Galip Yılmaz 22050111013

Abdullah Tülek 22050111079

Project Report: String Matching Algorithms

1- Boyer-Moore Implementation Approach

We implemented the standard Boyer-Moore algorithm using the Bad Character Heuristic. To ensure robustness against special characters, we replaced the traditional fixed-size array with a HashMap (or a large array where applicable) to support the full Unicode character set without ArrayIndexOutOfBounds errors.

2- GoCrazy Algorithm Design and Rationale

For the custom algorithm, we implemented an optimized Boyer-Moore-Horspool variant with a Hybrid Table Strategy.

- **Rationale:** Standard Boyer-Moore has high preprocessing overhead due to the "Good Suffix" rule. Horspool simplifies this by only using the "Bad Character" rule based on the last character of the window, making it faster for average cases.
- **Creative Optimization:** We implemented a hybrid lookup system. If the pattern is purely ASCII, it uses a lightweight int[] array for maximum speed. If it contains Unicode, it switches to a HashMap. This minimizes initialization overhead while maintaining safety.

3- Pre-Analysis Strategy

Our strategy evolved significantly after analyzing the initial test runs. We focused heavily on "Initialization Overhead vs. Search Speed."

- **The "Zero-Overhead" Rule:** We observed that for texts shorter than 300 characters, the overhead of creating HashMaps (for BM/Horspool) or arrays (for KMP) often takes longer than the actual search. Since most test cases fell into this category, we configured our pre-analysis to strictly default to Naive for all inputs under 300 characters.
- **Heavy Situations:** We reserved GoCrazy (Horspool) only for texts longer than 300 characters, where its character-skipping logic provides a tangible benefit over Naive's O(N·M).

4- Analysis of Results

The test results confirmed our hypothesis about overhead. The Naive algorithm outperformed complex algorithms on short texts (0-300 chars) due to Java's loop optimizations. However, on the "Very Long Text" test case, GoCrazy was significantly the fastest along other algorithms. Our pre-analysis logic successfully selected the fastest algorithm in almost every scenario by correctly identifying the threshold where initialization cost becomes negligible compared to search efficiency.

5- Our Research & Transparency

- **Documentation of Resources & Strategy:** To develop the "GoCrazy" algorithm and the Pre-Analysis strategy, we utilized a combination of academic resources and AI assistance.
- **Algorithm Research:** We researched the Boyer-Moore-Horspool algorithm on GeeksForGeeks and Wikipedia to understand how it simplifies the standard Boyer-Moore algorithm by removing the "Good Suffix" rule.
- **AI Assistance:** We used Google Gemini as a pair programmer and debugger throughout the process.
- **Optimization:** When our initial GoCrazy implementation crashed on Unicode characters, we consulted the Gemini to understand why int[256] was insufficient. It suggested using a HashMap for safety or checking for ASCII compatibility to use arrays for speed. This led to our Hybrid Table Strategy.
- **Data Analysis:** We fed the execution time tables into Gemini to identify why the "Naive" algorithm was winning so frequently. The AI pointed out the "Initialization Overhead" factor, which we hadn't considered. This insight was crucial for tuning the StudentPreAnalysis logic (setting the 300-character threshold).
- **Transparency:** We didn’t simply copy paste the generated code. We actively iterated on the logic based on the test results. For instance, our initial Pre-Analysis strategy was too complex and failed to select the fastest algorithm. After discussing the results with the AI, we realized that a simpler strategy prioritizing Naive for small inputs was more effective. The final code reflects a refined understanding of how Java's JVM optimizes simple loops versus the cost of object instantiation.

Our Journey (Reflection)

This assignment was an eye-opener regarding "Theoretical Complexity vs. Practical Overhead."

- **What We Learned:** We initially expected complex algorithms like KMP or Boyer-Moore to dominate. However, we learned that object creation and preprocessing have a cost. For the specific dataset provided (mostly small strings), the brute-force of Naive algorithm approach was practically faster than smarter and more complex algorithms.
- **Challenges:** Optimizing the GoCrazy algorithm to beat Naive was difficult because Naive sets a very high bar on small inputs. We also dealt with Unicode issues, which taught us about Java's character handling.
- **Feedback:** It was interesting to see that O(N·M) complexity is not always bad in practice when N and M are small.