Complex Problems for Fast Learners

Student Name: Nitakshi Azad	**UID:**22BCS10900
Branch: CSE	**Section/Group:** KRG_IOT_2(B)
Semester: 6th	**Date of Performance:** 22/04/2025
Subject Name: PBLJ	**Subject Code:** 22CSH-359

Problem -1

1. Aim: Consider a function **public String matchFound(String input 1, String input 2),** where

- **input1** will contain only a single word with only 1 character replaces by an underscore '_'
- **input2** will contain a series of words separated by colons and no space character in between
- **input2** will not contain any other special character other than underscore and alphabetic characters.

The methods should return output in a String type variable **"output1"** which contains all the words from input2 separated by colon which matches with input 1. All words in output1 should be in uppercase.

2. Objective:

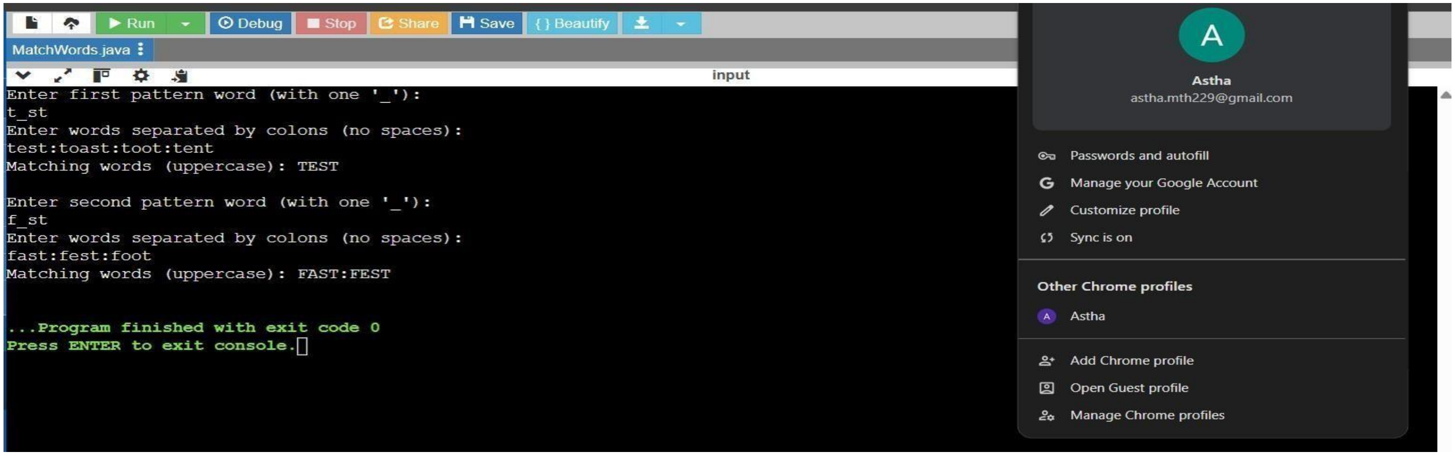
- To implement a method that identifies matching words from a pattern containing a single underscore.
- To understand how to use string splitting and comparison in Java.
- To practice string manipulation techniques like converting to uppercase and adding delimiters.
- To learn how to compare characters while handling special pattern symbols.
- To build a simple and interactive Java program using user input with the Scanner class.

3. Implementation/Code:

```
import java.util.Scanner;
public class MatchWords {
public String matchFound(
String input1, String input2) {
  String[] wordList = input2.split(":");
  String output1 = "";
  for (String word : wordList)      {
```

```
if (word.length() != input1.length()) {
continue;
}
boolean isMatch = true;
for (int i = 0; i < input1.length(); i++) {
if (input1.charAt(i) != '_' && input1.charAt(i) != word.charAt(i)) {
isMatch = false;
break;
}
} if (isMatch)
{  if (!output1.equals("")) {
output1 += ":";
}
output1 += word.toUpperCase();
}  }
return output1;
}
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        MatchWords obj = new MatchWords();
        System.out.println("Enter first pattern word (with one '_'):");
        String input1_1 = sc.nextLine();
        System.out.println("Enter words separated by colons (no spaces):");
        String input2_1 = sc.nextLine();
        String result1 = obj.matchFound(input1_1, input2_1);
        System.out.println("Matching words (uppercase): " + result1);
        System.out.println("\nEnter second pattern word (with one '_'):");
        String input1_2 = sc.nextLine();
        System.out.println("Enter words separated by colons (no spaces):");  String
        input2_2 = sc.nextLine();
        String    result2 =        obj.matchFound(input1_2,     input2_2);
        System.out.println("Matching words (uppercase): " + result2);
        sc.close();
     }
  }
```

Output: -



```
Enter first pattern word (with one '_'):
t_st
Enter words separated by colons (no spaces):
test:toast:toot:tent
Matching words (uppercase): TEST

Enter second pattern word (with one '_'):
f_st
Enter words separated by colons (no spaces):
fast:fest:foot
Matching words (uppercase): FAST:FEST


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 1

LearningOutcomes:-

- Learnt how to split a string using a specific delimiter like a colon.
- Understood how to compare each character of two strings with a condition.
- Gained hands-on experience with converting strings to uppercase using toUpperCase().
- Practiced building conditional logic with loops and flags (isMatch).
- Became familiar with basic input/output operations in Java using the Scanner class.

Problem-2

1. **Aim:** Given a String (In Uppercase alphabets or Lowercase alphabets), new alphabets is to be appended with following rule:

   1) If the alphabet is present in the input string, use the numeric value of that alphabet. E.g. a or A numeric value is 1 and so on. New alphabet to be appended between 2 alphabets:

      a) If (sum of numeric value of 2 alphabets) %26 is 0, then append 0. E.g. string is ay. Numeric value of a is 1, y is 25. Sum is 26. Remainder is 0, the new string will be a0y.

      b) Otherwise (sum of numeric value of 2 alphabets) %26 numeric value alphabet is to be appended. E.g. ac is string. Numeric value of a is 1, c is 3, sum is 4. Remainder with 26 is 4. Alphabet to be appended is d. output will be adc.

   2) If a digit is present, it will be the same in the output string. E.g. string is 12, output string is 12.

   3) If only a single alphabet is present, it will be the same in the output string. E.g. input string is 1a, output will be 1a.

   4) If space is present, it will be the same in the output string. E.g. string is ac 12a, output will be adc 12a.

   Constraint: Whether string alphabets are In Uppercase or Lowercase, appended alphabets must be in lower case. Output string must also be in lowercase.

2. Objectives:

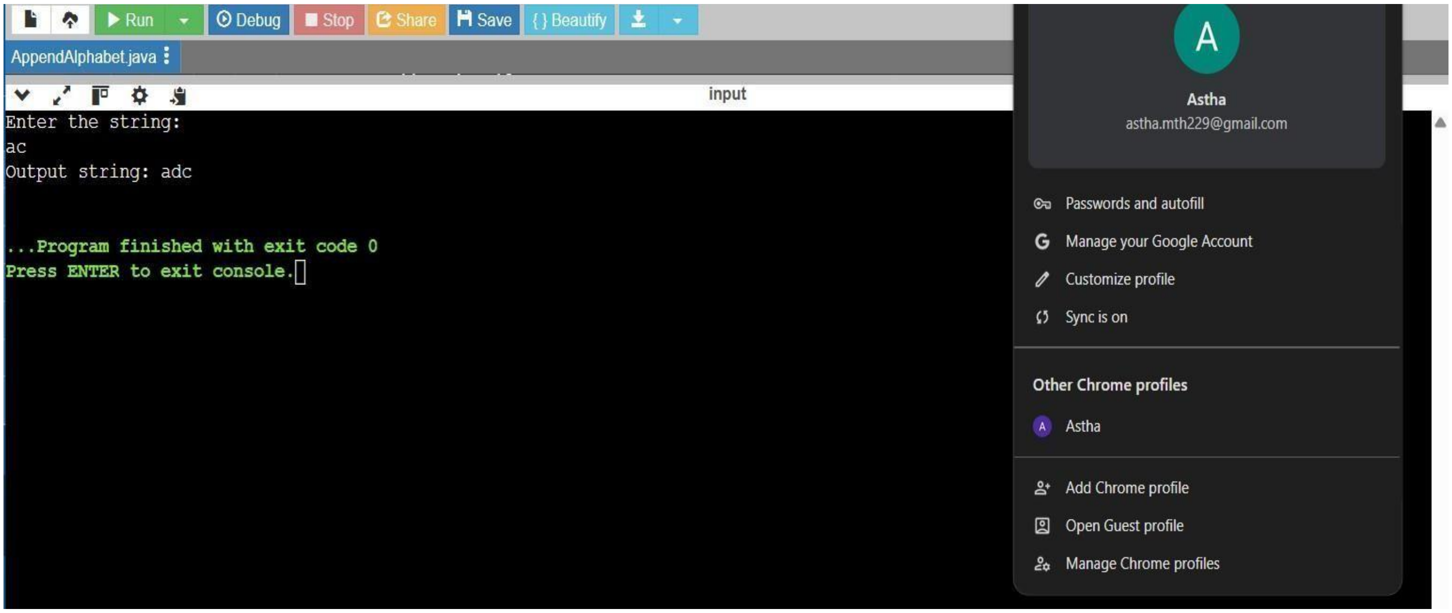
- To implement a logic that inserts new characters between alphabets based on numeric value rules. This helps in strengthening string manipulation and character arithmetic in Java.
- To create a function that handles both letters and digits in a string appropriately. The function must preserve digits and spaces, and handle letters in a case-insensitive manner.
- To understand and apply ASCII-based arithmetic to derive alphabet positions. This includes converting characters to their numeric values and vice versa.
- To ensure consistent lowercase formatting of the final output string. This enforces proper string casing rules regardless of input format.
- To build a Java program that accepts user input and generates a transformed output. This enhances skills in using Scanner and returning processed results.

3. Implementation/Code:

```
 import java.util.Scanner; public class AppendAlphabet {
 public static String processString(String input) {
 StringBuilder result = new StringBuilder();
 input = input.toLowerCase();
 for (int i = 0; i < input.length(); i++) {
 char ch1 = input.charAt(i);
 result.append(ch1);
 if (i + 1 < input.length())
 {
char ch2 = input.charAt(i + 1); if (Character.isLetter(ch1)
&& Character.isLetter(ch2)) {
int val1 = ch1 - 'a' + 1;
int val2 = ch2 - 'a' + 1;
int sum = val1 + val2;
if (sum % 26 == 0) {
result.append("0");
} else {
char mid = (char) ((sum % 26 - 1) + 'a');
result.append(mid);
```

```
                    }
                }
            }
        }
    return result.toString();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the string:");
        String input = sc.nextLine();
        String output = processString(input);
        System.out.println("Output string: " + output);
        sc.close();
    }
}
```

4. Output:



```
Enter the string:
ac
Output string: adc


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 2

5. Learning Outcomes:

- Learned how to loop through a string and access characters based on their positions. This includes comparing characters and accessing the next character in sequence.
- Gained understanding of converting characters to numeric positions using ASCII logic. For example, `'a'` is mapped to 1 using `(char - 'a' + 1)`.
- Practiced using conditional statements to insert characters or numbers dynamically. This improves control flow skills in Java based on custom conditions.
- Understood the importance of handling different character types like digits, spaces, and letters. Non-letter characters are left unchanged in the output for correctness.
- Developed the ability to construct strings dynamically using `StringBuilder`. This improves performance and efficiency in building strings in Java.

Problem – 3

1. **Aim:** String t is generated by random shuffling string s and then add one more letter at a random position. Return the letter that was added to t.

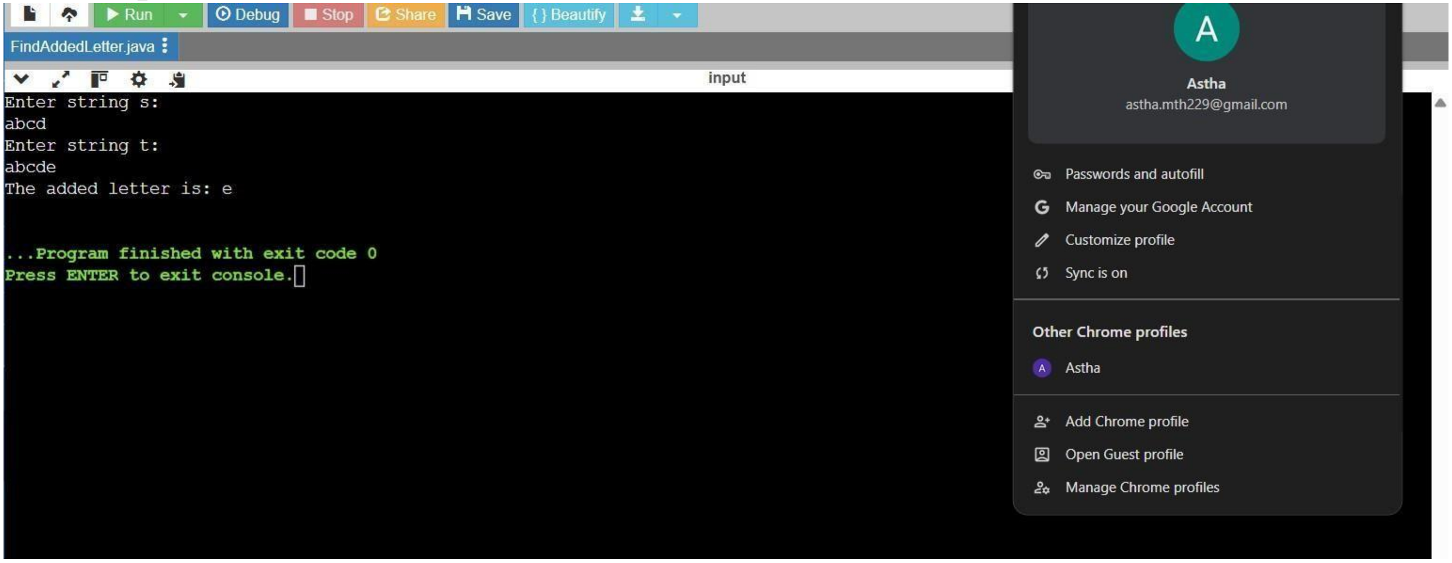
2. **Objectives:**

- To create a program that identifies an extra character added to a shuffled string. The task focuses on comparing two strings to find the difference.
- To practice converting strings into character arrays for easier processing. This supports iteration over individual characters of a string.
- To understand how to use ASCII values for solving character comparison problems. The difference in character sums helps in identifying the extra letter.
- To build logic that works regardless of the order of characters in the input. This promotes a logic-based rather than position-based comparison.
- To develop a Java application that takes user input and returns accurate results. It enhances hands-on experience with the `Scanner` class and method calls.

3. **Implementation/Code:**

```
import java.util.Scanner;
public class FindAddedLetter {
public static char findTheDifference(String s, String t) {
 int sumS = 0, sumT = 0;
 for (char ch : s.toCharArray()) {
 sumS += ch; }
 for (char ch : t.toCharArray()) {
 sumT += ch;
}
return (char) (sumT - sumS);
}
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
         System.out.println("Enter string s:");
         String s = sc.nextLine();
         System.out.println("Enter string t:");
         String t = sc.nextLine();
         char addedChar = findTheDifference(s, t);
         System.out.println("The added letter is: " + addedChar); sc.close();
    }
 }
```

4. Output:



```
Enter string s:
abcd
Enter string t:
abcde
The added letter is: e


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 3

5. Learning Outcomes

- Learned how to iterate over characters in a string using toCharArray() method. This enables processing each character individually for operations like summing.

- Gained understanding of how characters have numeric (ASCII) values in Java. Adding ASCII values of characters helps detect differences efficiently.
- Understood the concept of subtracting total values of strings to find the extra character. This technique is simple, fast, and avoids sorting or extra data structures.
- Practiced building compact and efficient logic using basic Java constructs. No need for complex algorithms—just loops and arithmetic.
- Enhanced skills in taking string inputs, invoking methods, and printing the result. Overall, this helps in developing clean and functional Java code.

Problem – 4

1. **Aim:** A string containing only parentheses is balanced if the following is true: 1. if it is an empty string 2. if A and B are correct, AB is correct, 3. if A is correct, (A) and {A} and [A] are also correct. Examples of some correctly balanced strings are: "{}()", "[{()}]", "({()})" Examples of some unbalanced strings are: "{}(", "({)}", "[[", "}{" etc. • Given a string, determine if it is balanced or not.

- Input Format: There will be multiple lines in the input file, each having a single non-empty string. You should read input till end-of-file.
- Output Format: For each case, print 'true' if the string is balanced, 'false' otherwise. • Sample Input: {}() ({()}) {}([] Sample Output: true true false true

2. **Objectives:**

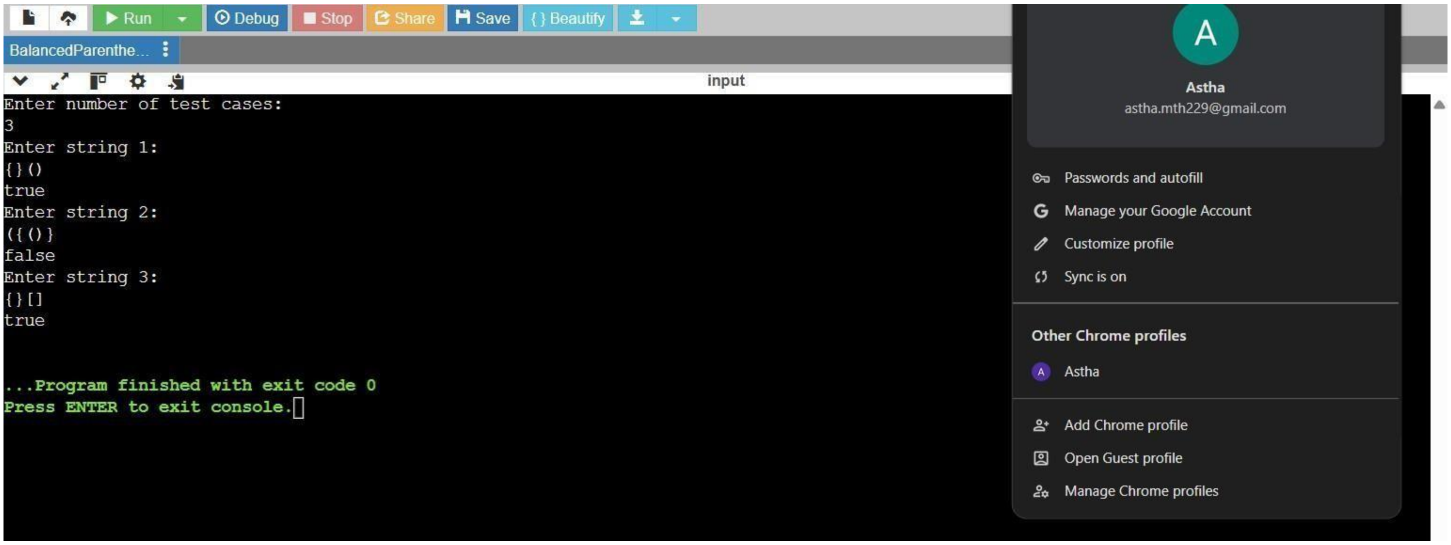
- To develop a Java program that checks if a string containing brackets is balanced. The goal is to verify proper nesting and pairing of parentheses, braces, and brackets.
- To implement stack-based logic for validating open and close brackets. This helps in understanding real-world use cases of stack data structures.
- To handle multiple test cases using loop constructs and user input. This improves ability to build reusable and dynamic logic.
- To correctly use conditionals for matching bracket types and validating structure. Ensures that each opening bracket has a corresponding and correctly placed closing one.
- To read input strings till end-of-file and generate appropriate boolean outputs. This objective emphasizes reading, processing, and responding to user input efficiently.

3. **Implementation/Code:**

```
import java.util.*;  public
  class BalancedParentheses { public static boolean
          isBalanced(String      str)      {
```

```
Stack<Character> stack = new Stack<>(); for
(char ch : str.toCharArray()) { if (ch == '(' ||
ch == '{' || ch == '[') { stack.push(ch);
    } else if (ch == ')' || ch == '}' || ch ==
      ']') { if (stack.isEmpty()) return
      false; char open = stack.pop();  if
      ((ch == ')' && open != '(') ||
        (ch == '}' && open != '{') || (ch
        == ']' && open != '[')) { return
        false;
      }
    }
  }  return
  stack.isEmpty();
}  public static void main(String[]
args) {
  Scanner    sc    =    new    Scanner(System.in);
  System.out.println("Enter number of test cases:");
  int n = Integer.parseInt(sc.nextLine()); for (int i =
  0; i < n; i++) {
    System.out.println("Enter string " + (i + 1) + ":");
    String input = sc.nextLine();
    System.out.println(isBalanced(input));
  }
  sc.close();
}
 }
```

4. Output:



```
Enter number of test cases:
3
Enter string 1:
{}()
true
Enter string 2:
({()}
false
Enter string 3:
{}[]
true


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 4

5. Learning Outcomes:

- Understood how stacks help in solving problems related to balanced expressions. Learned to push and pop elements for checking the latest unmatched opening bracket.
- Gained experience in comparing characters and applying logical operators. This reinforces conditional checking with multiple possible matches.
- Learned how to process multiple test cases using loops and control structures. Helps in handling repeated input-output operations effectively
- Strengthened skills in Java syntax for reading input and output formatting. Especially useful for competitive coding and real-time validation problems.
- Gained confidence in implementing core data structure concepts in practical scenarios. This includes problem solving using stacks and validating nested patterns.

Problem – 5

1. Aim: Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value. If target is not found in the array, return [-1, -1]. You must write an algorithm with O (log n) runtime complexity.

- Example 1:
- Input: nums = [5,7,7,8,8,10], target = 8

- Output: [3,4]
- Constraints: 0 <= nums.length <= 10^5 : -10^9 <= nums[i] <= $10^{\boxed{9}}$ nums is a non-decreasing array.
- -10^9 <= target <= 10^9

2. Objectives:

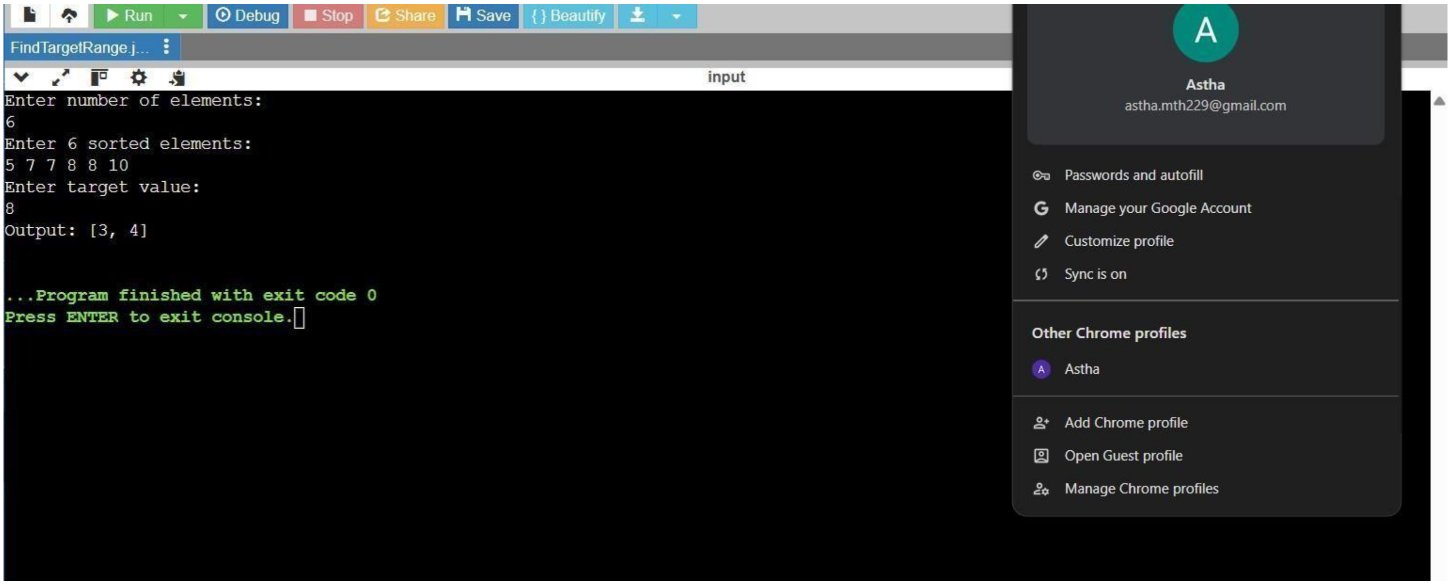
- To implement a binary search algorithm that finds the first and last position of a target. This helps in understanding efficient search techniques in sorted arrays.
- To ensure the solution runs in O(log n) time complexity as required. This promotes writing optimized and scalable code for large inputs.
- To practice breaking down problems into helper methods for clarity and reuse. Separate methods like `findFirst` and `findLast` make the logic easy to follow.
- To read array input from the user and apply binary search on user-defined values. This strengthens skills in dynamic input handling and real-time problem solving.
- To handle edge cases such as when the target is not found in the array. Ensures robustness and correctness of the program in all scenarios.

3. Implementation/Code:

```
import java.util.*;

        public class FindTargetRange {
           public static int[] searchRange(int[] nums, int target) {

              int[] result = new int[2]; result[0] = findFirst(nums,

              target); result[1] = findLast(nums, target); return

              result;

           }

           public static int findFirst(int[] nums, int target) { int

              left = 0, right = nums.length - 1;

              int index = -1;
while (left <= right) {

                 int mid = left + (right - left) / 2;
```

```
                if (nums[mid] == target) {

                index = mid; right = mid - 1;

                } else if (nums[mid] < target) { left

                   = mid + 1;

                } else {
 right = mid - 1;

                }

             }
return index;

           }

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

              index = -1;

              while (left <= right) {

                int mid = left + (right - left) / 2;

                if (nums[mid] == target) { index =

                mid; left = mid + 1;

                } else if (nums[mid] < target) { left

                   = mid + 1;

                } else {
 right = mid - 1;

                }
                }
    return index;

            }

            public static void main(String[] args) {

              Scanner    sc    =    new    Scanner(System.in);

              System.out.println("Enter number of elements:");

              int n = sc.nextInt();

              int[] nums = new int[n];

              System.out.println("Enter " + n + " sorted elements:");

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

              }

              System.out.println("Enter              target

              value:"); int target = sc.nextInt(); int[] result =

              searchRange(nums, target);

              System.out.println("Output: [" + result[0] + ", " + result[1] + "]"); sc.close();
            } }
```

4. Output:



```
Enter number of elements:
6
Enter 6 sorted elements:
5 7 7 8 8 10
Enter target value:
8
Output: [3, 4]


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 2

5. Learning Outcomes:

- Learned how binary search can be modified to find the first or last occurrence. This includes adjusting the search space even after finding the target.
- Gained practical experience in implementing logarithmic time search algorithms. Useful for solving problems efficiently in coding interviews and contests.
- Understood how to use indices to track positions and update based on comparisons. Helped in building confidence for writing condition-based search logic.
- Strengthened understanding of array traversal and working with loops and conditions. Essential for performing operations on sorted lists.
- Became familiar with writing modular, readable code using helper functions. Promotes clean coding habits and easier debugging in larger applications.