special characters
            foreach (char c in str)
            {
                if (c == '\\' || c == '|')
                {
                    sb.Append('\\');
                }
                sb.Append(c);
            }
            sb.Append('|'); // Delimiter between strings
        }

        return sb.ToString();
    }

    public IList<string> decode(string s)
    {
        var result = new List<string>();
        var current = new StringBuilder();
        bool escaped = false;

        foreach (char c in s)
        {
            if (escaped)
            {
                current.Append(c);
                escaped = false;
            }
            else if (c == '\\')
            {
                escaped = true;
            }
            else if (c == '|')
            {
                result.Add(current.ToString());
                current.Clear();
            }
            else
            {
                current.Append(c);
            }
        }

        return result;
    }
}

```

### Approach 3: Optimal - Length-Prefixed Encoding

**Detailed Reasoning:** Prefix each string with its length followed by a delimiter. This completely avoids issues with special characters in strings and provides unambiguous parsing. This is the most robust approach used in many real protocols.

**Time Complexity:**  $O(n)$  where  $n$  is total length of all strings

**Space Complexity:**  $O(n)$  for encoded string

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

public class Codec
{
    /// <summary>
    /// Encodes strings using length-prefixed format: "length#string"
    /// Time:  $O(n)$ , Space:  $O(n)$ 
    /// </summary>
    public string encode(IList<string> strs)
    {
        var sb = new StringBuilder();

        foreach (string str in strs)
        {
            // Format: "length#string"
            sb.Append(str.Length);
            sb.Append('#');
            sb.Append(str);
        }

        return sb.ToString();
    }

    /// <summary>
    /// Decodes length-prefixed encoded string back to original list
    /// Time:  $O(n)$ , Space:  $O(n)$ 
    /// </summary>
    public IList<string> decode(string s)
    {
        var result = new List<string>();
        int i = 0;

        while (i < s.Length)
        {
            // Find the delimiter '#'
            int delimiterPos = s.IndexOf('#', i);

            // Extract length
            int length = int.Parse(s.Substring(i, delimiterPos - i));

            // Extract string of specified length
            string str = s.Substring(delimiterPos + 1, length);
            result.Add(str);
        }
    }
}
```

```

        // Move to next encoded string
        i = delimiterPos + 1 + length;
    }

    return result;
}
}

// Usage:
// Codec codec = new Codec();
// IList<string> strs = new List<string> {"hello", "world"};
// string encoded = codec.encode(strs);
// IList<string> decoded = codec.decode(encoded);

```

## Key Takeaways:

- **Length-prefixed encoding:** Robust solution for arbitrary string content
- **Avoid delimiter conflicts:** Length prefix eliminates ambiguity
- **Protocol design:** This pattern is used in many network protocols and file formats

This completes all 8 problems for Day 1: Arrays & Hashing with the detailed structure you requested. Each problem includes the complete analysis with multiple approaches, interview questions, and detailed explanations following your specifications.