NextGenX AI Club – Core Team Recruitment Task

Domain: Data Structures & Algorithms (DSA)

Introduction

I am **Arpit Singh**, a **2nd year B.Tech Computer Science and Engineering** student at **IILM University**, currently associated with **Batch 2CSE5 (Old)**, with the updated batch allocation awaited. I am submitting this task as part of the **NextGenX AI Club – Core Team Recruitment Process**, under the **Data Structures and Algorithms (DSA)** domain.

This submission is developed using **Java**, with a deliberate focus on **core DSA fundamentals**, logical clarity, and clean implementation. Instead of relying on built-in shortcuts or advanced libraries, I have approached each problem from a **foundational perspective**, ensuring that the solutions reflect a strong understanding of how data structures work internally.

While solving the problems, special attention was given to **robust input handling**, edge-case management, and overall code stability. Additional effort was invested in **polishing the programs** so that they do not fail due to invalid inputs, unexpected user behavior, or limited input assumptions. Wherever applicable, the solutions were generalized to handle **strings, numbers, and special characters**, making them more flexible and realistic.

The overall objective of this submission is not only to provide correct answers, but to demonstrate **problem-solving discipline, defensive programming practices, and clean DSA-based thinking**, in alignment with the technical standards expected from a NextGenX AI Club core team member.

Personal Details

- **Name:** Arpit Singh
- **Email:** arpit.singh.cs28@iilm.edu
- **Phone Number:** 8273308199
- **Course:** B.Tech Computer Science and Engineering
- **Year:** 2nd Year
- **Batch:** 2CSE5 (Old) – *New batch allocation awaited*
- **Roll Number:** 2410030287
- **Language Used:** Java

Problem 1: Frequency Counter

Core Algorithmic Logic (Language Independent)

The fundamental logic of the frequency counter is based on three steps:
(1) sorting the input elements, **(2) grouping identical elements**, and **(3) counting consecutive occurrences**.
By sorting the input first, identical elements become adjacent. This allows the frequency of each unique element to be calculated using a single linear traversal, without requiring additional data structures such as hash tables. This approach is generic and works for any comparable data type.

Code Structure & Detailed Logic Explanation

1. Input Validation for Total Number of Elements

Before allocating the array, the program validates the total number of elements. The input is first read as a string and checked using a regular expression to ensure it represents a valid positive integer. This prevents invalid inputs such as alphabets, symbols, negative values, or zero from causing runtime errors like invalid array sizes.

The loop continues until a valid input is provided, ensuring that the program never fails at initialization.

---

2. Generic Data Representation Using String Array

The elements are stored in a `String[]` array instead of a numeric array. This design choice allows the frequency counter to handle **numbers, strings, and special characters uniformly**, without altering the core algorithm.

Treating all inputs as strings ensures that the frequency logic remains independent of the data type, while still maintaining correctness through proper comparison methods.

---

3. Sorting Logic (Selection Sort – Comparison Based)

The `sortArray` method implements a comparison-based sorting algorithm similar to **Selection Sort**. For each index, the element is compared with all subsequent elements, and a swap is performed if the order is incorrect.

The `compareTo()` method is used for comparison, which enables lexicographical ordering for strings and consistent ordering for numeric values represented as strings. This sorting step is critical because it ensures that identical elements are placed consecutively, which directly enables efficient frequency counting.

---

4. Robust Element Input Handling

Each element is read inside a validation loop to ensure that empty inputs are not accepted. If an empty value is entered, the program prompts the user again. This guarantees that every array position contains a valid element before further processing.

This defensive input handling prevents logical errors and ensures data integrity throughout the execution.

---

5. Frequency Computation Using Linear Traversal

After sorting, the array is traversed once. A counter variable tracks the frequency of the current element. If the current element matches the next element, the counter is incremented. When a different element is encountered, the frequency of the previous element is printed, and the counter is reset.

This method efficiently computes frequencies in a single pass, leveraging the sorted order of elements and avoiding unnecessary recomputation.

---

6. Output Formatting and Resource Management

The frequency of each element is displayed in a clear and structured format, explicitly stating the number of occurrences. The `Scanner` resource is closed at the end of execution, following standard best practices for resource management.

Robustness and Code Polishing

The solution is intentionally hardened to prevent common failure scenarios:

- Invalid array size inputs are rejected
- Empty element inputs are disallowed
- The program never terminates unexpectedly due to incorrect user input
- The algorithm supports heterogeneous input types through string-based handling

These refinements improve the reliability and stability of the program while preserving core DSA principles.

Time Complexity

- Sorting (Selection Sort): **O(n²)**
- Frequency traversal: **O(n)**
- Overall Time Complexity: **O(n²)**

Space Complexity

- **O(1)** auxiliary space (in-place sorting)

Problem 2: Custom Stack Implementation

Core Algorithmic Logic (Language Independent)

A stack is a linear data structure that follows the **Last In First Out (LIFO)** principle. The fundamental operations of a stack are **Push**, **Pop**, **Peek**, and **IsEmpty**. Internally, a stack can be implemented using a sequential storage structure and a pointer that tracks the position of the top element.

The key idea is to control access to the structure by allowing insertions and deletions only from one end, while explicitly handling overflow and underflow conditions to maintain correctness and stability.

Code Structure & Detailed Logic Explanation

1. Stack Representation Using Array and Top Pointer

The stack is represented using an array along with an integer variable `top`. The array stores the stack elements, while `top` keeps track of the index of the most recently inserted element. Initially, `top` is set to `-1`, indicating that the stack is empty.

This approach closely follows the classical array-based stack implementation taught in DSA.

---

2. Generic Data Handling Using String Array

The stack is implemented using a `String[]` array. This allows the stack to store **numbers, strings, and special characters** without changing the core stack logic. By treating all inputs as strings, the implementation demonstrates that the stack data structure is independent of the data type it stores.

---

3. Push Operation (Overflow-Safe)

The push operation first checks whether the stack is full by comparing `top` with `size - 1`. If the stack is full, an overflow condition is reported, and no insertion is performed.

If space is available, the `top` pointer is incremented, and the new element is inserted at the updated position. This ensures that elements are added in a controlled and predictable manner.

---

4. Pop Operation (Underflow-Safe)

The pop operation checks whether the stack is empty by verifying if `top` is equal to `-1`. If the stack is empty, an underflow condition is reported.

If the stack contains elements, the value at the `top` index is removed, and the `top` pointer is decremented. This correctly removes the most recently inserted element, maintaining the LIFO property.

---

5. Peek Operation

The peek operation retrieves the element currently at the top of the stack without removing it. This operation is guarded by an empty-stack check to prevent invalid access.

---

6. IsEmpty Operation

The IsEmpty operation simply checks whether the `top` pointer is at its initial value (`-1`). This provides a constant-time check for stack emptiness.

---

7. Menu-Driven Execution

The stack operations are accessed through a menu-driven interface. The program continuously prompts the user to select an operation until an explicit exit option is chosen. This structure allows repeated execution of stack operations within a single run.

Robust Input Handling & Code Polishing

Several defensive programming measures are applied to strengthen the stack implementation:

- Stack size input is validated to ensure it is a positive integer.
- Menu choices accept arbitrary input and validate the selection range.
- Invalid inputs do not terminate the program and instead trigger user re-prompting.
- The push operation accepts strings, numbers, and special characters.
- Overflow and underflow conditions are explicitly handled.

These refinements ensure that the stack implementation is **stable, resilient, and resistant to incorrect user input**, without relying on built-in stack libraries.

Time Complexity

- Push: **O(1)**
- Pop: **O(1)**
- Peek: **O(1)**
- IsEmpty: **O(1)**

Space Complexity

- **O(n)**, where n is the size of the stack

Problem 3: Longest Unique Substring

Core Algorithmic Logic (Language Independent)

The problem of finding the longest substring without repeating characters is solved using the **sliding window technique**. The core idea is to maintain a window over the string such that all characters inside the window are unique. As the window expands, if a duplicate character is encountered, the window is adjusted by moving its starting point forward until the duplicate is removed.

This approach ensures that each character is processed efficiently and avoids recomputing substrings, resulting in linear time complexity.

Code Structure & Detailed Logic Explanation

1. Input Acquisition

The input string is read in a single step and stored for processing. No assumptions are made about the nature of the input; it may contain alphabets, digits, or special characters.

This ensures the algorithm remains generic and not restricted to a specific character set.

---

2. Character Tracking Mechanism

A fixed-size boolean array is used to track whether a character is currently present in the active window. Each index of this array corresponds to a character's ASCII value.

This allows constant-time checks to determine whether a character has already appeared in the current substring.

---

3. Sliding Window Initialization

Two pointers are used:

- A **start pointer**, representing the beginning of the current window
- An **end pointer**, representing the current character being processed

Initially, both pointers start at the beginning of the string, and the maximum substring length is set to zero.

---

4. Window Expansion and Duplicate Handling

As the end pointer moves forward, the algorithm checks whether the current character has already been visited within the window.

- If the character has not appeared before, it is marked as visited, and the window is expanded.
- If the character is already present, the start pointer is moved forward step by step, unmarking characters until the duplicate character is removed from the window.

This guarantees that the window always contains unique characters.

---

5. Maximum Length Calculation

After each successful window expansion, the length of the current valid substring is calculated. If this length is greater than the previously recorded maximum, the maximum value is updated.

This process continues until the entire string has been traversed.

---

6. Output Generation

Once the traversal is complete, the final maximum length of a substring without repeating characters is displayed. The output is formatted clearly to convey the result without ambiguity.

Robustness and Code Polishing

The implementation includes several stability-focused considerations:

- The algorithm works correctly for alphabets, numbers, and special characters.
- No assumptions are made about string length or content.
- Fixed-size auxiliary storage ensures predictable memory usage.
- The solution avoids unnecessary nested loops and redundant checks.

These refinements ensure the program remains **efficient, reliable, and resistant to edge cases**.

Time Complexity

- **O(n)**, where `n` is the length of the string
  Each character is processed at most twice—once when added to the window and once when removed.

Space Complexity

- **O(1)** auxiliary space
  A fixed-size boolean array is used for character tracking.

Final Combined Overview: Design Approach & Code Quality

Across all three problems, the primary focus was on applying **core Data Structures and Algorithms principles** with an emphasis on **clarity, correctness, and robustness**, rather than relying on built-in shortcuts or high-level abstractions.

Each solution was designed around a **language-independent algorithmic idea first**, and then carefully translated into Java with explicit control over data handling, boundary conditions, and user input validation.

Consistent Algorithmic Discipline

- The **Frequency Counter** is built on a sort-and-scan strategy, ensuring frequency computation through ordered grouping and linear traversal.
- The **Custom Stack** follows the classical array-based implementation using a top pointer, explicitly handling overflow and underflow conditions.
- The **Longest Unique Substring** uses the sliding window technique to maintain uniqueness efficiently while traversing the input only once.

In all cases, the logic is driven by **fundamental DSA concepts**, making the solutions easy to reason about and verify.

Robust Input Handling & Defensive Programming

A significant amount of effort was dedicated to **hardening the code against invalid input**, which is a common cause of runtime failures in real-world programs. Across the entire project:

- All numeric inputs are validated before conversion.
- Invalid menu choices are handled gracefully without terminating execution.
- Empty or incorrect element inputs are rejected and re-prompted.
- Programs continue execution safely until a valid exit condition is met.

This defensive approach ensures that none of the solutions fail due to unexpected or incorrect user input, making the programs stable and predictable.

Generic and Flexible Data Handling

Wherever applicable, the implementations were generalized to support **strings, numbers, and special characters**:

- The Frequency Counter uses a `String` array to handle heterogeneous inputs while keeping the algorithm unchanged.
- The Stack implementation stores elements as strings, demonstrating that the stack data structure is independent of the type of data it holds.

This design choice reflects an understanding of how data structures operate conceptually, beyond language-specific constraints.

Code Readability and Maintainability

Throughout the project:

- Code structure is kept modular and readable.
- Logical sections are clearly separated.
- Variable naming and control flow are explicit and intentional.

These practices improve maintainability and make the code easier to review, debug, and extend.

Final Remarks

This submission prioritizes **DSA fundamentals, clean logic, and program stability**. Rather than optimizing prematurely or using advanced libraries, the focus remains on writing solutions that are **correct, resilient, and aligned with foundational computer science principles**.

The overall approach reflects disciplined problem-solving, careful attention to edge cases, and an effort to produce code that performs reliably under varied input conditions—qualities expected from a core team member in a technical club like NextGenX AI Club.