Report

Implementation of the Knuth–Morris–Pratt (KMP) String Matching Algorithm

Author: Nygman Dias, SE-2435

1. Introduction

This report describes the implementation of the Knuth–Morris–Pratt (KMP) string-matching algorithm in Java as part of the Bonus Task.
The goal of KMP is to efficiently find all occurrences of a pattern within a text by avoiding unnecessary character comparisons using a prefix-function (LPS array).

The project includes:

- A full Java implementation of KMP
- Three test cases (short, medium, long strings)
- Sample outputs recorded in a separate text file
- Complexity analysis
- Commented source code

2. Algorithm Overview

KMP improves upon the naive string-matching algorithm by precomputing an array called **LPS (Longest Prefix which is also a Suffix)**.
The LPS array tells the algorithm how much to shift the pattern without rechecking characters that are already known to match.

How it works:

1. Precompute LPS array for the pattern
2. Scan the text with two pointers
3. When mismatch occurs, instead of moving back, use LPS to skip comparisons

4. When characters match, advance both pointers
5. If the pattern is found, record the starting index

This ensures linear-time pattern searching.

3. Implementation Summary

The Java program contains:

- computeLPS() — creates the prefix function
- KMPSearch() — searches for the pattern
- Main.java — runs three test cases with different string lengths

The code is fully commented and easy to follow.

4. Test Cases and Results

Test Case 1 — Short String

Text: "ababcabcab"
Pattern: "abc"
Result: Found at index: 2, 5

Test Case 2 — Medium String

Text: "thequickbrownfoxjumpsoverthelazydogquickbrown"
Pattern: "quick"
Result: Found at index: 3, 36

Test Case 3 — Long String

Contains ~200 characters.
Pattern: "algorithm"
Result: Found only once (index depends on exact input).

All results are included inside *sample_outputs.txt*.

5. Time and Space Complexity Analysis

Time Complexity

Step	**Complexity**
Building LPS array	**O(m)**
Searching in text	**O(n)**
Total	**O(n + m)**

Where:

- n = text length
- m = pattern length

KMP is asymptotically optimal for single-pattern matching.

Space Complexity

- LPS array requires **O(m)** extra memory
- Other variables use constant space
- **Total Space: O(m)**

Thus, KMP is both **time-efficient** and **space-efficient** for pattern matching.

6. Conclusion

The KMP algorithm was successfully implemented and tested using short, medium, and long input strings.
The results demonstrate its linear running time and efficiency compared to naive approaches.
A GitHub repository would typically include:

- Source code
- Sample input and output
- This report

KMP remains one of the most practical algorithms for efficient substring search.