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.*;

class ReverseAString{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        String str = sc.next();

        StringBuilder st1 = new StringBuilder();

        st1.append(str);

        str = st1.reverse().toString();

        System.out.println(str);
    }


}
```

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.

```java
import java.util.*;

class RemoveDuplicates{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        String str = sc.next();

        HashSet<Character> hs = new HashSet<>();
        StringBuilder stb1 = new StringBuilder();

        for(int i=0;i<str.length();i++){

            if(!hs.contains(str.charAt(i))){
                stb1.append(str.charAt(i)+"");
                hs.add(str.charAt(i));
            }

        }

        str = stb1.toString();
        System.out.println(str);

    }


}
```

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.*;

class Concatenate{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();
        String[] arr = new String[n];

        for(int i=0;i<n;i++)
            arr[i] = sc.next();

        StringBuffer stb1 = new StringBuffer();

        for(int i=0;i<n;i++)
            stb1.append(arr[i]+" ");

        String str = stb1.toString();
        System.out.println(str);

    }

}
```

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.

```
import java.util.*;

class CompareBufferWithBuilder{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);
        String str = sc.next();

        StringBuilder stb1 = new StringBuilder();
        StringBuffer stb2 = new StringBuffer();

        long ini = System.nanoTime();

        for(int i=0;i<1000000;i++){
            stb1.append(str);
        }

        System.out.println("Execution Time for String Builder is :
"+(System.nanoTime()-ini)+" nano seconds");

        ini = System.nanoTime();

        for(int i=0;i<1000000;i++){
            stb2.append(str);
        }
```

```
        System.out.println("Execution Time for String Buffer is : "+(System.nanoTime()-ini)+" nano seconds");

    }

}
```

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.*;
import java.util.*;

class ReadAFileUsingFileReader{

    public static void main(String[] args){
        try{
            FileReader f1 = new FileReader("LinearAndBinarySearch.java");
            BufferedReader b1 = new BufferedReader(f1);

            while(true){

                String str = b1.readLine();
                if(str == null)
                    break;
                System.out.println(str);

            }
            f1.close();
            b1.close();
        }
```

```
        catch(FileNotFoundException e){
            System.out.println(e);
        }
        catch(IOException e){
            System.out.println(e);
        }
    }


}
```

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.*;
import java.util.*;

class WordCountInAFile{

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

        String word = sc.next();
        int c = 0;

        try{
            FileReader f1 = new FileReader("LinearAndBinarySearch.java");
            BufferedReader b1 = new BufferedReader(f1);
```

```
            while(true){

                String str = b1.readLine();
                if(str == null)
                    break;

                String str2 = "";
                for(int i=0;i<str.length();i++){
                    if((str.charAt(i) >='a' &&
str.charAt(i)<='z')||(str.charAt(i) >='A' && str.charAt(i)<='Z'))
                        str2 += str.charAt(i);
                    else{
                        if(str2.equals(word))
                            c++;
                        str2 = "";
                    }
                }
                if(str2.equals(word))
                    c++;

            }
            f1.close();
            b1.close();
            System.out.println("Number of occurrences of word "+word+" is :
"+c);
        }
        catch(FileNotFoundException e){
            System.out.println(e);
        }
        catch(IOException e){
            System.out.println(e);
        }
    }


}
```

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.*;

class ReadAFileUsingInputStreamReader{

    public static void main(String[] args){
        try{
            FileInputStream f1 = new
FileInputStream("LinearAndBinarySearch.java");
            InputStreamReader i1 = new InputStreamReader(f1);
            BufferedReader b1 = new BufferedReader(i1);

            while(true){

                String str = b1.readLine();
                if(str == null)
                    break;
                System.out.println(str);

            }
            f1.close();
            i1.close();
            b1.close();
        }
        catch(FileNotFoundException e){
            System.out.println(e);
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
```

```
}
```

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.*;
import java.util.*;

class ReadUserInputAndWriteToFile{

    public static void main(String[] args){
        try{
            InputStreamReader i1 = new InputStreamReader(System.in);
            BufferedReader b1 = new BufferedReader(i1);

            FileWriter writer = new FileWriter("name.txt");

            System.out.println("Enter any text : ");
            String str = b1.readLine();

            writer.write(str);

            i1.close();
            writer.close();
```

```
            b1.close();
        }
        catch(FileNotFoundException e){
            System.out.println(e);
        }
        catch(IOException e){
            System.out.println(e);
        }
    }


}
```

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

class CompareAll{
```

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

        String str1 = "hello";

        StringBuilder stb1 = new StringBuilder();
        StringBuffer stb2 = new StringBuffer();

        long ini = System.nanoTime();

        for(int i=0;i<1000000;i++){
            stb1.append(str1);
        }

        System.out.println("Execution Time for String Builder is : 
"+(System.nanoTime()-ini)+" nano seconds");

        ini = System.nanoTime();

        for(int i=0;i<1000000;i++){
            stb2.append(str1);
        }

        System.out.println("Execution Time for String Buffer is : 
"+(System.nanoTime()-ini)+" nano seconds");


        try{
            FileReader f1 = new FileReader("100mbfile.txt");
            BufferedReader b1 = new BufferedReader(f1);

            FileInputStream fi1 = new FileInputStream("100mbfile.txt");
            InputStreamReader i1 = new InputStreamReader(fi1);
            BufferedReader b2 = new BufferedReader(i1);

            int c1 = 0,c2 = 0;
            ini = System.nanoTime();
            while(true){

                String str = b1.readLine();
                if(str == null)
                    break;

                for(int i=0;i<str.length();i++){
                    if(str.charAt(i) == ' ')
                        c1++;
```

```
                }

            }
            System.out.println("Word count is "+c1);
            System.out.println("Time of execution for File Reader is 
"+(System.nanoTime()-ini)+" nano seconds");

            ini = System.nanoTime();
            while(true){

                String str = b2.readLine();
                if(str == null)
                    break;

                for(int i=0;i<str.length();i++){
                    if(str.charAt(i) == ' ')
                        c2++;
                }

            }
            System.out.println("Word count is "+c2);
            System.out.println("Time of execution for InputStreamReader is 
"+(System.nanoTime()-ini)+" nano seconds");
        }
        catch(FileNotFoundException e){
            System.out.println(e);
        }
        catch(IOException e){
            System.out.println(e);
        }

    }

}
```

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.*;

class FirstNegativeNumber{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        int[] arr = new int[n];

        for(int i=0;i<n;i++)
            arr[i] = sc.nextInt();

        System.out.println("Index of First Negative element is : "+linearSearch(arr));

    }

    public static int linearSearch(int[] arr){

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

        return -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.*;

class WordSearch{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        List<String> l1 = new ArrayList<>();

        while(true){
            System.out.println("Enter a sentence : (or type end to quit 
adding)");
            String sentence = sc.nextLine();


            if(sentence.equals("end"))
                break;

            l1.add(sentence);


        }

        System.out.println("Enter the target word : ");
        String target = sc.next();

        boolean found = false;

        for(int i=0;i<l1.size();i++){
```

```
            StringBuilder str = new StringBuilder("");

            for(int j = 0;j<l1.get(i).length();j++){
                if(l1.get(i).charAt(j) != ' ')
                    str.append(l1.get(i).charAt(j));
                else{
                    if(str.toString().equals(target)){
                        System.out.println(l1.get(i));
                        found = true;
                        break;
                    }
                    str.setLength(0);
                }

            }
            if(str.toString().equals(target)){
                System.out.println(l1.get(i));
                found = true;
                break;
            }
            if(found)
                break;

        }
        if(!found)
            System.out.println("Cannot find the given word");


    }


}
```

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
import java.util.*;

class RotatedSortedArray{

    public static void main(String[] args){

        int[] arr = {5,6,7,8,9,10,1,2,3};

        int min = binarySearch(arr);

        System.out.println(min);

    }

    public static int binarySearch(int[] arr){

        int low = 0, high = arr.length-1;

        while(low < high){

            int mid = (low+high)/2;

            if(arr[mid] > high)
                low = mid+1;
            else
                high = mid;
        }

        return arr[low];

    }
```

```
}
```

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.

```
import java.util.*;

class PeakElement{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        int[] arr = new int[n];

        for(int i=0;i<n;i++)
            arr[i] = sc.nextInt();

        System.out.println(peakSearch(arr));

    }
```

```
    public static int peakSearch(int[] arr){

        int low = 0, high = arr.length-1;

        while(low<=high){

            int mid = (low+high)/2;

            if((mid-1) > 0 && (mid+1) < arr.length && arr[mid] > arr[mid+1] &&
arr[mid] > arr[mid-1])
                return arr[mid];

            if(mid-1<0 || mid+1>=arr.length)
                return arr[mid];

            else if(mid+1<arr.length && arr[mid]<arr[mid+1])
                low = mid+1;

            else
                high = mid-1;

        }

        return -1;
    }

}
```

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`.

```
import java.util.*;

class SearchIn2DMatrix{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();
        int m = sc.nextInt();

        int[][] matrix = new int[n][m];

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++)
                matrix[i][j] = sc.nextInt();

        }

        int target = sc.nextInt();

        System.out.println(binarySearch(matrix, target));
    }

    public static boolean binarySearch(int[][] matrix, int target){

        int row = 0;
        int column = matrix[0].length-1;

        while(row < matrix.length && column >= 0){

            if(matrix[row][column] == target)
```

```
                return true;

            if(matrix[row][column] < target)
                row++;
            else
                column--;

        }
        return false;
    }

}
```

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.*;

class FirstAndLastOccurrence{

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

```

```java
        int n = sc.nextInt();
        int[] arr = new int[n];

        for(int i=0;i<n;i++)
            arr[i] = sc.nextInt();

        int target = sc.nextInt();

        System.out.println("First occurence is : "+firstOccurence(arr,
target));
        System.out.println("Last occurrence is : "+lastOccurence(arr, target));
    }

    public static int firstOccurence(int[] arr, int target){

        int low = 0, high = arr.length-1;
        int first = -1;

        while(low <= high){

            int mid = (low+high)/2;

            if(arr[mid] == target){
                first = mid;
                high = mid - 1;
            }
            else if(arr[mid] < target)
                low = mid + 1;
            else
                high = mid - 1;

        }

        return first;

    }
    public static int lastOccurence(int[] arr, int target){

        int low = 0, high = arr.length-1;
        int last = -1;

        while(low <= high){

            int mid = (low+high)/2;
```

```
            if(arr[mid] == target){
                last = mid;
                low = mid + 1;
            }
            else if(arr[mid] < target)
                low = mid + 1;
            else
                high = mid - 1;

        }

        return 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`.

```
import java.util.*;

class LinearAndBinarySearch{

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

```
        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();
        int[] arr = new int[n];

        for(int i=0;i<n;i++)
            arr[i] = sc.nextInt();

        int target = sc.nextInt();

        System.out.println("First missing positive integer is :
"+firstMissing(arr));

        System.out.println("Index of the target element is :
"+binarySearch(arr, target));

    }

    public static int firstMissing(int[] arr){

        int arr2[] = new int[arr.length];

        for(int i=0;i<arr.length;i++){
            int ind = arr[i]-1;
            if(ind>=0 && ind<arr.length){
                arr2[ind] = -1;
            }
        }

        for(int i=0;i<arr.length;i++){
            if(arr2[i] >= 0)
                return i+1;
        }

        return -1;

    }

    public static int binarySearch(int[] arr, int target){


        HashMap<Integer, Integer> hm = new HashMap<>();

        for(int i=0;i<arr.length;i++)
```

```
            hm.put(arr[i], i);

        Arrays.sort(arr);

        int low = 0, high = arr.length-1;

        while(low<=high){

            int mid = high - (high - low)/2;

            if(arr[mid] == target)
                return hm.get(arr[mid]);
            else if(arr[mid] > target)
                high = mid - 1;
            else
                low = mid + 1;
        }
        return -1;

    }

}
```