Best Practices

1. StringBuilder

- **Use when:** You need to perform many string manipulations (e.g., concatenation, insertion) inside a loop or in a performance-sensitive scenario.
- **Best Practices:**
  - **Preferred over `String` for mutable strings** in performance-critical code.
  - Use its **`append()`** method instead of concatenation using `+` for efficiency.
  - Initialize with a reasonable **capacity** to avoid resizing when the size is known in advance.

2. StringBuffer

- **Use when:** Thread-safety is required while manipulating strings in multi-threaded environments.
- **Best Practices:**
  - Use `StringBuffer` for thread-safe string manipulation when synchronization is necessary.
  - Avoid using `StringBuffer` in single-threaded environments if performance is a concern, as it's slower than `StringBuilder`.

3. FileReader

- **Use when:** You need to read character files (text files) efficiently.
- **Best Practices:**
  - Always wrap `FileReader` with a **`BufferedReader`** for better performance when reading lines.
  - Handle **IOExceptions** properly.
  - Use `FileReader` for small files; for larger files, consider using streams like `FileInputStream`.

4. InputStreamReader

- **Use when:** You need to convert byte streams into character streams (e.g., reading from non-text files or working with encodings).
- **Best Practices:**
  - Wrap `InputStreamReader` with `BufferedReader` to enhance performance.
  - Always specify the correct **charset** to avoid encoding issues, especially for non-ASCII text.
  - Always close the reader using **try-with-resources** to avoid resource leakage.

5. Linear Search

- **Use when:** Data is unsorted or small-sized, or when simplicity is preferred over performance.
- **Best Practices:**
  - **Return early**: If the element is found, return immediately to avoid unnecessary checks.
  - Avoid using linear search on large data sets; consider binary search or hash-based approaches if performance is critical.

6. Binary Search

- **Use when:** Data is already sorted, and you need an efficient search method.
- **Best Practices:**
  - Ensure the list is **sorted** before using binary search.
  - Use **recursive or iterative** approaches as needed (iterative is generally preferred for better performance).
  - Always check for **index bounds** to avoid `ArrayIndexOutOfBoundsException`.
  - Implement binary search carefully, ensuring the middle index calculation avoids overflow: `mid = low + (high - low) / 2` instead of `mid = (low + high) / 2`.

Problem Statements

StringBuilder Problem 1: Reverse a String Using StringBuilder

Problem:
Write a program that uses **StringBuilder** to reverse a given string. For example, if the input is `"hello"`, the output should be `"olleh"`.

Approach:

1. Create a new `StringBuilder` object.
2. Append the string to the `StringBuilder`.
3. Use the `reverse()` method of `StringBuilder` to reverse the string.
4. Convert the `StringBuilder` back to a string and return it.

```
import java.util.Scanner;

public class ReverseString {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String input = sc.nextLine();
        StringBuilder sb = new StringBuilder(input);
        System.out.println(sb.reverse().toString());
        sc.close();
    }
}
```

StringBuilder Problem 2: Remove Duplicates from a String Using StringBuilder

Problem:
Write a program that uses **StringBuilder** to remove all duplicate characters from a given string while maintaining the original order.

Approach:

1. Initialize an empty `StringBuilder` and a `HashSet` to keep track of characters.
2. Iterate over each character in the string:

- If the character is not in the `HashSet`, append it to the `StringBuilder` and add it to the `HashSet`.

3. Return the `StringBuilder` as a string without duplicates.

```
import java.util.*;

public class RemoveDuplicates {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String input = sc.nextLine();
        StringBuilder result = new StringBuilder();
        HashSet<Character> set = new HashSet<>();
        for (int i = 0; i < input.length(); i++) {
            char c = input.charAt(i);
            if (!set.contains(c)) {
                result.append(c);
                set.add(c);
            }
        }
        System.out.println(result.toString());
        sc.close();
    }
}
```

StringBuffer Problem 1: Concatenate Strings Efficiently Using StringBuffer

Problem:
You are given an array of strings. Write a program that uses **StringBuffer** to concatenate all the strings in the array efficiently.

Approach:

1. Create a new `StringBuffer` object.
2. Iterate through each string in the array and append it to the `StringBuffer`.
3. Return the concatenated string after the loop finishes.
4. Using `StringBuffer` ensures efficient string concatenation due to its mutable nature.

```
import java.util.*;
```

```java
public class ConcatenateStrings {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of strings: ");
        int n = sc.nextInt();
        sc.nextLine();
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < n; i++) {
            System.out.print("Enter string " + (i + 1) + ": ");
            String str = sc.nextLine();
            sb.append(str);
        }
        System.out.println("Concatenated String: " + sb.toString());
    }
}
```

StringBuffer Problem 2: Compare StringBuffer with StringBuilder for String Concatenation

Problem:
Write a program that compares the performance of **StringBuffer** and **StringBuilder** for concatenating strings. For large datasets (e.g., concatenating 1 million strings), compare the execution time of both classes.

Approach:

1. Initialize two `StringBuffer` and `StringBuilder` objects.
2. Perform string concatenation in both objects, appending 1 million strings (e.g., `"hello"`).
3. Measure the time taken to complete the concatenation using `System.nanoTime()` for both `StringBuffer` and `StringBuilder`.
4. Output the time taken by both classes for comparison.

```java
public class CompareBufferBuilder {
    public static void main(String[] args) {
        int n = 1000000;
        String text = "hello";

        long start1 = System.nanoTime();
```

```
        StringBuffer sbuf = new StringBuffer();
        for (int i = 0; i < n; i++) {
            sbuf.append(text);
        }
        long end1 = System.nanoTime();
        System.out.println("StringBuffer time: " + (end1 - start1) + "
ns");

        long start2 = System.nanoTime();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < n; i++) {
            sb.append(text);
        }
        long end2 = System.nanoTime();
        System.out.println("StringBuilder time: " + (end2 - start2) + "
ns");
    }
}
```

FileReader Problem 1: Read a File Line by Line Using FileReader

Problem:
Write a program that uses **FileReader** to read a text file line by line and print each line to the console.

Approach:

1. Create a `FileReader` object to read from the file.
2. Wrap the `FileReader` in a `BufferedReader` to read lines efficiently.
3. Use a loop to read each line using the `readLine()` method and print it to the console.
4. Close the file after reading all the lines.

```
import java.io.*;

public class ReadFile {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter file path: ");
        String filePath = br.readLine();
```

```
        FileReader fr = new FileReader(filePath);
        BufferedReader reader = new BufferedReader(fr);
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        reader.close();
    }
}
```

FileReader Problem 2: Count the Occurrence of a Word in a File Using FileReader

Problem:
Write a program that uses **FileReader** and **BufferedReader** to read a file and count how many times a specific word appears in the file.

Approach:

1. Create a `FileReader` to read from the file and wrap it in a `BufferedReader`.
2. Initialize a counter variable to keep track of word occurrences.
3. For each line in the file, split it into words and check if the target word exists.
4. Increment the counter each time the word is found.
5. Print the final count.

```
import java.io.*;

public class WordCounter {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter file path: ");
        String filePath = br.readLine();
        System.out.print("Enter word to search: ");
        String word = br.readLine();

        FileReader fr = new FileReader(filePath);
        BufferedReader reader = new BufferedReader(fr);
        String line;
```

```
        int count = 0;

        while ((line = reader.readLine()) != null) {
            String[] words = line.split("\\s+");
            for (int i = 0; i < words.length; i++) {
                if (words[i].equals(word)) {
                    count++;
                }
            }
        }
        reader.close();
        System.out.println("Word count: " + count);
    }
}
```

InputStreamReader Problem 1: Convert Byte Stream to Character Stream Using InputStreamReader

Problem:
Write a program that uses **InputStreamReader** to read binary data from a file and print it as characters. The file contains data encoded in a specific charset (e.g., UTF-8).

Approach:

1. Create a `FileInputStream` object to read the binary data from the file.
2. Wrap the `FileInputStream` in an `InputStreamReader` to convert the byte stream into a character stream.
3. Use a `BufferedReader` to read characters efficiently from the `InputStreamReader`.
4. Read the file line by line and print the characters to the console.
5. Handle any encoding exceptions as needed.

```
import java.io.*;

public class ByteToCharReader {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter file path: ");
        String filePath = br.readLine();
```

```
        FileInputStream fis = new FileInputStream(filePath);
        InputStreamReader isr = new InputStreamReader(fis, "UTF-8");
        BufferedReader reader = new BufferedReader(isr);
        String line;

        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        reader.close();
    }
}
```

InputStreamReader Problem 2: Read User Input and Write to File Using InputStreamReader

Problem:
Write a program that uses **InputStreamReader** to read user input from the console and write the input to a file. Each input should be written as a new line in the file.

Approach:

1. Create an `InputStreamReader` to read from `System.in` (the console).
2. Wrap the `InputStreamReader` in a `BufferedReader` for efficient reading.
3. Create a `FileWriter` to write to the file.
4. Read user input using `readLine()` and write the input to the file.
5. Repeat the process until the user enters "exit" to stop inputting.
6. Close the file after the input is finished.

```
import java.io.*;

public class InputToFile {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter output file path: ");
        String filePath = reader.readLine();

        FileWriter fw = new FileWriter(filePath);
        String input;
```

```java
        System.out.println("Enter text (type 'exit' to finish):");

        while (true) {
            input = reader.readLine();
            if (input.equals("exit")) {
                break;
            }
            fw.write(input + "\n");
        }
        fw.close();
        System.out.println("Data written to file.");
    }
}
```

Challenge Problem: Compare StringBuilder, StringBuffer, FileReader, and InputStreamReader

Problem:
Write a program that:

1. Uses **StringBuilder** and **StringBuffer** to concatenate a list of strings 1,000,000 times.
2. Uses **FileReader** and **InputStreamReader** to read a large file (e.g., 100MB) and print the number of words in the file.

Approach:

1. **StringBuilder and StringBuffer:**
   - Create a list of strings (e.g., `"hello"`).
   - Concatenate the strings 1,000,000 times using both `StringBuilder` and `StringBuffer`.
   - Measure and compare the time taken for each.
2. **FileReader and InputStreamReader:**
   - Read a large text file (100MB) using **FileReader** and **InputStreamReader**.
   - Count the number of words by splitting the text on whitespace characters.
   - Print the word count and compare the time taken for reading the file.

```
import java.io.*;

public class ChallengeComparison {
    public static void main(String[] args) throws IOException {

        String str = "hello";
        int times = 1000000;

        long sbStart = System.nanoTime();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < times; i++) {
            sb.append(str);
        }
        long sbEnd = System.nanoTime();
        System.out.println("StringBuilder time: " + (sbEnd - sbStart) + "
ns");

        long sbufStart = System.nanoTime();
        StringBuffer sbuf = new StringBuffer();
        for (int i = 0; i < times; i++) {
            sbuf.append(str);
        }
        long sbufEnd = System.nanoTime();
        System.out.println("StringBuffer time: " + (sbufEnd - sbufStart) +
" ns");

        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter large file path: ");
        String path = br.readLine();

        FileReader fr = new FileReader(path);
        BufferedReader reader1 = new BufferedReader(fr);
        String line;
        int wordCount = 0;
        while ((line = reader1.readLine()) != null) {
            String[] words = line.trim().split("\\s+");
            wordCount += words.length;
        }
        reader1.close();
        System.out.println("Total words (FileReader): " + wordCount);

        FileInputStream fis = new FileInputStream(path);
```

```
        InputStreamReader isr = new InputStreamReader(fis);
        BufferedReader reader2 = new BufferedReader(isr);
        wordCount = 0;
        while ((line = reader2.readLine()) != null) {
            String[] words = line.trim().split("\\s+");
            wordCount += words.length;
        }
        reader2.close();
        System.out.println("Total words (InputStreamReader): " +
wordCount);
    }
}
```

Linear Search Problem 1: Search for the First Negative Number

Problem:
You are given an integer array. Write a program that performs **Linear Search** to find the **first negative number** in the array. If a negative number is found, return its index. If no negative number is found, return -1.

Approach:

1. Iterate through the array from the start.
2. Check if the current element is negative.
3. If a negative number is found, return its index.
4. If the loop completes without finding a negative number, return -1.

```
import java.util.Scanner;

public class FirstNegativeIndex {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter size of array: ");
        int n = sc.nextInt();
        int[] arr = new int[n];

        System.out.println("Enter array elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }

```

```
        for (int i = 0; i < n; i++) {
            if (arr[i] < 0) {
                System.out.println("First negative number index: " + i);
                return;
            }
        }
        System.out.println("-1");
    }
}
```

Linear Search Problem 2: Search for a Specific Word in a List of Sentences

Problem:
You are given an array of sentences (strings). Write a program that performs **Linear Search** to find the **first sentence** containing a specific word. If the word is found, return the sentence. If no sentence contains the word, return "Not Found".

Approach:

1. Iterate through the list of sentences.
2. For each sentence, check if it contains the specific word.
3. If the word is found, return the current sentence.
4. If no sentence contains the word, return "Not Found".

```
import java.util.Scanner;

public class SearchWordInSentences {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of sentences: ");
        int n = sc.nextInt();
        sc.nextLine(); // consume newline

        String[] sentences = new String[n];
        System.out.println("Enter sentences:");
        for (int i = 0; i < n; i++) {
            sentences[i] = sc.nextLine();
```

```
        }

        System.out.print("Enter word to search: ");
        String word = sc.nextLine();

        for (int i = 0; i < n; i++) {
            if (sentences[i].contains(word)) {
                System.out.println("Found in: " + sentences[i]);
                return;
            }
        }
        System.out.println("Not Found");
    }
}
```

Binary Search Problem 1: Find the Rotation Point in a Rotated Sorted Array

Problem:
You are given a **rotated sorted array**. Write a program that performs **Binary Search** to find the **index of the smallest element** in the array (the rotation point).

Approach:

1. Initialize `left` as 0 and `right` as `n - 1`.
2. Perform a binary search:
    - Find the middle element `mid = (left + right) / 2`.
    - If `arr[mid] > arr[right]`, then the smallest element is in the right half, so update `left = mid + 1`.
    - If `arr[mid] < arr[right]`, the smallest element is in the left half, so update `right = mid`.
3. Continue until `left` equals `right`, and then return `arr[left]` (the rotation point).

```
import java.util.Scanner;

public class RotationPoint {
    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter array size: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter rotated sorted array:");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }

        int left = 0, right = n - 1;
        while (left < right) {
            int mid = (left + right) / 2;
            if (arr[mid] > arr[right]) left = mid + 1;
            else right = mid;
        }
        System.out.println("Rotation point index: " + left);
    }
}
```

Binary Search Problem 2: Find the Peak Element in an Array

Problem:
A peak element is an element that is **greater than its neighbors**. Write a program that performs **Binary Search** to find a peak element in an array. If there are multiple peak elements, return any one of them.

Approach:

1. Initialize `left` as 0 and `right` as `n - 1`.
2. Perform a binary search:
    - Find the middle element `mid = (left + right) / 2`.
    - If `arr[mid] > arr[mid - 1]` and `arr[mid] > arr[mid + 1]`, `arr[mid]` is a peak element.
    - If `arr[mid] < arr[mid - 1]`, then search the left half, updating `right = mid - 1`.
    - If `arr[mid] < arr[mid + 1]`, then search the right half, updating `left = mid + 1`.

3. Continue until a peak element is found.

```java
import java.util.Scanner;

public class PeakElement {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter size of array: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter array elements:");
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();

        int left = 0, right = n - 1;
        while (left < right) {
            int mid = (left + right) / 2;
            if (arr[mid] < arr[mid + 1]) left = mid + 1;
            else right = mid;
        }
        System.out.println("Peak element: " + arr[left]);
    }
}
```

Binary Search Problem 3: Search for a Target Value in a 2D Sorted Matrix

Problem:
You are given a 2D matrix where each row is sorted in ascending order, and the first element of each row is greater than the last element of the previous row. Write a program that performs **Binary Search** to find a target value in the matrix. If the value is found, return `true`. Otherwise, return `false`.

Approach:

1. Treat the matrix as a **1D array** (flattened version).
2. Initialize `left` as 0 and `right` as `rows * columns - 1`.
3. Perform binary search:
    - Find the middle element index `mid = (left + right) / 2`.
    - Convert `mid` to row and column indices using `row = mid / numColumns` and `col = mid % numColumns`.

- Compare the middle element with the target:
    - If it matches, return `true`.
    - If the target is smaller, search the left half by updating `right = mid - 1`.
    - If the target is larger, search the right half by updating `left = mid + 1`.

4. If the element is not found, return `false`.

```java
import java.util.Scanner;

public class Search2DMatrix {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter rows and columns: ");
        int rows = sc.nextInt();
        int cols = sc.nextInt();
        int[][] matrix = new int[rows][cols];

        System.out.println("Enter matrix elements (row-wise sorted):");
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                matrix[i][j] = sc.nextInt();

        System.out.print("Enter target to search: ");
        int target = sc.nextInt();

        int left = 0, right = rows * cols - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            int r = mid / cols, c = mid % cols;
            if (matrix[r][c] == target) {
                System.out.println("Found");
                return;
            } else if (matrix[r][c] < target) left = mid + 1;
            else right = mid - 1;
        }
        System.out.println("Not Found");
    }
}
```

Binary Search Problem 4: Find the First and Last Occurrence of an Element in a Sorted Array

Problem:
Given a **sorted array** and a target element, write a program that uses **Binary Search** to find the **first and last occurrence** of the target element in the array. If the element is not found, return `-1`.

Approach:

1. Use binary search to find the **first occurrence**:
   - Perform a regular binary search, but if the target is found, continue searching on the left side (`right = mid - 1`) to find the first occurrence.
2. Use binary search to find the **last occurrence**:
   - Similar to finding the first occurrence, but once the target is found, continue searching on the right side (`left = mid + 1`) to find the last occurrence.
3. Return the indices of the first and last occurrence. If not found, return `-1`.

```
import java.util.Scanner;

public class FirstLastOccurrence {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter sorted array size: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter sorted array:");
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();

        System.out.print("Enter target element: ");
        int target = sc.nextInt();

        int first = -1, last = -1;
        int left = 0, right = n - 1;


        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] == target) {
                first = mid;
                right = mid - 1;
```

```
            } else if (arr[mid] < target) left = mid + 1;
            else right = mid - 1;
        }

        left = 0; right = n - 1;

        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] == target) {
                last = mid;
                left = mid + 1;
            } else if (arr[mid] < target) left = mid + 1;
            else right = mid - 1;
        }

        if (first == -1) System.out.println("-1");
        else System.out.println("First: " + first + ", Last: " + last);
    }
}
```

Challenge Problem (for both Linear and Binary Search)

Problem:
You are given a list of integers. Write a program that uses **Linear Search** to find the **first missing positive integer** in the list and **Binary Search** to find the **index of a given target number**.

Approach:

1. **Linear Search for the first missing positive integer:**
    - Iterate through the list and mark each number in the list as visited (you can use negative marking or a separate array).
    - Traverse the array again to find the first positive integer that is not marked.
2. **Binary Search for the target index:**
    - After sorting the array, perform binary search to find the index of the given target number.
    - Return the index if found, otherwise return `-1`.

```java
import java.util.Arrays;
import java.util.Scanner;

public class ChallengeSearch {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter size of array: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
        boolean[] present = new boolean[n + 1];

        System.out.println("Enter array elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
            if (arr[i] > 0 && arr[i] <= n) present[arr[i]] = true;
        }
        for (int i = 1; i <= n; i++) {
            if (!present[i]) {
                System.out.println("First missing positive: " + i);
                break;
            }
        }
        Arrays.sort(arr);
        System.out.print("Enter target to search: ");
        int target = sc.nextInt();

        int left = 0, right = n - 1, index = -1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] == target) {
                index = mid;
                break;
            } else if (arr[mid] < target) left = mid + 1;
            else right = mid - 1;
        }

        System.out.println("Target index: " + index);
    }
}
```