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.

Sol:
```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a string:");
        String input = sc.nextLine();
        StringBuilder sb = new StringBuilder();
        sb.append(input);
        sb.reverse();
        String reversed = sb.toString();
        System.out.println("Reversed string: " + reversed);
    }
}
```

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.

Sol:
```
import java.util.Scanner;
import java.util.HashSet;
public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter a string:");
      String input = sc.nextLine();
      StringBuilder sb = new StringBuilder();
      HashSet<Character> seen = new HashSet<>();
      for (int i = 0; i < input.length(); i++) {
         char ch = input.charAt(i);
         if (!seen.contains(ch)) {
            sb.append(ch);
            seen.add(ch);
         }
      }
      System.out.println("String after removing duplicates: " + sb.toString());
   }
}
```

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.

Sol:
```
public class Main {
   public static void main(String[] args) {
      String[] words = {"Hello", "World", "From", "Java"};
```

```
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < words.length; i++) {
            sb.append(words[i]);
            if (i < words.length - 1) {
                sb.append(" ");
            }
        }
        String result = sb.toString();
        System.out.println(result);
    }
}
```

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.

Sol:

```
public class Main {
    public static void main(String[] args) {
        int iterations = 1000000;
        long startTime1 = System.nanoTime();
        StringBuffer stringBuffer = new StringBuffer();
        for (int i = 0; i < iterations; i++) {
            stringBuffer.append("hello");
        }
        long endTime1 = System.nanoTime();
        long durationBuffer = endTime1 - startTime1;
        long startTime2 = System.nanoTime();
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0; i < iterations; i++) {
```

```
            stringBuilder.append("hello");
        }
        long endTime2 = System.nanoTime();
        long durationBuilder = endTime2 - startTime2;
        System.out.println("Time taken by StringBuffer: " + durationBuffer + " ns");
        System.out.println("Time taken by StringBuilder: " + durationBuilder + " 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.

Sol:
```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
public class Main {
    public static void main(String[] args) {
        BufferedReader reader = null;
        try {
            FileReader fileReader = new FileReader("input.txt");
            reader = new BufferedReader(fileReader);
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("An error occurred while reading the file.");
            e.printStackTrace();
        } finally {
            try {
                if (reader != null) {
                    reader.close();
```

```
            }
        } catch (IOException e) {
            System.out.println("An error occurred while closing the file.");
            e.printStackTrace();
        }
    }
  }
}
```

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.

Sol:

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
public class Main {
    public static void main(String[] args) {
        BufferedReader reader = null;
        try {
            FileReader fileReader = new FileReader("input.txt");
            reader = new BufferedReader(fileReader);
            String targetWord = "java";
            int count = 0;
            String line;
            while ((line = reader.readLine()) != null) {
                String[] words = line.split("\\s+");
                for (String word : words) {
                    if (word.equals(targetWord)) {
                        count++;
                    }
```

```
            }
        }
        System.out.println("The word \"" + targetWord + "\" appears " + count + " times in the
file.");
    } catch (IOException e) {
        System.out.println("An error occurred while reading the file.");
        e.printStackTrace();
    } finally {
        try {
            if (reader != null) {
                reader.close();
            }
        } catch (IOException e) {
            System.out.println("An error occurred while closing the file.");
            e.printStackTrace();
        }
    }
  }
}
```

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.

Sol:

```
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
public class Main {
  public static void main(String[] args) {
```

```
        BufferedReader reader = null;
        try {
            FileInputStream fileInputStream = new FileInputStream("input.txt");
            InputStreamReader inputStreamReader = new InputStreamReader(fileInputStream,
"UTF-8");
            reader = new BufferedReader(inputStreamReader);
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("An error occurred while reading the file.");
            e.printStackTrace();
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (IOException e) {
                System.out.println("An error occurred while closing the file.");
                e.printStackTrace();
            }
        }
    }
}
```

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.

Sol:

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;
public class Main {
    public static void main(String[] args) {
        BufferedReader reader = null;
        FileWriter writer = null;
        try {
            InputStreamReader inputStreamReader = new InputStreamReader(System.in);
            reader = new BufferedReader(inputStreamReader);
            writer = new FileWriter("output.txt");
            String line;
            while (true) {
                line = reader.readLine();
                if (line.equalsIgnoreCase("exit")) {
                    break;
                }
                writer.write(line + System.lineSeparator());
            }
            System.out.println("Input has been written to the file.");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                }
                if (writer != null) {
                    writer.close();
                }
            } catch (IOException e) {
                System.out.println("An error occurred while closing the resources.");
                e.printStackTrace();
            }
        }
    }
}
```

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.

Sol:

```
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
public class Main {
   public static void main(String[] args) {
      long startTime, endTime;
      String word = "hello";
      startTime = System.currentTimeMillis();
      StringBuilder sb = new StringBuilder();
      for (int i = 0; i < 1000000; i++) {
         sb.append(word);
      }
      endTime = System.currentTimeMillis();
      System.out.println("StringBuilder time: " + (endTime - startTime) + " ms");
      startTime = System.currentTimeMillis();
      StringBuffer sbf = new StringBuffer();
      for (int i = 0; i < 1000000; i++) {
         sbf.append(word);
      }
      endTime = System.currentTimeMillis();
      System.out.println("StringBuffer time: " + (endTime - startTime) + " ms");
      BufferedReader reader = null;
      try {
```

```
            startTime = System.currentTimeMillis();
            FileReader fileReader = new FileReader("largefile.txt");
            reader = new BufferedReader(fileReader);
            int wordCount = 0;
            String line;
            while ((line = reader.readLine()) != null) {
                String[] words = line.split("\\s+");
                wordCount += words.length;
            }
            endTime = System.currentTimeMillis();
            System.out.println("FileReader word count: " + wordCount);
            System.out.println("FileReader time: " + (endTime - startTime) + " ms");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        reader = null;
        try {
            startTime = System.currentTimeMillis();
            InputStreamReader inputStreamReader = new InputStreamReader(new
FileReader("largefile.txt"), "UTF-8");
            reader = new BufferedReader(inputStreamReader);
            int wordCount = 0;
            String line;
            while ((line = reader.readLine()) != null) {
                String[] words = line.split("\\s+");
                wordCount += words.length;
            }
            endTime = System.currentTimeMillis();
            System.out.println("InputStreamReader word count: " + wordCount);
            System.out.println("InputStreamReader time: " + (endTime - startTime) + " ms");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (reader != null) {
                    reader.close();
```

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

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.

Sol:

```
import java.util.Scanner;

public class Main {

  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 index = -1;

    for (int i = 0; i < n; i++) {

      if (arr[i] < 0) {
```

```
            index = i;

            break;

          }

        }

        System.out.println(index);

    }

}
```

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".

Sol:

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();

        sc.nextLine();

        String[] sentences = new String[n];

        for (int i = 0; i < n; i++) {
```

```
            sentences[i] = sc.nextLine();
        }
        String word = sc.nextLine();
        String result = "Not Found";
        for (int i = 0; i < n; i++) {
            if (sentences[i].contains(word)) {
                result = sentences[i];
                break;
            }
        }
        System.out.println(result);
    }
}
```

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).

Sol:

```java
import java.util.Scanner;

public class Main {

    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 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(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.

Sol:

```
import java.util.Scanner;

public class Main {

    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 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(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`.

Sol:

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

```
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int rows = sc.nextInt();
        int cols = sc.nextInt();
        int[][] matrix = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }
        int target = sc.nextInt();
        int left = 0, right = rows * cols - 1;
        boolean found = false;
        while (left <= right) {
            int mid = (left + right) / 2;
            int row = mid / cols, col = mid % cols;
            if (matrix[row][col] == target) {
                found = true;
                break;
            } else if (matrix[row][col] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
```

```
        }
      }
      System.out.println(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`.

Sol:

```
import java.util.Scanner;

public class Main {

    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++) {
```

```java
            arr[i] = sc.nextInt();
        }
        int target = sc.nextInt();
        int first = findFirstOccurrence(arr, target);
        int last = findLastOccurrence(arr, target);
        System.out.println(first + " " + last);
    }
    public static int findFirstOccurrence(int[] arr, int target) {
        int left = 0, right = arr.length - 1, result = -1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] == target) {
                result = mid;
                right = mid - 1;
            } else if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return result;
    }
    public static int findLastOccurrence(int[] arr, int target) {
        int left = 0, right = arr.length - 1, result = -1;
```

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

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`.

Sol:

```
import java.util.Arrays;

import java.util.Scanner;

public class Main {

    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();

        int firstMissingPositive = findFirstMissingPositive(arr);

        System.out.println("First Missing Positive: " + firstMissingPositive);

        Arrays.sort(arr);

        int targetIndex = binarySearch(arr, target);

        System.out.println("Target Index: " + targetIndex);

    }

    public static int findFirstMissingPositive(int[] arr) {

        int n = arr.length;

        for (int i = 0; i < n; i++) {

            if (arr[i] <= 0) {

                arr[i] = n + 1;

            }
```

```
    }
    for (int i = 0; i < n; i++) {
        int absVal = Math.abs(arr[i]);
        if (absVal <= n && arr[absVal - 1] > 0) {
            arr[absVal - 1] = -arr[absVal - 1];
        }
    }
    for (int i = 0; i < n; i++) {
        if (arr[i] > 0) {
            return i + 1;
        }
    }
    return n + 1;
}
public static int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
```

```
            }
        }
        return -1;
    }
}
```