# 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.

```java
import java.util.*;

public class ReverseString {

    public static String reverse(String str) {
        StringBuilder sb = new StringBuilder(str);
        sb.reverse();

        return sb.toString();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the string to reverse:");
        String str = sc.next();

        System.out.println("Reversed String: " + reverse(str));

        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 RemoveDuplicateCharacter {
    public static String removeDuplicate(String str) {

        HashSet<Character> set = new HashSet<>();
        StringBuilder newStr = new StringBuilder();
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            if (!set.contains(ch)) {
                newStr.append(ch);
                set.add(ch);
            }
        }
        return newStr.toString();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter a string to remove duplicate characters:");
        String str = sc.next();

        System.out.println("Modified String: " + removeDuplicate(str));

        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.*;

public class ConcatString{

    public static String concat(String arr[]) {

        StringBuffer sb = new StringBuffer();

        for (int i = 0; i < arr.length; i++) {
            if (i != 0) {
                sb.append(" ");
            }
            sb.append(arr[i]);
        }

        return sb.toString();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of array of strings:");
        int n = sc.nextInt();

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

```
        System.out.println("String formed concatenating all strings: " +
concat(arr));

        sc.close();
    }
}
```

---

### 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.

```
public class CompareBufferBuilder {
    public static void main(String[] args) {
        String dummyData = "HelloWorld!";

        // for StringBuilder
        StringBuilder sb = new StringBuilder();

        long startTime = System.nanoTime();

        for (int i = 0; i < 1000000; i++) {
            sb.append(dummyData);
        }

        long endTime = System.nanoTime();

```

```
        System.out.println("StringBuilder : " + ((endTime - startTime) /
1_000_000) + " ms");


        // for StringBuffer
        StringBuffer sbf = new StringBuffer();

        startTime = System.nanoTime();

        for (int i = 0; i < 1000000; i++) {
            sbf.append(dummyData);
        }

        endTime = System.nanoTime();

        System.out.println("StringBuffer : " + ((endTime - startTime) /
1_000_000) + " ms");
    }
}
```

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

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 read(String path) {
        try{

            FileReader fr = new FileReader(path);
            BufferedReader br = new BufferedReader(fr);

            String line = br.readLine();
            while (line != null) {
                System.out.println(line);
                line = br.readLine();
            }

            fr.close();
            br.close();

        } catch (Exception e) {
            e.printStackTrace();
        }


    }
    public static void main(String[] args) {
        String path = "text.txt";
        read(path);
    }
}
```

---

## 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.util.*;
import java.io.*;

public class CountWords {

    public static void count(String path, String word) {
        try{
            FileReader fr = new FileReader(path);
            BufferedReader br = new BufferedReader(fr);

            int freq = 0;
            String line = br.readLine();
            while (line != null) {
                System.out.println(line);
                String arr[] = line.split(" ");
                for (int i = 0; i < arr.length; i++) {
                    if (arr[i].equals(word)) {
                        freq++;
                    }
                }

                line = br.readLine();
            }

            System.out.println("Frequency of " + word + " in " + path + " is "
+ freq);
            fr.close();
            br.close();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }


    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the file name you want to read:");
        String path = sc.next();

        System.out.println("Enter a word you want to find the count of:");
        String word = sc.next();

        count(path, word);

        sc.close();
    }
}
```

## 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.

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

public class ReadFile {
    public static void read(String path) {
        try (FileInputStream fis = new FileInputStream(path);
                InputStreamReader isr = new InputStreamReader(fis, "UTF-8");
                BufferedReader br = new BufferedReader(isr)) {

            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }

        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Error reading the file: " + e.getMessage());
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter file path to read the file:");
        String path = sc.next();
        read(path);
        sc.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.

```java
import java.io.*;

public class InputStreamReaderCode {
    public static void main(String[] args) {
        String filePath = "output.txt";

        try {
            // Create InputStreamReader to read from System.in
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);

            // Create FileWriter to write to the file
            FileWriter fw = new FileWriter(filePath);

            System.out.println("Enter text (type 'exit' to stop):");

            String input;
            while (true) {
                input = br.readLine();

                if ("exit".equalsIgnoreCase(input)) {
                    break;
                }

                fw.write(input + "\n");
            }

            br.close();
```

```
            fw.close();

            System.out.println("Input saved to " + filePath);
        } catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
        }

    }
}
```

---

## 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.*;
import java.util.StringTokenizer;

public class StringBuilderStringBufferFileReaderInputStreamReader {
    // StringBuilder vs StringBuffer
    public static void StringBuilderVsStringBuffer() {
        String text = "DummyText";
```

```
        // for StringBuilder
        long startTime = System.nanoTime();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 1000000; i++) {
            sb.append(text);
        }
        long endTime = System.nanoTime();
        System.out.println("StringBuilder Time: " + ((endTime -
startTime)/100000) + " ms");

        // StringBuffer Performance
        startTime = System.nanoTime();
        StringBuffer sbf = new StringBuffer();
        for (int i = 0; i < 1000000; i++) {
            sbf.append(text);
        }
        endTime = System.nanoTime();
        System.out.println("StringBuffer Time: " + ((endTime -
startTime)/100000) + " ms");
    }

    // Method to read file using FileReader
    public static void readFileWithFileReader(String filePath) {
        long startTime = System.nanoTime();
        int wordCount = 0;

        try {
            FileReader fr = new FileReader(filePath);
            BufferedReader br = new BufferedReader(fr);
            String line;
            while ((line = br.readLine()) != null) {
                wordCount += new StringTokenizer(line).countTokens();
            }

            br.close();

        } catch (IOException e) {
            e.printStackTrace();
        }

        long endTime = System.nanoTime();
        System.out.println("FileReader - Word Count: " + wordCount + ", Time: "
+ ((endTime - startTime)/100000) + " ms");
    }
```

```java
    public static void readFileWithInputStreamReader(String filePath) {
        long startTime = System.nanoTime();
        int wordCount = 0;

        try {
            InputStreamReader isr = new InputStreamReader(new
FileInputStream(filePath));
            BufferedReader br = new BufferedReader(isr);
            String line;
            while ((line = br.readLine()) != null) {
                wordCount += new StringTokenizer(line).countTokens();
            }

            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        long endTime = System.nanoTime();
        System.out.println("InputStreamReader - Word Count: " + wordCount + ",
Time: " + ((endTime - startTime)/100000) + " ms");
    }

    public static void main(String[] args) {
        // StringBuilder vs StringBuffer
        StringBuilderVsStringBuffer();


        String filePath = "large_file.txt"; //1oomb

        // Test FileReader vs InputStreamReader
        readFileWithFileReader(filePath);
        readFileWithInputStreamReader(filePath);
    }
}
```

## 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.

```java
import java.util.*;

public class FirstNegativeNumber {

    public static int findIdx(int arr[]) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] < 0) {
                return i;
            }
        }

        return -1;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of array:");
        int n = sc.nextInt();

        int arr[] = new int[n];

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

        int idx = findIdx(arr);
        if (idx == -1) {
            System.out.println("No negative integer found!");
        } else {
```

```
            System.out.println("First negative integer found at index " + idx);
        }

        sc.close();
    }

}
```

---

### 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.*;

public class SearchWord {

    public static String findSentence(String arr[], String word) {
        for (int i = 0; i < arr.length; i++) {
            String sentence = arr[i];
            String words[] = sentence.split(" ");
            for (int j = 0; j < words.length; j++) {
                if (words[j].equals(word)) {
                    return sentence;
                }
            }
        }

        return null;
    }
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of array:");
        int n = sc.nextInt();

        String arr[] = new String[n];

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

        System.out.println("Enter a word to find first sentence containing
it:");
        String word = sc.next();

        String sentence = findSentence(arr, word);
        if (sentence == null) {
            System.out.println("No sentence found containing \"" + word+ "\"");
        } else {
            System.out.println("Sentence found: " + sentence);
        }

        sc.close();
    }
}
```

## 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).

```java
public class RotatedSortedArray {

    public static int findSmallestIdx(int arr[]) {
        int low = 0;
        int high = arr.length - 1;
        int mid = (low + high) / 2;

        while (low < high) {
            mid = (low + high) / 2;
            if (arr[mid] < arr[high]) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        return low;
    }

    public static void main(String[] args) {
        int arr[] = { 4, 5, 6, 7, 1, 3 }; //dummy array

        System.out.println(findSmallestIdx(arr));
    }
}
```

## 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.

```
public class FindPeakElement {

    public static int findPeakIdx(int arr[]) {
        int left = 0;
        int right = arr.length - 1;

        while (left < right) {
            int mid = (left + right) / 2;
            if (arr[mid] > arr[mid + 1] && arr[mid] > arr[mid - 1]) {
                return mid;
            }
            if (arr[mid] < arr[mid - 1]) {
                right = mid - 1;
            }
            if (arr[mid] < arr[mid + 1]) {
                left = mid + 1;
            }
        }

        return -1;
    }
```

```
    public static void main(String[] args) {
        int arr[] = { 1, 2, 3, 2, 1 }; //dummy

        System.out.println(findPeakIdx(arr));
    }
}
```

---

## 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
public class SearchIn2DMatrix {
    public static boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
            return false;

        int rows = matrix.length;
        int cols = matrix[0].length;
        int left = 0;
        int right = rows * cols - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int row = mid / cols, col = mid % cols;

            if (matrix[row][col] == target)
                return true;
            else if (matrix[row][col] < target)
                left = mid + 1;
            else
                right = mid - 1;
        }

        return false;
    }

    public static void main(String[] args) {
        int[][] matrix = {
                { 1, 3, 5 },
                { 7, 10, 11 },
                { 12, 14, 18 }
        };
        int target = 14;
        System.out.println(searchMatrix(matrix, target));
    }
}
```

## 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`.

```
public class FirstAndLastOccurrence {

    public static int findFirstOcc(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        int first = -1;

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

        return first;
    }

    public static int findLastOcc(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        int last = -1;
```

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

        return last;
    }

    public static int[] searchRange(int[] nums, int target) {
        return new int[] { findFirstOcc(nums, target), findLastOcc(nums,
target) };
    }

    public static void main(String[] args) {
        int[] nums = { 1, 2, 2, 2, 3, 4, 5 };
        int target = 2;
        int[] ans = searchRange(nums, target);
        System.out.println("First occurrence: " + ans[0] + ", Last occurrence:
" + ans[1]);
    }
}
```

---

## 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`.

```
import java.util.*;
public class LinearBinary {

    public static int findFirstMissingPositive(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] != nums[i])
{
                int temp = nums[i];
                nums[i] = nums[temp - 1];
                nums[temp - 1] = temp;
            }
        }
        for (int i = 0; i < n; i++) {
            if (nums[i] != i + 1) return i + 1;
        }
        return n + 1;
    }

    public static int binarySearch(int[] nums, int target) {
        Arrays.sort(nums);
        int left = 0, right = nums.length - 1;

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

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

        return -1;
    }

    public static void main(String[] args) {
        int[] nums = {3, 4, -1, 1};
        System.out.println("First Missing Positive: " +
findFirstMissingPositive(nums));

        int[] sortedArray = {1, 2, 3, 4, 5};
        int target = 3;
        System.out.println("Index of target: " + binarySearch(sortedArray,
target));
    }
}
```