String Matching Algorithms – Brief Report

1 Boyer–Moore Implementation Approach

For this assignment, I implemented the Boyer–Moore string matching algorithm using the Bad Character heuristic.

The algorithm preprocesses the pattern and then scans the text by comparing characters from right to left. When a mismatch occurs, the bad character table is used to determine how far the pattern can be shifted. This allows the algorithm to skip unnecessary comparisons and improves performance in many practical cases.

In my implementation, I store the last occurrence of each character in the pattern using a hash-based structure. A minimum shift of one is enforced to avoid infinite loops. Special cases, such as empty patterns or patterns longer than the text, are handled explicitly. With these choices, the implementation behaves correctly and passes all provided test cases.

—

2 GoCrazy Algorithm Design and Rationale

In addition to the classical algorithms, I designed a custom algorithm named **GoCrazy**.

GoCrazy is inspired by the naive string matching approach but introduces a different comparison strategy. Instead of checking the pattern strictly from left to right, it compares characters from both ends of the pattern moving inward. This often reveals mismatches earlier, especially when the pattern edges contain distinctive characters.

For the special case where the pattern length is one, GoCrazy simply scans the text for that character. This avoids unnecessary overhead and performs very efficiently in practice.

During testing, GoCrazy performed competitively and was the fastest algorithm in several cases, particularly for symmetric or palindrome-like patterns.

—

3 Pre-Analysis Strategy and Motivation

To avoid always running the same algorithm, I implemented a pre-analysis phase that selects an appropriate string matching algorithm based on simple input properties.

The following features are extracted:

- Pattern length (m)
- Text length (n)
- Number of distinct characters in the pattern
- Repetition ratio ($alphabetSize/m$)
- Symmetry score based on matching characters at both ends of the pattern

Using these features, I defined a set of heuristic rules:

1. If $m = 0$ or $m > n$, I select the Naive algorithm.
2. If $m \leq 2$, I again select Naive because preprocessing overhead is not justified.
3. If the repetition ratio is low and m is reasonably large, I select KMP, which handles repetitive patterns efficiently.
4. If the text is long and the pattern is short with many distinct characters, I select Rabin–Karp.
5. If both the text and the pattern are long and the alphabet size is large, I select Boyer–Moore.
6. If the pattern shows strong symmetry, I select GoCrazy.
7. If no rule clearly applies, I fall back to Naive.

The goal of these rules is not to be perfect, but to make reasonable choices without expensive preprocessing.

—

4 Analysis of Results

On the provided benchmark test cases, the pre-analysis strategy selected the fastest algorithm in a majority of cases. While the improvement is not dramatic for very small inputs, it becomes more noticeable as the text length increases.

One interesting observation is that the Naive algorithm is selected quite often. This is a deliberate choice: for small or ambiguous cases, the simplicity of Naive avoids unnecessary preprocessing overhead, which can outweigh theoretical advantages of more complex algorithms.

KMP was mostly selected for repetitive patterns, Boyer–Moore for larger alphabets and longer patterns, Rabin–Karp for long texts with short diverse patterns, and GoCrazy for symmetric patterns. Overall, the results align well with the known strengths of these algorithms.

—

5 My Journey

At the beginning of this assignment, I expected that more advanced algorithms such as Boyer–Moore would always outperform simpler ones.

After implementing and testing the algorithms, I realized that this assumption was not always correct. In several cases, especially with small inputs or highly repetitive patterns, simpler approaches performed just as well or even better.

Designing the GoCrazy algorithm allowed me to experiment with alternative comparison strategies, and the pre-analysis phase helped me think more carefully about when an algorithm is actually worth using. This assignment helped me better understand practical algorithm design beyond theoretical complexity analysis.

—

6 Use of ChatGPT

During this assignment, I used ChatGPT as a supportive tool to clarify certain theoretical concepts and to reflect on algorithm design ideas.

ChatGPT was mainly helpful for understanding algorithm intuitions and for organizing my thoughts while writing the report. All algorithm implementations, experiments, and final decisions were made by me, and I ensured that I fully understood the logic behind each part of the solution.

I treated ChatGPT as a learning aid rather than a source of final answers, and I verified all ideas through my own implementation and testing.

Name Surname: Berkay AKSOY
Student Number: 22050111081