Day 17:

Assignment 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.

A)

The Boyer-Moore algorithm is efficient due to two key heuristics:

1. Bad Character Heuristic: *When a mismatch occurs, the algorithm uses the position of the mismatched character in the pattern to skip sections of the text.*

2. Good Suffix Heuristic: *If a substring of the pattern matches but then a mismatch occurs, the algorithm uses the longest suffix of the matched part to shift the pattern.*

These heuristics allow Boyer-Moore to skip large portions of the text, which can lead to much faster search times compared to algorithms like the naive approach or even KMP in many practical cases.

Java code:

```
package Day17;
import java.util.Arrays;
public class BoyerMoore {
        private static final int ALPHABET_SIZE = 256;

  private static void preprocessBadCharacterHeuristic(char[] pattern, int m, int[] badChar) {
    Arrays.fill(badChar, -1);
    for (int i = 0; i < m; i++) {
      badChar[pattern[i]] = i;
    }
  }


  private static void preprocessGoodSuffixHeuristic(char[] pattern, int m, int[] suffix, int[] goodSuffix) {
    Arrays.fill(suffix, -1);
    Arrays.fill(goodSuffix, m);

    int f = 0, g = m - 1;
    suffix[m - 1] = m;
    for (int i = m - 2; i >= 0; i--) {
      if (i > g && suffix[i + m - 1 - f] < i - g) {
        suffix[i] = suffix[i + m - 1 - f];
      } else {
        if (i < g) {
                g = i;
            }
            f = i;
            while (g >= 0 && pattern[g] == pattern[g + m - 1 - f]) {
                g--;
            }
            suffix[i] = f - g;
        }
    }

    for (int i = 0; i < m; i++) {
        goodSuffix[i] = m;
    }

    for (int i = 0, j = 0; j < m - 1; j++) {
        if (suffix[j] == j + 1) {
            while (i < m - 1 - j) {
                if (goodSuffix[i] == m) {
                    goodSuffix[i] = m - 1 - j;
                }
                i++;
            }
        }
    }

    for (int i = 0; i < m - 1; i++) {
        goodSuffix[m - 1 - suffix[i]] = m - 1 - i;
    }
}


public static int boyerMooreSearch(String text, String pattern) {
    char[] txt = text.toCharArray();
    char[] pat = pattern.toCharArray();
    int n = txt.length;
    int m = pat.length;

    int[] badChar = new int[ALPHABET_SIZE];
    int[] suffix = new int[m];
    int[] goodSuffix = new int[m];

    preprocessBadCharacterHeuristic(pat, m, badChar);
    preprocessGoodSuffixHeuristic(pat, m, suffix, goodSuffix);

    int s = 0;
    int lastOccurrence = -1;
    while (s <= (n - m)) {
        int j = m - 1;
```

```
            while (j >= 0 && pat[j] == txt[s + j]) {
                j--;
            }


            if (j < 0) {
                lastOccurrence = s;
                s += goodSuffix[0];
            } else {
                s += Math.max(goodSuffix[j], j - badChar[txt[s + j]]);
            }
        }

        return lastOccurrence;
    }
    public static void main(String[] args) {
        String text = "ABAAABCDABC";
        String pattern = "ABC";
        int lastIndex = boyerMooreSearch(text, pattern);
        if (lastIndex != -1) {
            System.out.println("The last occurrence of pattern is at index: " + lastIndex);
        } else {
            System.out.println("Pattern not found");
        }
    }
}
```
Explanation of the Code

1. Bad Character Heuristic:
- The preprocessBadCharacterHeuristic function fills the bad character array. This array is used to determine how far to shift the pattern when a mismatch occurs.

2. Good Suffix Heuristic:
- The preprocessGoodSuffixHeuristic function builds the suffix and good suffix arrays. These arrays help determine the shift when a mismatch occurs at a suffix of the pattern.

3. Boyer-Moore Search:
- The boyerMooreSearch function searches for the pattern in the text using both heuristics. It updates the shift based on the bad character and good suffix values and tracks the last occurrence of the pattern.

4. Main Function:
- Tests the boyerMooreSearch function with a sample text and pattern, printing the index of the last occurrence of the pattern in the text.

Efficiency of Boyer-Moore Algorithm

The Boyer-Moore algorithm can skip large sections of the text, especially when the pattern and text have few common characters. This makes it very efficient in many practical scenarios, often outperforming other algorithms like the naive approach or KMP, particularly when searching for patterns in large texts.