



**DEPARTMENT OF**

**COMPUTER SCIENCE & ENGINEERING**

### **Experiment 9 (ASSIGNMENT)**

**Student Name: Rakshit**

**Branch: BE-CSE**

**Semester: 6**

**Subject Name: PBLJ LAB**

**UID: 22BCS12726**

**Section/Group: IOT-631/B**

**Date of Performance: 27 March 2025**

**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 understand and implement string manipulation techniques in Java.
- To use loops and conditions effectively to filter words.
- To practice handling string search and replacement operations.
- To format output according to requirements.

#### 3) **Implementation/Code:**

```
public class MatchFinder {  
    public static String matchFound(String input1, String input2) {  
        String[] words = input2.split(":");  
        String regexPattern = input1.replace("_", "[a-zA-Z]");  
        StringBuilder output1 = new StringBuilder();  
  
        for (String word : words) {
```

```
        if (word.matches(regexPattern)) {
            if (output1.length() > 0) {
                output1.append(":");
            }
            output1.append(word.toUpperCase());
        }
    }

    return output1.toString();
}

public static void main(String[] args) {
    String input1 = "c_t";
    String input2 = "cat:cut:cot:bat:bot:dog";
    System.out.println(matchFound(input1, input2));
    "CAT:CUT:COT"

    input1 = "h_m";
    input2 = "ham:him:hum:hymn";
    System.out.println(matchFound(input1, input2));
    "HAM:HIM:HUM"
}
```

#### 4. Output

```
CAT:CUT:COT
HAM:HIM:HUM
```

#### 5. Learning Outcome:

- **String Manipulation:** Learned how to replace a character dynamically and match patterns.
- **Regular Expressions:** Used regex to match words with missing characters.
- **Loops and Conditions:** Used a loop to iterate over words and check for matches.
- **StringBuilder Usage:** Used StringBuilder to efficiently build the result string.

## PROBLEM 2

- 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.

**Hint: Input:** s = "abcd", t = "abcde" **Output:** "e" 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:

- Accept two strings 's' and 't'.
- Identify the extra character in 't'.
- Store the result in a String variable "output1".
- Implement case-insensitive word matching logic for output1.

### 3) Implementation/Code:

```
import java.util.*;

public class FindAddedLetter {

    public static String findAddedLetter(String s, String t) {
        int charSumS = 0, charSumT = 0;

        for (char c : s.toCharArray()) {
            charSumS += c;
        }

        for (char c : t.toCharArray()) {
            charSumT += c;
        }

        return String.valueOf((char) (charSumT - charSumS));
    }

    public static String matchWords(String input1, String input2) {
        Set<String> set = new HashSet<> (Arrays.asList(input1.toUpperCase().split("
"))));
```

```
StringBuilder output = new StringBuilder();

for (String word : input2.split(" ")) {
    if (set.contains(word.toUpperCase())) {
        if (output.length() > 0) output.append(":");
        output.append(word.toUpperCase());
    }
}

return output.toString();
}

public static void main(String[] args) {
    String s = "abcd";
    String t = "abcde";
    String addedLetter = findAddedLetter(s, t);
    System.out.println("Added Letter: " + addedLetter);

    String input1 = "apple banana orange";
    String input2 = "banana grape apple";
    String output1 = matchWords(input1, input2);
    System.out.println("Matched Words: " + output1);
}
}
```

#### 4) Output:

```
Added Letter: e
Matched Words: BANANA:APPLE
```

#### 5) Learning Outcome:

- Understanding character manipulation and ASCII value calculations.
- Implementing efficient string comparison techniques.
- Utilizing HashSet for fast lookups and comparisons.
- Practicing string operations such as splitting and joining.

### PROBLEM 3

**1) Aim:.** The next greater element of some element  $x$  in an array is the first greater element that is to the right of  $x$  in the same array.

You are given two distinct 0-indexed integer arrays `nums1` and `nums2`, where `nums1` is a subset of `nums2`.

For each  $0 \leq i < \text{nums1.length}$ , find the index  $j$  such that `nums1[i] == nums2[j]` and determine the next greater element of `nums2[j]` in `nums2`. If there is no next greater element, then the answer for this query is -1.

Return an array `ans` of length `nums1.length` such that `ans[i]` is the next greater element as described above.

**Hint:**

Input: `nums1 = [4,1,2]`, `nums2 = [1,3,4,2]`

Output: `[-1,3,-1]`

Explanation: The next greater element for each value of `nums1` is as follows:

- 4 is underlined in `nums2 = [1,3,4,2]`. There is no next greater element, so the answer is -1.

- 1 is underlined in `nums2 = [1,3,4,2]`. The next greater element is 3.

- 2 is underlined in `nums2 = [1,3,4,2]`. There is no next greater element, so the answer is -1.

**2) Objective:** Implement an optimized solution using stack and HashMap to determine the next greater element.

**3) Implementation/Code:**

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

    public static String findNextGreaterElements(int[] nums1, int[] nums2, String[]
input2) {
        Map<Integer, Integer> nextGreaterMap = new HashMap<>();
        Stack<Integer> stack = new Stack<>();
        for (int num : nums2) {
            while (!stack.isEmpty() && stack.peek() < num) {
                nextGreaterMap.put(stack.pop(), num);
            }
            stack.push(num);
        }
    }
}
```

```
while (!stack.isEmpty()) {
    nextGreaterMap.put(stack.pop(), -1);
}
int[] ans = new int[nums1.length];
for (int i = 0; i < nums1.length; i++) {
    ans[i] = nextGreaterMap.get(nums1[i]);
}
List<String> matchedWords = new ArrayList<>();
Set<String> numSet = new HashSet<>();
for (int num : nums1) {
    numSet.add(String.valueOf(num));
}

for (String word : input2) {
    if (numSet.contains(word)) {
        matchedWords.add(word.toUpperCase());
    }
}

String output1 = String.join(":", matchedWords);

System.out.println("Next Greater Elements: " + Arrays.toString(ans));
System.out.println("Output1: " + output1);

return output1;
}

public static void main(String[] args) {
    int[] nums1 = {4, 1, 2};
    int[] nums2 = {1, 3, 4, 2};
    String[] input2 = {"1", "3", "4", "2", "5"};

    findNextGreaterElements(nums1, nums2, input2);
}
}
```

#### 4) Output:

Next Greater Elements: [-1, 3, -1]

Output1: 1:4:2

#### 5) Learning Outcome:

- Understand and apply stack data structure for efficient computation.
- Learn to use HashMap for quick lookups.
- Implement string operations and conversions in Java.
- Utilize sets and lists to process and match input words efficiently.

### 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

```
{ } O ( { } ) { } ( [ ]
```

#### Sample Output

```
true true false true
```

#### 2) Objective:

- Implement a stack-based approach to check for balanced parentheses.
- Read multiple lines of input until the end-of-file (EOF).
- Print 'true' if the string is balanced and 'false' otherwise.
- Handle different types of brackets: (), {}, and [].

### 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 (!isMatchingPair(open, ch)) return false;
            }
        }

        return stack.isEmpty();
    }

    private static boolean isMatchingPair(char open, char close) {
        return (open == '(' && close == ')') ||
            (open == '{' && close == '}') ||
            (open == '[' && close == ']');
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<String> results = new ArrayList<>();

        while (scanner.hasNext()) {
            String input = scanner.next();
            results.add(String.valueOf(isBalanced(input)));
        }

        scanner.close();
        System.out.println(String.join(" ", results));
    }
}
```



4) Output:

```
true true false true
```

5) Learning Outcome:

- Learned how to use a Stack to solve balanced parentheses problems.
- Gained experience in reading input till EOF using Scanner.
- Understood the importance of edge case handling in bracket matching.
- Practiced working with Java's Stack and List data structures.

**PROBLEM 5**

1) **Aim:.** Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '\*' where:

'?' Matches any single character.

'\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

**Example 1:**

**Input:** s = "aa", p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Constraints:**

0 <= s.length, p.length <= 2000

s contains only lowercase English letters.

p contains only lowercase English letters, '?' or '\*'.

2) **Objective:**

- Understand and implement dynamic programming for pattern matching.
- Learn how to handle wildcard characters in a pattern.
- Efficiently match strings using memoization.

3) **Implementation/Code:**

```
class WildcardMatching {  
    public boolean isMatch(String s, String p) {  
        int m = s.length();
```

```
int n = p.length();
boolean[][] dp = new boolean[m + 1][n + 1];
dp[0][0] = true;

for (int j = 1; j <= n; j++) {
    if (p.charAt(j - 1) == '*') {
        dp[0][j] = dp[0][j - 1];
    }
}

for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        if (p.charAt(j - 1) == '?' || p.charAt(j - 1) == s.charAt(i - 1)) {
            dp[i][j] = dp[i - 1][j - 1];
        } else if (p.charAt(j - 1) == '*') {
            dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
        }
    }
}
return dp[m][n];
}

public static void main(String[] args) {
    WildcardMatching wm = new WildcardMatching();
    String s = "aa";
    String p = "a";
    System.out.println("Input: s = \"\" + s + "\", p = \"\" + p + "\"");
    System.out.println("Output: " + wm.isMatch(s, p)); // Expected: false
}
}
```

#### 4) Output:

```
Input: s = "aa", p = "a"
Output: false
```

#### 5) Learning Outcome:

- Learned how to use dynamic programming for pattern matching problems.
- Understood handling of '?' and '\*' in wildcard matching.



**DEPARTMENT OF**

**COMPUTER SCIENCE & ENGINEERING**

Research Learning Empower