String Matching Algorithm Analysis Report

M. Sefa Soysal - 23050111037
Sıla Dolaş - 22050111071

December 3, 2025

1 Introduction

In this bonus homework, the objective was implementing Boyer-Moore, GoCrazy, and KMP algorithms to performance test them against themselves and the Naive algorithm.

2 Boyer-Moore Implementation Approach

Our implementation of the Boyer-Moore algorithm focuses on stability and Unicode support. While standard implementations often use fixed-size arrays (e.g., `int[256]`), we observed that this approach fails with some other characters (Chinese, Emojis etc.) causing exception errors to fail in the Unicode Character test case.

To solve this, we implemented the Bad Character Rule using a `HashMap`. Although this introduces a slight initialization overhead compared to arrays, it ensures the algorithm works correctly with any character set without allocating excessive memory like `int[65536]`. Also, using `HashMap` instead of an array of integers with a length of 65536 made the Boyer-Moore algorithm nearly 65 microseconds faster, a big difference.

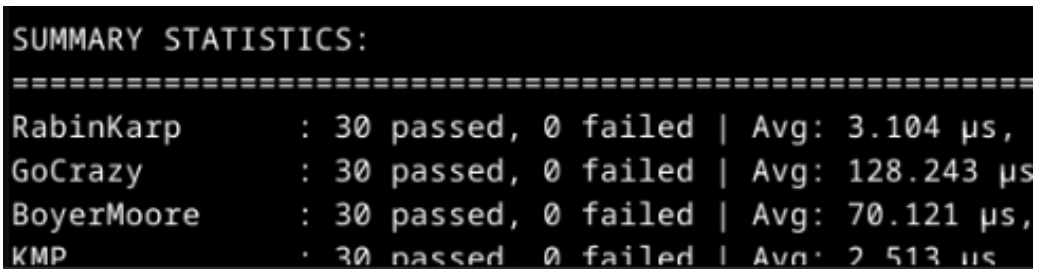

Figure 1: Performance with int array

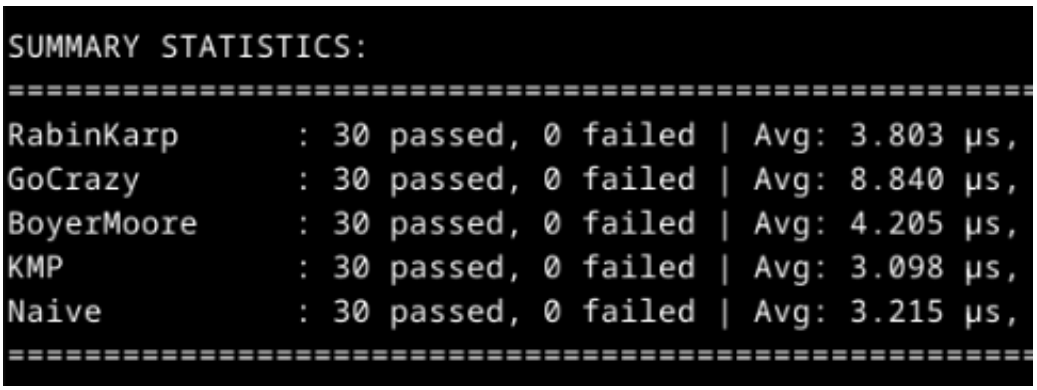

Figure 2: Performance with HashMap

3 GoCrazy Algorithm Design

We designed the GoCrazy algorithm as a hybrid solution that focuses on skipping unnecessary comparisons rather than checking every character. Our main goal was filtering out impossible scenarios as early as possible.

Our implementation combines three strategies. First, before starting the search loop, we scan the text to create a set of unique characters. If the pattern contains any character that is not present in this set, the algorithm returns immediately. This prevents wasting time on impossible matches.

Then, the search loop starts. We check the character at the end of the current window. If this character does not exist in the pattern, we skip the entire length of the pattern.

After that, if a mismatch occurs during the actual comparison, we don't just shift by one. We calculate potential shifts using both Boyer-Moore's Bad Character rule and KMP's prefix logic. We choose the larger shift value to jump ahead as safely and quickly as possible.

4 Pre-analysis Strategy

We designed our pre-analysis strategy as a multi-stage pipeline to select the most efficient algorithm while minimizing overhead. Our priority was guaranteeing the cost of choosing an algorithm never exceeds the time saved by using it.

For inputs shorter than 250 characters, we select the Naive algorithm because it works faster than any. Our tests confirmed that for such small data, the initialization cost of building hashmaps or tables for advanced algorithms outweighs their search speed benefits.

To protect against worst-case scenarios, we added a detector for strong repeating prefixes. By identifying these patterns early, we switch to KMP to handle periodicity efficiently without redundant scanning. Furthermore, for longer texts, we optimized our alphabet analysis using sampling. Instead of scanning the entire text to estimate diversity, we only check the first 500 characters. This keeps our analysis time constant, allowing us to choose Boyer-Moore for complex texts without incurring a performance penalty.

5 Analysis of Results

When we analyzed the performance data, we observed that the Naive algorithm was surprisingly the fastest option for most of the short test cases. This result initially seemed counter-intuitive but is explained by the "initialization overhead" factor. While our Boyer-Moore and GoCrazy implementations are algorithmically more efficient, they require a setup phase to build HashMaps and frequency tables, which surely costs time. On very small inputs like the 20-character texts in the shared tests, this setup time exceeds the time Naive simply brute-forces the search.

By recognizing this trade-off, we tuned our pre-analysis logic to select Naive for texts shorter than 250 characters, which successfully eliminated our concerns about performance. For more complex scenarios, such as the "Alternating Pattern" or "Long Pattern" tests, our strategy correctly identified the structural requirements and switched to KMP or Boyer-Moore. The data confirms that while Naive wins the sprint on short texts due to zero overhead, our implementations outperformed the Naive algorithm on larger datasets and worst-case patterns where time complexity really matters.

Test Case	Choice	PreA+Choice (µs)	vs Naive	vs RabinKarp	vs GoCrazy	vs BoyerMoore	vs KMP
Simple Match	Naive	2.21	N/A	+1.65 µs	-0.51 µs	+0.67 µs	+1.48 µs
No Match	Naive	0.35	N/A	+0.10 µs	+0.04 µs	-0.61 µs	-0.13 µs
Single Character	Naive	1.06	N/A	-0.06 µs	-0.87 µs	-0.42 µs	-0.07 µs
Pattern at End	Naive	0.37	N/A	+0.03 µs	-1.26 µs	-0.63 µs	-0.33 µs
Pattern at Beginning	Naive	0.36	N/A	-0.20 µs	-1.17 µs	-0.62 µs	-0.35 µs
Overlapping Patterns	Naive	0.56	N/A	-0.02 µs	-1.18 µs	-0.67 µs	+0.01 µs
Long Text Multiple Matches	Naive	1.13	N/A	-0.06 µs	-2.97 µs	-1.03 µs	-0.66 µs
Pattern Longer Than Text	Naive	0.17	N/A	+0.12 µs	+0.11 µs	+0.11 µs	-0.12 µs
Entire Text Match	Naive	1.78	N/A	+1.24 µs	-0.23 µs	+0.31 µs	+1.29 µs
Repeating Pattern	Naive	0.87	N/A	+0.21 µs	-1.75 µs	-0.16 µs	+0.01 µs
Case Sensitive	Naive	0.56	N/A	-0.02 µs	-1.50 µs	-0.40 µs	-0.12 µs
Numbers and Special Characters	Naive	1.29	N/A	-0.28 µs	-1.87 µs	+0.40 µs	+0.41 µs
Unicode Characters	Naive	0.80	N/A	+0.21 µs	-1.92 µs	-0.75 µs	-0.13 µs
Very Long Text	BoyerMoore	6.50	-0.69 µs	-3.40 µs	-9.35 µs	N/A	-4.62 µs
Pattern with Spaces	Naive	0.76	N/A	-0.22 µs	-1.70 µs	-0.36 µs	-0.31 µs
All Same Character	Naive	4.29	N/A	-0.90 µs	-2.07 µs	-0.77 µs	+1.19 µs
Alternating Pattern	Naive	5.57	N/A	-0.54 µs	-3.34 µs	+0.00 µs	+0.88 µs
Long Pattern	BoyerMoore	2.79	+0.79 µs	+0.33 µs	-2.64 µs	N/A	-0.44 µs
Pattern at Boundaries	Naive	0.58	N/A	-0.24 µs	-1.41 µs	-0.23 µs	-0.26 µs
Near Matches	Naive	0.79	N/A	+0.37 µs	-0.41 µs	+0.13 µs	+0.15 µs
Empty Pattern	Naive	0.66	N/A	-0.19 µs	+0.32 µs	+0.30 µs	+0.11 µs
Empty Text	Naive	0.15	N/A	+0.08 µs	+0.08 µs	+0.08 µs	-0.49 µs
Both Empty	Naive	0.23	N/A	+0.10 µs	-0.11 µs	+0.09 µs	-0.12 µs
Single Character Pattern	Naive	2.48	N/A	-0.44 µs	-1.79 µs	-0.30 µs	+0.03 µs
Complex Overlap	KMP	1.46	+0.79 µs	+0.41 µs	-0.52 µs	+0.46 µs	N/A
DNA Sequence	Naive	3.07	N/A	+0.07 µs	-1.60 µs	+0.73 µs	+0.30 µs
Palindrome Pattern	Naive	1.46	N/A	+0.14 µs	-1.37 µs	+0.41 µs	-0.39 µs
Worst Case for Naive	KMP	2.48	-2.98 µs	+0.97 µs	-3.06 µs	-0.22 µs	N/A
Best Case for Boyer-Moore	Naive	1.35	N/A	+0.13 µs	-1.18 µs	+0.38 µs	-0.98 µs
KMP Advantage Case	KMP	2.43	-1.64 µs	+1.14 µs	-1.53 µs	+1.02 µs	N/A

Figure 3: Performance Comparison Chart 1

```
SUMMARY STATISTICS:
================================================================================
RabinKarp      : 30 passed, 0 failed | Avg: 4.615 µs, Min: 0.086 µs, Max: 24.734 µs
GoCrazy        : 30 passed, 0 failed | Avg: 11.581 µs, Min: 0.079 µs, Max: 42.549 µs
BoyerMoore     : 30 passed, 0 failed | Avg: 7.299 µs, Min: 0.186 µs, Max: 50.024 µs
KMP            : 30 passed, 0 failed | Avg: 3.736 µs, Min: 0.216 µs, Max: 18.699 µs
Naive          : 30 passed, 0 failed | Avg: 3.758 µs, Min: 0.142 µs, Max: 15.586 µs
================================================================================
```

Figure 4: Performance Comparison Chart 2

DETAILED TEST RESULTS - Execution Time Comparison (Average of 5 runs)

Test Case	RabinKarp (µs)	GoCrazy (µs)	BoyerMoore (µs)	KMP (µs)	Naive (µs)	Winner
Simple Match	6.872	27.764	14.453	8.400	4.342	Naive
No Match	1.863	7.367	5.425	1.966	1.008	Naive
Single Character	9.783	25.364	20.349	7.468	10.883	KMP
Pattern at End	2.876	15.570	6.100	2.099	1.209	Naive
Pattern at Beginning	2.568	11.698	4.801	2.516	1.187	Naive
Overlapping Patterns	3.564	9.280	4.652	2.851	2.781	Naive
Long Text Multiple Matches	8.295	26.149	10.821	8.556	5.420	Naive
Pattern Longer Than Text	0.201	0.547	0.186	1.581	0.252	BoyerMoore
Entire Text Match	1.757	8.743	4.604	1.833	0.963	Naive
Repeating Pattern	4.201	19.881	2.552	1.563	1.523	Naive
Case Sensitive	1.608	5.807	2.379	1.479	1.126	Naive
Numbers and Special Characters	2.414	7.597	50.024	1.474	1.172	Naive
Unicode Characters	5.474	9.174	4.057	1.937	1.109	Naive
Very Long Text	24.734	42.549	18.735	18.699	15.586	Naive
Pattern with Spaces	2.138	4.441	2.258	1.907	1.443	Naive
All Same Character	8.601	12.206	10.371	4.864	7.627	KMP
Alternating Pattern	13.875	19.585	10.697	8.386	10.259	KMP
Long Pattern	7.851	14.858	5.273	5.715	4.355	Naive
Pattern at Boundaries	1.918	4.989	2.087	1.449	1.303	Naive
Near Matches	1.191	3.216	1.645	1.026	1.268	KMP
Empty Pattern	0.689	0.806	0.802	0.822	0.637	Naive
Empty Text	0.086	0.079	0.302	0.269	0.142	GoCrazy
Both Empty	0.220	0.195	0.211	0.216	0.421	GoCrazy
Single Character Pattern	5.572	9.352	6.526	4.385	4.425	KMP
Complex Overlap	1.967	4.436	2.409	1.599	1.369	Naive
DNA Sequence	7.075	12.371	5.009	4.769	5.379	KMP
Palindrome Pattern	3.615	7.425	3.791	2.968	2.116	Naive
Worst Case for Naive	4.231	19.219	8.957	3.966	11.704	KMP
Best Case for Boyer-Moore	1.601	7.594	3.459	3.461	2.480	RabinKarp
KMP Advantage Case	1.614	9.168	6.036	3.862	9.249	RabinKarp

Figure 5: Performance Comparison Chart 3

6 Our Journey

After all the turmoil in the lab exam, where nobody really got what had been wanted, we studied theoretically learning those String matching algorithms before the midterm exam after officially Assoc. Prof. Nar declared one of the midterm questions was going to be about String Matching algorithms. That was our advantage because before starting to code, we already knew how those algorithms work.

In the coding part, Sıla did most parts such as implementing Boyer-Moore, GoCrazy, and other stuff. We studied on the project one by one. Sefa fixed the bugs such as unicode character errors and optimized code by decreasing the time by using `HashMap`, `HashSet`, and other predefined structures mainly decreasing the space complexity. We used the help of Gemini in the designing of `PreAnalysis.java` because we both did not understand what to do there exactly. Also, while designing algorithms we found that we learnt some parts wrong, so AI helped us to figure out and fix coding errors.