03. Solved Coding PYQs - String & Matrix Problems

Problem 1: String Compression (Run-Length Encoding)

Problem Statement

Write a function to perform basic string compression using the counts of repeated characters. For example, `"aabccccccaaa"` becomes `"a2b1c5a3"`. If the compressed string is not smaller than the original string, return the original string.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <string>

using namespace std;

string compressString(const string& str) {
    if (str.empty()) return str;

    string compressed = "";
    int count = 1;

    for (size_t i = 0; i < str.length(); ++i) {
        if (i + 1 < str.length() && str[i] == str[i + 1]) {
            count++;
        } else {
            compressed += str[i];
            compressed += to_string(count);
            count = 1;
        }
    }

    return compressed.length() < str.length() ? compressed : str;
}

int main() {
    string input = "aabccccccaaa";
    cout << "Compressed: " << compressString(input) << endl; // Output: a2b1c5a3
    return 0;
}
```

Java Solution

```
public class StringCompressor {
    public static String compress(String str) {
        if (str == null || str.isEmpty()) return str;

        StringBuilder sb = new StringBuilder();
        int count = 1;

        for (int i = 0; i < str.length(); i++) {
            if (i + 1 < str.length() && str.charAt(i) == str.charAt(i + 1)) {
                count++;
            } else {
                sb.append(str.charAt(i));
                sb.append(count);
                count = 1;
            }
        }

        String compressed = sb.toString();
        return compressed.length() < str.length() ? compressed : str;
    }

    public static void main(String[] args) {
        String input = "aabcccccaaa";
        System.out.println("Compressed: " + compress(input)); // Output: a2b1c5a3
    }
}
```

Python Solution

```
def compress_string(s):
    if not s:
        return s

    compressed = []
    count = 1

    for i in range(len(s)):
        if i + 1 < len(s) and s[i] == s[i + 1]:
            count += 1
        else:
            compressed.append(s[i] + str(count))
            count = 1

    result = "".join(compressed)
    return result if len(result) < len(s) else s
```

```
# Driver test
input_str = "aabcccccaaa"
print("Compressed:", compress_string(input_str))  # Output: a2b1c5a3
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — One pass through string of length N.
- **Space Complexity**: $\mathcal{O}(N)$ — Memory needed for the compressed output.

Problem 2: Valid Anagram Check

Problem Statement

Given two strings `s` and `t`, return `true` if `t` is an anagram of `s`, and `false` otherwise.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

bool isAnagram(string s, string t) {
    if (s.length() != t.length()) return false;

    vector<int> freq(256, 0);
    for (size_t i = 0; i < s.length(); i++) {
        freq[(unsigned char)s[i]]++;
        freq[(unsigned char)t[i]]--;
    }

    for (int count : freq) {
        if (count != 0) return false;
    }
    return true;
}

int main() {
    cout << boolalpha << isAnagram("anagram", "nagaram") << endl; // Output: true
    return 0;
}
```

Java Solution

```
public class AnagramCheck {
    public static boolean isAnagram(String s, String t) {
        if (s == null || t == null || s.length() != t.length()) return false;

        int[] freq = new int[256];
        for (int i = 0; i < s.length(); i++) {
            freq[s.charAt(i)]++;
            freq[t.charAt(i)]--;
        }

        for (int count : freq) {
            if (count != 0) return false;
        }
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isAnagram("anagram", "nagaram")); // Output: true
    }
}
```

Python Solution

```
def is_anagram(s: str, t: str) -> bool:
    if len(s) != len(t):
        return False
    freq = {}
    for char in s:
        freq[char] = freq.get(char, 0) + 1
    for char in t:
        if char not in freq or freq[char] == 0:
            return False
        freq[char] -= 1
    return True

# Driver test
print(is_anagram("anagram", "nagaram"))  # Output: True
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — Linear scan of strings.
- **Space Complexity**: $\mathcal{O}(1)$ — Fixed size frequency table of 256 ASCII characters.

Problem 3: Longest Substring Without Repeating Characters

Problem Statement

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

Solution Implementations

C++ Solution

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

int lengthOfLongestSubstring(string s) {
    vector<int> lastIndex(256, -1);
    int maxLength = 0;
    int start = 0;

    for (int i = 0; i < (int)s.length(); i++) {
        unsigned char ch = s[i];
        if (lastIndex[ch] >= start) {
            start = lastIndex[ch] + 1;
        }
        lastIndex[ch] = i;
        maxLength = max(maxLength, i - start + 1);
    }
    return maxLength;
}

int main() {
    string s = "abcabcbb";
    cout << "Longest Substring Length: " << lengthOfLongestSubstring(s) << endl; //
Output: 3
    return 0;
}
```

Java Solution

```
import java.util.*;

public class LongestUniqueSubstring {
    public static int lengthOfLongestSubstring(String s) {
        if (s == null) return 0;
```

```
        int[] lastIndex = new int[256];
        Arrays.fill(lastIndex, -1);

        int maxLength = 0;
        int start = 0;

        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (lastIndex[ch] >= start) {
                start = lastIndex[ch] + 1;
            }
            lastIndex[ch] = i;
            maxLength = Math.max(maxLength, i - start + 1);
        }
        return maxLength;
    }

    public static void main(String[] args) {
        String s = "abcabcbb";
        System.out.println("Longest Substring Length: " +
lengthOfLongestSubstring(s)); // Output: 3
    }
}
```

Python Solution

```
def length_of_longest_substring(s: str) -> int:
    char_map = {}
    max_length = 0
    start = 0

    for i, char in enumerate(s):
        if char in char_map and char_map[char] >= start:
            start = char_map[char] + 1
        char_map[char] = i
        max_length = max(max_length, i - start + 1)

    return max_length

# Driver test
s = "abcabcbb"
print("Longest Substring Length:", length_of_longest_substring(s))  # Output: 3
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — Sliding window algorithm traverses input string once.
- **Space Complexity**: $\mathcal{O}(\min(N, M))$ — Space for character index map/array where M is character set size.

Problem 4: Matrix Spiral Order Traversal

Problem Statement

Given an $M \times N$ 2D matrix, return all elements of the matrix in **spiral order**.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>

using namespace std;

vector<int> spiralOrder(const vector<vector<int>>& matrix) {
    if (matrix.empty()) return {};
    int top = 0, bottom = matrix.size() - 1;
    int left = 0, right = matrix[0].size() - 1;
    vector<int> result;

    while (top <= bottom && left <= right) {
        for (int col = left; col <= right; col++) {
            result.push_back(matrix[top][col]);
        }
        top++;

        for (int row = top; row <= bottom; row++) {
            result.push_back(matrix[row][right]);
        }
        right--;

        if (top <= bottom) {
            for (int col = right; col >= left; col--) {
                result.push_back(matrix[bottom][col]);
            }
            bottom--;
        }

        if (left <= right) {
            for (int row = bottom; row >= top; row--) {
                result.push_back(matrix[row][left]);
            }
            left++;
        }
    }

    return result;
```

```
}

int main() {
    vector<vector<int>> matrix = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    vector<int> res = spiralOrder(matrix);
    cout << "Spiral Order: ";
    for (int val : res) cout << val << " ";
    cout << endl; // Output: 1 2 3 4 8 12 11 10 9 5 6 7
    return 0;
}
```

Java Solution

```
import java.util.*;

public class MatrixSpiral {
    public static List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        if (matrix == null || matrix.length == 0) return result;

        int top = 0, bottom = matrix.length - 1;
        int left = 0, right = matrix[0].length - 1;

        while (top <= bottom && left <= right) {
            for (int col = left; col <= right; col++) {
                result.add(matrix[top][col]);
            }
            top++;

            for (int row = top; row <= bottom; row++) {
                result.add(matrix[row][right]);
            }
            right--;

            if (top <= bottom) {
                for (int col = right; col >= left; col--) {
                    result.add(matrix[bottom][col]);
                }
                bottom--;
            }

            if (left <= right) {
                for (int row = bottom; row >= top; row--) {
                    result.add(matrix[row][left]);
                }
```

```
                left++;
            }
        }
        return result;
    }

    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 10, 11, 12}
        };
        System.out.println("Spiral Order: " + spiralOrder(matrix));
    }
}
```

Python Solution

```
def spiral_order(matrix):
    if not matrix or not matrix[0]:
        return []

    top, bottom = 0, len(matrix) - 1
    left, right = 0, len(matrix[0]) - 1
    result = []

    while top <= bottom and left <= right:
        for col in range(left, right + 1):
            result.append(matrix[top][col])
        top += 1

        for row in range(top, bottom + 1):
            result.append(matrix[row][right])
        right -= 1

        if top <= bottom:
            for col in range(right, left - 1, -1):
                result.append(matrix[bottom][col])
            bottom -= 1

        if left <= right:
            for row in range(bottom, top - 1, -1):
                result.append(matrix[row][left])
            left += 1

    return result

# Driver test
matrix = [
```

```
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
print("Spiral Order:", spiral_order(matrix))  # Output: [1, 2, 3, 4, 8, 12, 11, 10, 9, 5,
6, 7]
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(M \times N)$ — Every matrix element is visited exactly once.
- **Space Complexity**: $\mathcal{O}(1)$ — Auxiliary space (excluding output list space).