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

Author: *Madyshev Dias*
Project: Bonus Task – String Algorithms
Algorithm Implemented: Knuth–Morris–Pratt (KMP)

1. Introduction

This report presents the implementation, testing, and analysis of the **Knuth–Morris–Pratt (KMP)** string searching algorithm in Java.
The KMP algorithm is an efficient linear-time method for finding occurrences of a pattern within a given text. Unlike the naïve algorithm, which may re-check characters multiple times, KMP uses a preprocessed prefix table called the LPS (Longest Prefix-Suffix) array to reduce redundant comparisons.

The goal of this project is to implement KMP **from scratch**, test it on strings of different lengths, and analyze its performance characteristics.

2. Algorithm Overview

The KMP algorithm consists of two main parts:

2.1 LPS (Longest Prefix-Suffix) Array Construction

The LPS array indicates, for each position in the pattern, the length of the longest prefix that is also a suffix.
It allows the search algorithm to skip characters that are known not to match, instead of rechecking them.

For example, for the pattern `"ababaca"`, the LPS array would be:

```
[0, 0, 1, 2, 3, 0, 1]
```

Building the LPS array requires only one pass through the pattern, making this phase **O(m)**.

2.2 Search Phase

The main search uses both the text and the pattern while referring to the LPS array when mismatches occur.
Instead of returning to the beginning of the pattern (as the naïve algorithm does), KMP repositions the pattern efficiently using information stored in LPS.

This ensures that:

- each character in the text is examined at most once,
- redundant comparisons never happen.

The search phase runs in **O(n)**.

3. Implementation Summary

The implementation was written entirely in Java and includes:

- `KMP.java` — full KMP implementation
- `Main.java` — three test cases of different lengths
- clear inline comments
- console-based demonstration

The code was developed and executed using **IntelliJ IDEA**, and version control was managed via **GitHub**.

4. Testing and Results

Three test cases were created to evaluate algorithm behavior under different input sizes.

4.1 Short Test

- **Text length:** ~12 characters
- **Pattern:** `"abc"`
- **Result:** The algorithm correctly identified all matches.

4.2 Medium Test

- **Text length:** ~40 characters
- **Pattern:** `"abcabc"`
- **Result:** Multiple matches found, demonstrating correct handling of overlapping patterns.

4.3 Long Test

- **Text length:** ~600,000 characters (generated programmatically)
- **Pattern:** `"abcab"`
- **Result:** Execution completed extremely fast due to linear complexity.
  The test confirms the scalability and efficiency of KMP for large datasets.

5. Complexity Analysis

Time Complexity

Phase	Complexity	Explanation
LPS construction	**O(m)**	Scans the pattern once
Search phase	**O(n)**	Scans the full text once
Total	**O(n + m)**	Linear time

This is a major improvement over the naïve algorithm's worst-case **O(n × m)** complexity.

Space Complexity

The only extra memory used is for the LPS array:

- **LPS array size:** `m`

- **Total extra space: O(m)**

No additional data structures are required.

6. Conclusion

The Knuth–Morris–Pratt (KMP) algorithm was successfully implemented from scratch in Java.
Testing with three different text lengths confirms that the algorithm performs accurately and efficiently across all scenarios. Its linear time performance becomes especially noticeable on large input strings.

KMP remains one of the most elegant and efficient algorithms for substring searching, and this implementation demonstrates its practical benefits compared to simpler methods.

7. Repository Link

All project materials—including source code, input/output samples, and this report—are available at:

👉 https://github.com/redd1eg/bonusttask