

Day 11:

Task 1: String Operations

Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.

Answer:

```
package com.wipro.graphalgo;

public class ss {

    public static String middleSubstring(String str1, String str2,
int length) {

        if (str1 == null || str1.isEmpty() || str2 == null ||
str2.isEmpty()) {
            return "Input strings cannot be null or empty";
        }

        String concatenated = str1.concat(str2);
        String reversed = new
StringBuilder(concatenated).reverse().toString();

        if (length > reversed.length()) {
            return "Substring length cannot be larger than the
concatenated string";
        }

        int startIndex = (reversed.length() - length) / 2;

        String middleSubstring = reversed.substring(startIndex,
startIndex + length);

        return middleSubstring;
    }

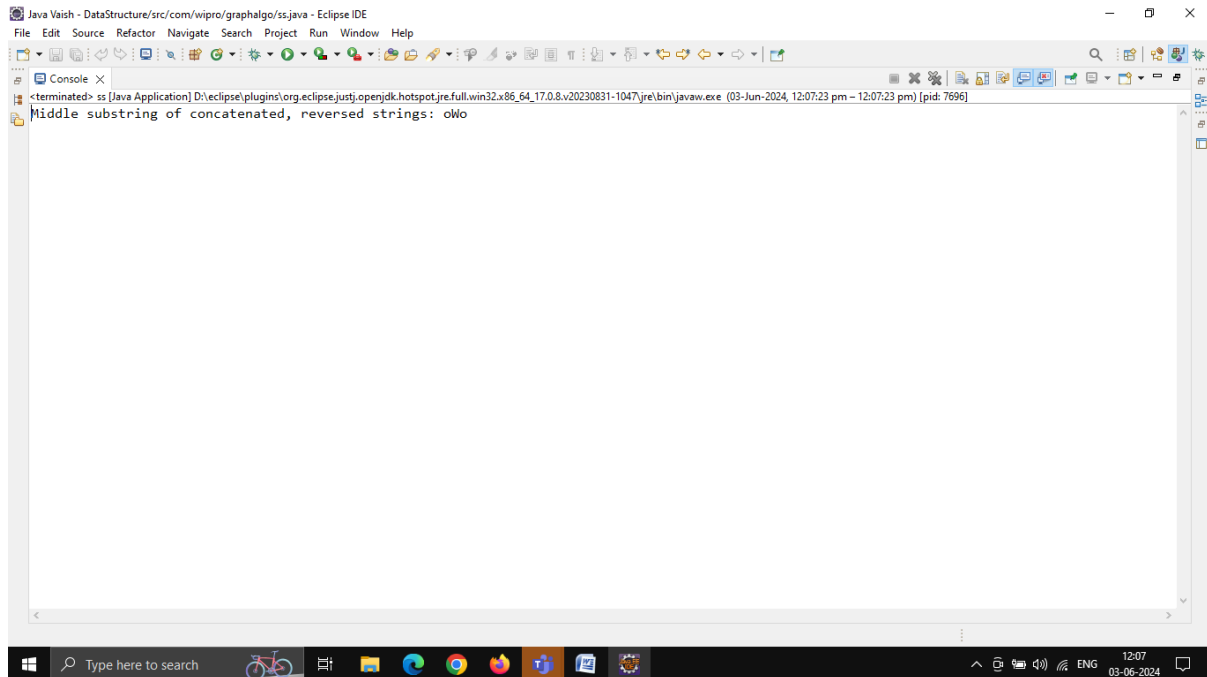
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";
```

```

        int length = 3;

        System.out.println("Middle substring of concatenated,
reversed strings: " +
                           middleSubstring(str1, str2, length));
    }
}

```



Task 2: Naive Pattern Search

Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.

Answer:

```

package com.wipro.graphalgo;
public class patterns {
    public static void searchPattern(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();
        int comparisons = 0;

        for (int i = 0; i <= textLength - patternLength; i++) {

```

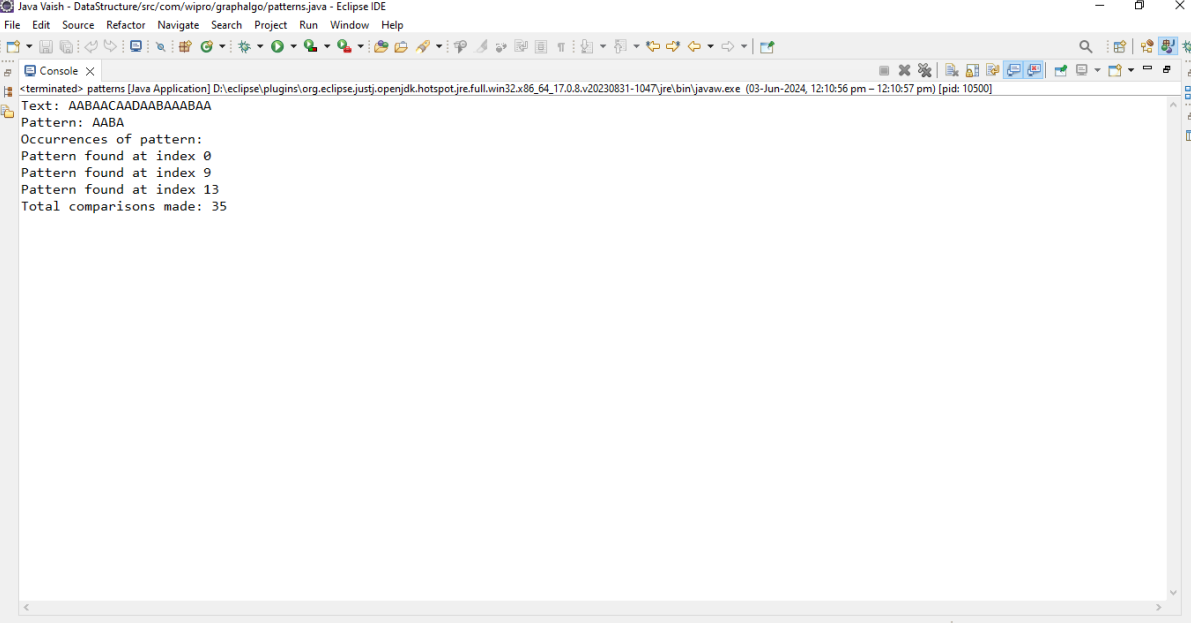
```

        int j;
        for (j = 0; j < patternLength; j++) {
            comparisons++;
            if (text.charAt(i + j) != pattern.charAt(j))
                break;
        }
        if (j == patternLength)
            System.out.println("Pattern found at index " + i);
    }

    System.out.println("Total comparisons made: " +
        comparisons);
}

public static void main(String[] args) {
    String text = "AABAACAADAABAAABAA";
    String pattern = "AABA";
    System.out.println("Text: " + text);
    System.out.println("Pattern: " + pattern);
    System.out.println("Occurrences of pattern:");
    searchPattern(text, pattern);
}
}

```



The screenshot shows the Eclipse IDE interface. The top bar indicates the file path: Java Vaish - DataStructure/src/com/wipro/graphalgo/patterns.java - Eclipse IDE. The console window at the bottom displays the following output:

```

Text: AABAACAADAABAAABAA
Pattern: AABA
Occurrences of pattern:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
Total comparisons made: 35

```

The taskbar at the bottom shows the Windows operating system with various application icons and the system clock displaying 12:11 on 03-06-2024.

Task 3: Implementing the KMP Algorithm

Code the Knuth-Morris-Pratt (KMP) algorithm in C# for pattern searching which pre-processes the pattern to reduce the number of

comparisons. Explain how this pre-processing improves the search time compared to the naive approach.

Answer:

```
package com.wipro.graphalgo;
public class Kmpal{
    private static int[] computeLPSArray(String pattern) {
        int patternLength = pattern.length();
        int[] lps = new int[patternLength];
        int len = 0;
        int i = 1;

        lps[0] = 0;

        while (i < patternLength) {
            if (pattern.charAt(i) == pattern.charAt(len)) {
                len++;
                lps[i] = len;
                i++;
            } else {
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }

        return lps;
    }

    public static void KMPsearch(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();
        int[] lps = computeLPSArray(pattern);
        int i = 0;
        int j = 0;

        while (i < textLength) {
            if (pattern.charAt(j) == text.charAt(i)) {
                j++;
                i++;
            }

            if (j == patternLength) {
                System.out.println("Pattern found at index " + (i - j));
            }
        }
    }
}
```

Java Vaish - DataStructure/src/com/vipro/graphalgo/Kmpal.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Console X

terminated> Kmpal [Java Application] D:\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.8.v20230831-1047\jre\bin\javaw.exe (03-Jun-2024, 12:25:09 pm - 12:25:10 pm) [pid: 6976]

Text: ABABDABACDABABCABAB

Pattern: ABABCABAB

Occurrences of pattern:

Pattern found at index 10

Type here to search

12:25 03-06-2024

Implement the Rabin-Karp algorithm for substring search using a rolling hash. Discuss the impact of hash collisions on the algorithm's performance and how to handle them.

Answer:

```

package com.wipro.graphalgo;

import java.util.ArrayList;
import java.util.List;

public class Rabin{

    public static List<Integer> search(String text, String pattern)
    {
        List<Integer> indices = new ArrayList<>();
        int n = text.length();
        int m = pattern.length();
        int prime = 101;
        int pHash = 0;
        int tHash = 0;
        int h = 1;

        for (int i = 0; i < m - 1; i++)
            h = (h * prime) % prime;

        for (int i = 0; i < m; i++) {
            pHash = (prime * pHash + pattern.charAt(i)) % prime;
            tHash = (prime * tHash + text.charAt(i)) % prime;
        }

        for (int i = 0; i <= n - m; i++) {
            if (pHash == tHash) {

                int j;
                for (j = 0; j < m; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j))
                        break;
                }
                if (j == m)
                    indices.add(i);
            }

            if (i < n - m) {
                tHash = (prime * (tHash - text.charAt(i) * h) + text.charAt(i
+ m)) % prime;

                if (tHash < 0)
                    tHash += prime;
            }
        }
        return indices;
    }

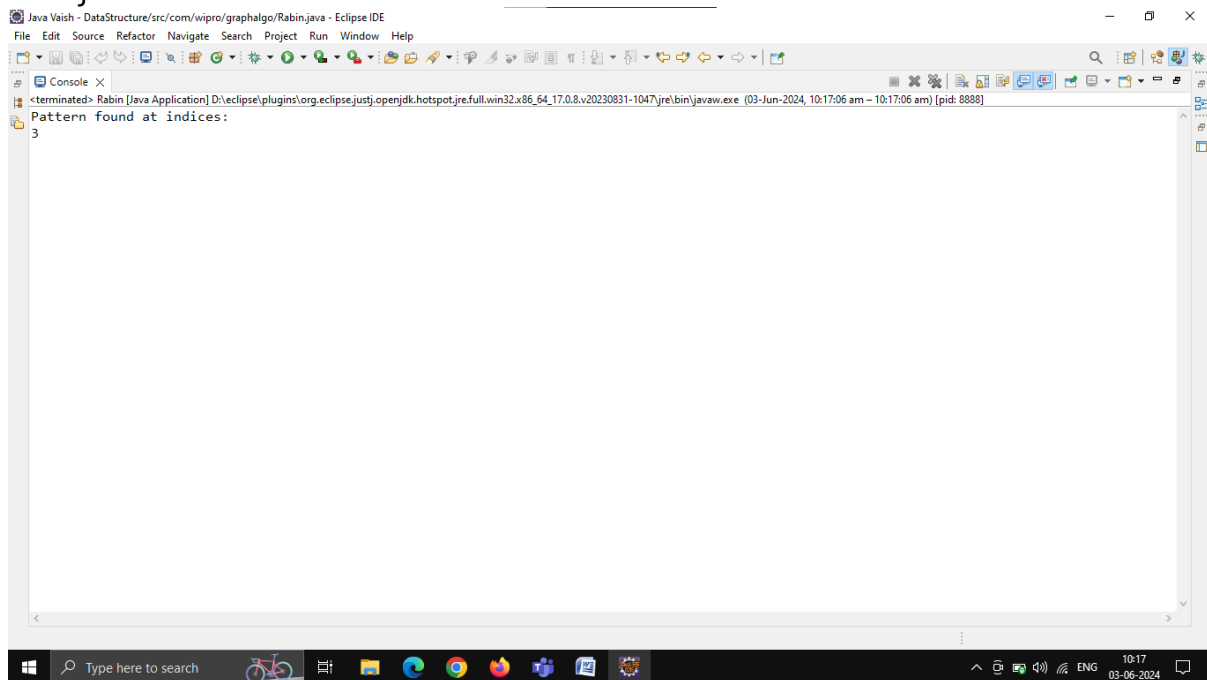
    public static void main(String[] args) {

```

```

String text = "ASDASFSDDGDFHWRFFV";
String pattern = "ASFS";
List<Integer> indices = search(text, pattern);
if (indices.isEmpty())
    System.out.println("Pattern not found");
else {
    System.out.println("Pattern found at indices:");
    for (int index : indices)
        System.out.println(index);
    }
}

```



Task 5: Boyer-Moore Algorithm Application

Use the Boyer-Moore algorithm to write a function that finds the last occurrence of a substring in a given string and returns its index
Explain why this algorithm can outperform others in certain scenarios

Answer:

```

package com.wipro.graphalgo;

public class Boyer {

    private static final int ASCII_SIZE = 256;

    public static int lastOccurrence(String text, String pattern) {

```

```

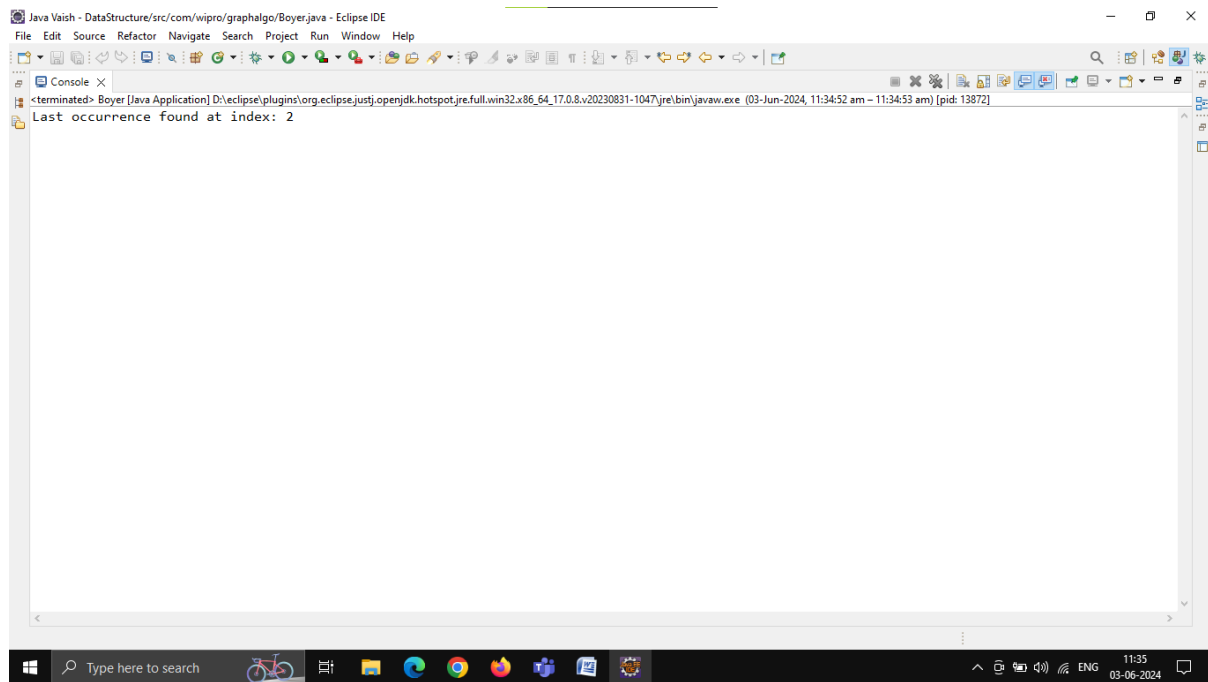
    int[] last = buildLastTable(pattern);
    int n = text.length();
    int m = pattern.length();
    int i = m - 1;
    int j = m - 1;

    while (i < n) {
        if (text.charAt(i) == pattern.charAt(j)) {
            if (j == 0) {
                return i;
            }
            i--;
            j--;
        } else {
            i += m - Math.min(j, 1 + last[text.charAt(i)]);
            j = m - 1;
        }
    }
    return -1;
}

private static int[] buildLastTable(String pattern) {
    int[] last = new int[ASCII_SIZE];
    for (int i = 0; i < ASCII_SIZE; i++) {
        last[i] = -1;
    }
    for (int i = 0; i < pattern.length(); i++) {
        last[pattern.charAt(i)] = i;
    }
    return last;
}

public static void main(String[] args) {
    String text = "ababcbabcbabcabc";
    String pattern = "abc";
    int lastIndex = lastOccurrence(text, pattern);
    if (lastIndex != -1) {
        System.out.println("Last occurrence found at index: " +
lastIndex);
    } else {
        System.out.println("Pattern not found in the text.");
    }
}
}

```

Day 12:

Task 1: Bit Manipulation Basics

Create a function that counts the number of set bits (1s) in the binary representation of an integer. Extend this to count the total number of set bits in all integers from 1 to n.

Answer:

```
package com.wipro.ds;
public class Bit {
    public static int countSetBits(int num) {
        int count = 0;
        while (num > 0) {
            count += num & 1;
            num >>= 1;
        }
        return count;
    }

    public static int countTotalSetBits(int n) {
        int totalCount = 0;
        for (int i = 1; i <= n; i++) {
            totalCount += countSetBits(i);
        }
    }
}
```

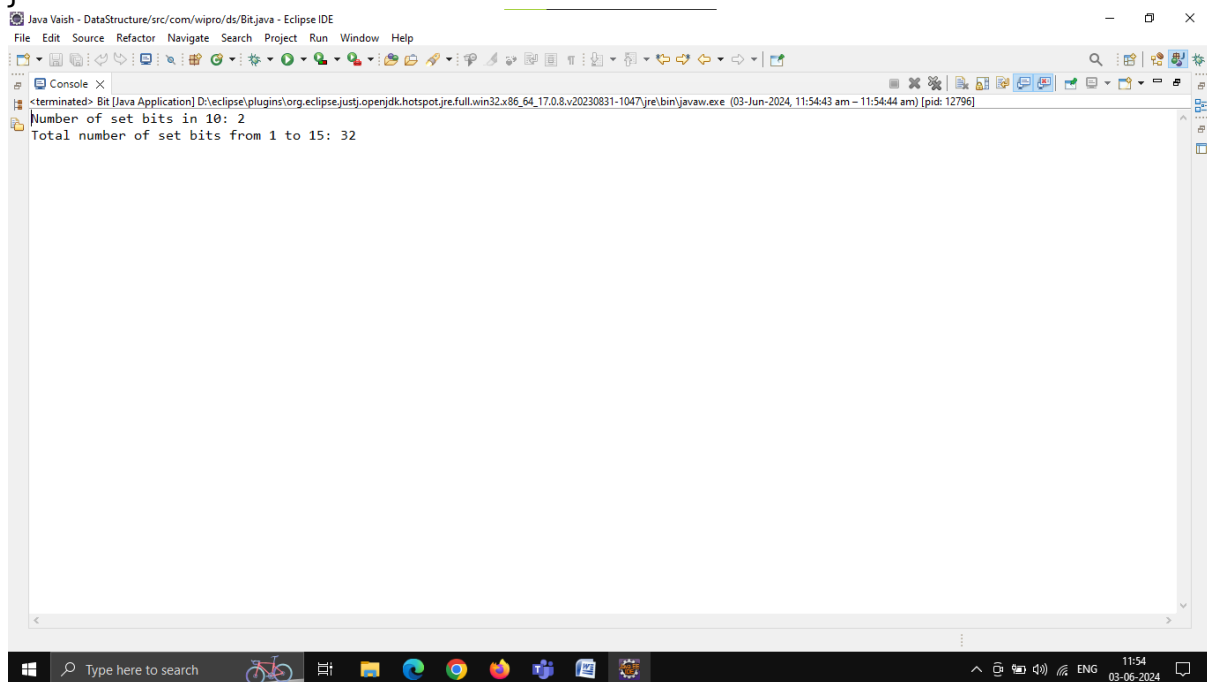
```

        return totalCount;
    }

    public static void main(String[] args) {
        int num = 10;
        System.out.println("Number of set bits in " + num + ": " +
countSetBits(num));

        int n = 15;
        System.out.println("Total number of set bits from 1 to " + n
+ ": " + countTotalSetBits(n));
    }
}

```



Task 2: Unique Elements Identification

Given an array of integers where every element appears twice except for two, write a function that efficiently finds these two non-repeating elements using bitwise XOR operations.

Answer:

```

package com.wipro.ds;
public class Unique{
    public static int[] findNonRepeatingElements(int[] nums) {
        int xor = 0;
        for (int num : nums) {
            xor ^= num;
        }
    }
}

```

```

    }
    int rightmostSetBit = xor & -xor;

    int[] result = new int[2];
    for (int num : nums) {
        if ((num & rightmostSetBit) == 0) {
            result[0] ^= num;
        } else {
            result[1] ^= num;
        }
    }
    return result;
}

public static void main(String[] args) {
    int[] nums = {2, 4, 7, 9, 2, 4, 5, 9};
    int[] result = findNonRepeatingElements(nums);
    System.out.println("Non-repeating elements are: " +
result[0] + " and " + result[1]);
}
}

```

