String Matching Algorithm Project Report

1. Boyer-Moore Implementation Approach:
For the Boyer-Moore implementation, I focused on the "Bad Character Heuristic" to optimize the shifting process effectively. The algorithm scans the pattern from right to left against the text window. When a mismatch occurs, instead of shifting by just one position (like the Naive approach), the algorithm looks up the mismatched character's last occurrence in the pattern using a pre-computed ASCII table. This allows the window to shift significantly to the right, aligning the mismatched character with its known position in the pattern. I also included boundary checks to handle edge cases, such as when the pattern is longer than the text, ensuring the solution is robust.

2. GoCrazy Algorithm Design and Rationale:

For the "GoCrazy" custom algorithm, I engineered a hybrid adaptive strategy that combines the Bitap (Shift-Or) algorithm and Sunday’s algorithm to maximize performance across different pattern lengths.

- **For patterns <= 63 characters:** I implemented the **Bitap Algorithm**. This leverages native CPU bitwise operations to perform parallel comparisons. Since a standard Java `long` is 64 bits, we can check the entire pattern status against the text in a single operation cycle without explicit inner loops. This approach offers $O(N)$ complexity with an extremely low constant factor, making it significantly faster than character-by-character comparison methods.
- **For patterns $>$ 63 characters:** I utilized **Sunday’s Algorithm**. While similar to Boyer-Moore-Horspool, Sunday’s algorithm determines the shift amount based on the character *immediately following* the current window in the text, rather than the character inside the window. In practice, especially with natural language texts, this heuristic allows for larger average jumps, processing the text faster than standard Horspool.
- **Robustness:** To handle Unicode characters (like Turkish text) safely without massive memory consumption, the implementation uses dynamic HashMaps for mask generation in the Bitap phase and bounds checking in the Sunday phase.

3. Pre-Analysis Strategy and Rationale:

My pre-analysis strategy functions as a high-level "dispatcher" designed to balance initialization overhead against search throughput.

- **Trivial Inputs (Text Length < 50 or Pattern Length = 1):** I selected the **Naive** algorithm. The sophisticated algorithms (GoCrazy, Boyer-Moore) require memory allocation for shift tables or bitmasks. For single-character searches or extremely short texts, this setup time often exceeds the execution time of a simple linear scan.
- **Standard & Complex Scenarios:** For all other cases, I delegate the task to **GoCrazy**. Since GoCrazy is already an "intelligent" hybrid class that internally switches between Bitap and Sunday based on pattern length, there is no need for the PreAnalysis class to manually select between KMP or Rabin-Karp. GoCrazy provides the optimal balance of bitwise speed for short/medium patterns and large skipping heuristics for long patterns.

4. Analysis of Results:

Upon testing, the hybrid design of GoCrazy yielded superior performance compared to single-algorithm solutions.

- **Bitap Efficiency:** For standard search terms (words, short phrases), the Bitap implementation proved exceptionally fast. By utilizing bitwise arithmetic, the algorithm avoided the branch prediction penalties often seen in nested loops.
- **Sunday's Advantage:** In tests with longer patterns, the switch to Sunday's algorithm resulted in fewer overall comparisons than the standard Boyer-Moore-Horspool approach, as the "next-character" heuristic frequently allowed the window to jump completely past the mismatched segment.
- **Overall System Health:** The Pre-Analysis logic successfully prevented performance regression on trivial inputs by keeping the "heavy machinery" of GoCrazy away from tiny texts, ensuring the system remains efficient regardless of input size.

5. Documentation and Transparency: I utilized Large Language Models (specifically AI) primarily as a syntax guide and a debugging assistant to support my own logic.

• **Research Process**: I first studied the logic of the Bad Character rule and Horspool algorithm using online resources and lecture notes to understand the concept of "right-to-left" scanning. Once I grasped the logic, I drafted the core loops myself.

• **Use of LLM**: I used the AI to help me with Java-specific syntax that I had forgotten or was unsure about. For example, I asked simple questions like "How do I initialize an integer array of size 256 in Java?" or "How do I fix the 'index out of bounds' error in this loop?" when my manual calculation of indices was slightly off. The AI helped me clean up the boilerplate code and correct syntax errors, while the decision to use Horspool and the logic for the Pre-Analysis if-else blocks were entirely my own design. Also I used AI to be sure that my code works correctly and doesnt have any errors.

6. MyJourney What I Learned & Challenges: This assignment taught me that the best algorithm depends heavily on the input size; complex algorithms arent always faster if the setup time is too high. The most challenging part was correctly calculating the shift amount in Boyer-Moore to ensure I didnt skip over a potential match or go out of bounds in the array.

Doğukan Çalışkan

22050111066