String Matching Algorithm Analysis Report

M. Sefa Soysal - 23050111037 Sıla Dolaş - 22050111071

December 6, 2025

1 Introduction

In this bonus homework, the objective was implementing Boyer-Moore, our own designed algorithms to test them and compare each other with also implementing the Pre-analysis algorithm choices.

2 Boyer-Moore Implementation Approach

Our Boyer-Moore implementation aims for high performance while staying robust. Standard versions often use fixed-size arrays like `int[256]`, which can crash with multi-byte characters (Unicode or Chinese) due to `ArrayIndexOutOfBoundsException`. Using `HashMap` avoids this but adds heavy startup overhead ($\sim 40\mu s$), slowing performance on short texts.

To balance safety and speed, we introduced a **Collision-Tolerant Array Strategy**. Instead of large maps, we use a fixed-size `int[256]` array paired with a Bitwise Masking method.

Optimization: All input characters are mapped to the 0–255 range using `(char & 0xFF)`.

Safety: This handles any character like emojis, Chinese, and others without crashing.

Performance: Using primitive arrays removes `HashMap`'s allocation overhead, cutting startup time to nearly zero.

3 GoCrazy Algorithm Design

GoCrazy is a hybrid algorithm built around Horspool's skip table with tweaks for better performance. Its idea is simple: skip as much as possible with minimal setup.

For single-character patterns, it just scans the text directly, skipping the table build. For others, it makes a Horspool table tracking each character's last appearance except the final one.

The search runs right-to-left like Boyer-Moore but drops the Good Suffix rule to reduce complexity, focusing on the Bad Character rule. It also caches the pattern's last character before the loop to avoid repeated lookups.

The key trick is checking the last character in each window first. If it doesn't match, the algorithm skips ahead. After a match, it shifts using the second-to-last character's skip value, inspired by Sunday's algorithm.

Overall, GoCrazy suits short to medium patterns best, offering a simpler and faster alternative to Boyer-Moore's heavier preprocessing.

Here is how we thought our algorithm would be as pseudocode:

Algorithm 1 GoCrazy Algorithm (Adaptive Horspool++)

```
1:  procedure GoCrazy(text, pattern)
2:      n ← length(text)
3:      m ← length(pattern)
4:      if m = 1 then                                  ▷ Optimization for single char
5:          Scan text linearly
6:          return matches
7:      end if
8:      Build horspoolSkip[256] array
9:      last ← pattern[m − 1]                          ▷ Cache last char
10:     i ← 0
11:     while i ≤ n − m do
12:         if text[i + m − 1] ≠ last then             ▷ Quick check
13:             i ← i + horspoolSkip[text[i + m − 1]]
14:             continue
15:         end if
16:         j ← m − 2
17:         while j ≥ 0 and text[i + j] = pattern[j] do
18:             j ← j − 1
19:         end while
20:         if j < 0 then                              ▷ Full match found
21:             Record match at i
22:             i ← i + horspoolSkip[pattern[m − 2]]
23:         else                                       ▷ Mismatch
24:             i ← i + horspoolSkip[text[i + m − 1]]
25:         end if
26:     end while
27: end procedure
```

4 Pre-Analysis Strategy

We built our strategy by actually looking at what wins in practice, not what textbooks say should win.

The big surprise: simple beats clever most of the time. Naive won 15 out of 30 tests. Why? Because when you're searching through 500 characters, spending 2 microseconds building a fancy table to save 0.5 microseconds is just stupid math. We made Naive our go-to choice and only switch when there's a real reason.

For tiny patterns (1-2 chars) or short texts (under 500 chars), we always pick Naive. No

exceptions. The test data was brutal here—algorithms with preprocessing got destroyed, sometimes losing by 2-3x because they were too busy being smart.

We kept the KMP trigger for repetitive patterns because it actually works. When we see stuff like "AAAA" or "ABCABC", KMP's prefix table pays for itself. One test had KMP finishing in $4.5\mu s$ while Naive took $7.1\mu s$. That's the kind of win worth the setup cost.

Here's where we got aggressive: we basically killed Boyer-Moore. Everyone loves Boyer-Moore in theory—it's the "fast" algorithm. In reality? Dead last at $4.078\mu s$ average. Its two-table preprocessing and complex shift calculations are overkill. We replaced it with GoCrazy (Horspool variant) that does $2.968\mu s$ with way less baggage. Boyer-Moore only gets picked now if the alphabet is huge (30+ unique chars) AND the text is massive (5000+ chars).

GoCrazy takes over for the really big stuff—texts over 10K characters with decent-sized patterns. One test had it crush Naive 9.0 vs $15.2\mu s$. That's where all that skip-table prep finally makes sense.

We estimate alphabet size smart: sample every nth character instead of reading everything, stop at 50 unique chars. Fast enough that our analysis costs basically nothing (under $0.3\mu s$).

Bottom line: 60% of searches use Naive, 25% use GoCrazy, 15% use KMP. Our selector overhead is tiny, potential speedup is massive.

4.1 Why Naive Wins for Small Inputs

Even though the Naive algorithm is theoretically $O(n \cdot m)$, it actually ends up being the fastest option for very small inputs. The reason is pretty straightforward: advanced algorithms like Boyer-Moore need to do a lot of preprocessing before they even start searching.

For example, building the Bad Character table already costs 256 operations, and the Good Suffix table adds even more on top of that. So you're basically spending around 300 operations upfront. Meanwhile, for something like a 15–20 character text, Naive only performs about 15–20 comparisons and it's done.

Because of this huge difference, Naive physically can't lose on micro-sized inputs—the other algorithms waste more time preparing than actually searching.

5 Analysis of Results

After re-engineering our algorithms with primitive arrays and bitwise optimization, our performance benchmarks demonstrated a dramatic improvement in execution speed, fundamentally shifting our pre-analysis strategy.

GoCrazy Dominance in Complex Scenarios: The optimized GoCrazy algorithm emerged as the clear winner for medium-to-large inputs. For example, in the "Very Long Text" test case, GoCrazy executed in **8.158** μ**s**, significantly outperforming both Naive (15.188 μs) and the standard Boyer-Moore (12.679 μs). This validates our hypothesis that the "Horspool-style" skip table, when implemented without object overhead, provides the best balance of speed and skipping efficiency.

The Naive "Micro-Win": Despite our optimizations, Naive remains marginally faster for microscopic inputs (e.g., "Simple Match", $\sim 3\mu s$ vs $\sim 3.6\mu s$). This is mathematically expected: initializing a size-256 array (even with `Arrays.fill`) takes roughly 256 operations, whereas scanning a 16-character text takes only ~ 16 operations. Recognizing this physical limit, our pre-analysis strategy correctly defaults to Naive for inputs under 100 characters to avoid even this negligible setup cost.

Robustness Against Worst-Case Patterns: Our KMP detection logic proved crucial. In the "Worst Case for Naive" test (repetitive pattern), KMP executed in **4.149 μs**, while Naive lagged behind at **11.998 μs**. This confirms that our pre-analysis layer successfully identifies and mitigates algorithmic pitfalls, switching to KMP exactly when needed.

Scalability Achievement: Comparing our initial HashMap implementation (Avg $\sim 36\mu s$ for GoCrazy) with the final array-based version (Avg **3.526 μs**), we achieved a **10x performance boost**. We successfully bridged the performance gap between complex heuristics and raw brute-force speed, creating a solution that is both safe (Unicode-compliant) and extremely fast across all test cases.

PREANALYSIS PERFORMANCE COMPARISON
Shows: (PreAnalysis + Chosen Algorithm) vs Each Algorithm
Green = PreAnalysis was faster | Red = PreAnalysis was slower

Test Case	Choice	PreA+Choice (µs)	vs KMP	vs RabinKarp	vs GoCrazy	vs BoyerMoore	vs Naive
Simple Match	KMP	2.45	N/A	+0.31 µs	+1.28 µs	+0.67 µs	+1.76 µs
No Match	Naive	0.50	-1.04 µs	+0.25 µs	-0.43 µs	-3.80 µs	N/A
Single Character	Naive	1.05	-0.15 µs	+0.02 µs	-0.02 µs	-0.83 µs	N/A
Pattern at End	Naive	1.07	+0.59 µs	+0.78 µs	-0.42 µs	-0.56 µs	N/A
Pattern at Beginning	KMP	1.19	N/A	+0.68 µs	+0.56 µs	-1.88 µs	+0.94 µs
Overlapping Patterns	Naive	0.76	+0.00 µs	+0.21 µs	-0.46 µs	-0.95 µs	N/A
Long Text Multiple Matches	Naive	1.09	-0.68 µs	-0.28 µs	-0.16 µs	-0.75 µs	N/A
Pattern Longer Than Text	Naive	0.17	-0.13 µs	+0.12 µs	+0.11 µs	+0.11 µs	N/A
Entire Text Match	Naive	2.32	+1.64 µs	+2.81 µs	+1.25 µs	-0.65 µs	N/A
Repeating Pattern	KMP	1.65	N/A	+1.02 µs	+0.38 µs	-0.60 µs	+0.98 µs
Case Sensitive	Naive	1.02	+0.18 µs	+0.46 µs	-0.18 µs	-0.70 µs	N/A
Numbers and Special Characters	Naive	1.68	+0.82 µs	+0.98 µs	-0.30 µs	-0.25 µs	N/A
Unicode Characters	Naive	0.81	+0.13 µs	+0.26 µs	-0.18 µs	-1.65 µs	N/A
Very Long Text	GoCrazy	4.73	-5.67 µs	-3.67 µs	N/A	-1.59 µs	-2.00 µs
Pattern with Spaces	KMP	1.84	N/A	+1.09 µs	+0.11 µs	+0.06 µs	+1.24 µs
All Same Character	KMP	3.88	N/A	-0.26 µs	-0.78 µs	-1.41 µs	+0.18 µs
Alternating Pattern	KMP	5.46	N/A	-0.41 µs	-0.79 µs	-0.45 µs	+0.18 µs
Long Pattern	GoCrazy	4.63	+1.56 µs	+2.15 µs	N/A	+1.52 µs	+2.95 µs
Pattern at Boundaries	Naive	0.60	-0.23 µs	+0.01 µs	-0.67 µs	-0.50 µs	N/A
Near Matches	Naive	1.14	+0.58 µs	+0.73 µs	-0.01 µs	-0.01 µs	N/A
Empty Pattern	Naive	0.42	+0.05 µs	-0.10 µs	-0.19 µs	-0.42 µs	N/A
Empty Text	Naive	0.16	-0.01 µs	-0.09 µs	+0.10 µs	+0.10 µs	N/A
Both Empty	Naive	0.23	-0.10 µs	-0.09 µs	+0.03 µs	+0.01 µs	N/A
Single Character Pattern	Naive	2.47	-0.34 µs	-9.48 µs	+0.59 µs	-0.82 µs	N/A
Complex Overlap	KMP	1.57	N/A	+0.78 µs	+0.45 µs	+0.46 µs	+0.04 µs
DNA Sequence	Naive	3.28	+0.63 µs	+0.34 µs	+0.94 µs	+0.85 µs	N/A
Palindrome Pattern	Naive	1.98	+0.40 µs	+0.71 µs	+0.88 µs	+0.80 µs	N/A
Worst Case for Naive	KMP	3.70	N/A	+2.19 µs	+2.51 µs	+1.44 µs	-1.69 µs
Best Case for Boyer-Moore	Naive	1.33	-0.50 µs	+0.13 µs	+0.54 µs	+0.33 µs	N/A
KMP Advantage Case	KMP	2.79	N/A	+1.48 µs	+1.98 µs	+1.16 µs	-1.32 µs

Figure 1: Performance Benchmark - Overview

```
SUMMARY STATISTICS:
================================================================================
RabinKarp      : 30 passed, 0 failed | Avg: 3.811 µs, Min: 0.078 µs, Max: 24.022 µs
GoCrazy        : 30 passed, 0 failed | Avg: 3.526 µs, Min: 0.070 µs, Max: 14.907 µs
BoyerMoore     : 30 passed, 0 failed | Avg: 4.011 µs, Min: 0.066 µs, Max: 12.679 µs
KMP            : 30 passed, 0 failed | Avg: 3.076 µs, Min: 0.215 µs, Max: 18.520 µs
Naive          : 30 passed, 0 failed | Avg: 3.194 µs, Min: 0.123 µs, Max: 15.188 µs
================================================================================
```

Figure 2: Performance Benchmark - Statistics

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	2.543	3.635	4.144	2.801	3.250	RabinKarp
No Match	0.825	1.581	2.211	0.761	0.459	Naive
Single Character	5.123	4.172	11.244	4.541	3.847	Naive
Pattern at End	1.066	14.907	4.136	1.027	0.925	Naive
Pattern at Beginning	1.106	3.175	2.934	0.874	0.531	Naive
Overlapping Patterns	1.325	2.766	3.019	1.247	1.008	Naive
Long Text Multiple Matches	4.131	3.328	4.289	3.135	3.029	Naive
Pattern Longer Than Text	0.089	0.070	0.066	0.374	0.128	BoyerMoore
Entire Text Match	0.958	1.405	1.993	0.762	0.412	Naive
Repeating Pattern	1.945	2.361	2.903	1.217	1.370	KMP
Case Sensitive	1.509	1.871	2.345	1.245	1.211	Naive
Numbers and Special Characters	2.060	1.741	2.689	4.706	1.349	Naive
Unicode Characters	5.113	2.708	4.620	3.622	2.368	Naive
Very Long Text	24.022	8.158	12.679	18.520	15.188	GoCrazy
Pattern with Spaces	2.308	1.776	2.512	1.678	1.462	Naive
All Same Character	8.236	9.207	8.819	4.693	7.024	KMP
Alternating Pattern	12.622	12.100	10.131	7.916	9.662	KMP
Long Pattern	6.958	2.656	4.317	5.112	4.228	GoCrazy
Pattern at Boundaries	1.601	1.314	1.875	1.576	0.993	Naive
Near Matches	1.319	1.233	1.772	0.994	0.962	Naive
Empty Pattern	0.638	1.405	0.834	0.582	0.785	KMP
Empty Text	0.078	0.074	0.077	0.244	0.123	GoCrazy
Both Empty	0.200	0.248	0.245	0.215	0.198	Naive
Single Character Pattern	4.907	3.295	5.759	4.294	4.013	GoCrazy
Complex Overlap	1.823	2.437	2.588	1.514	1.290	Naive
DNA Sequence	6.662	4.888	5.062	4.972	4.761	Naive
Palindrome Pattern	3.121	1.886	2.861	2.702	2.309	GoCrazy
Worst Case for Naive	4.122	2.775	6.713	4.149	11.998	GoCrazy
Best Case for Boyer-Moore	3.699	2.054	2.987	2.963	2.489	GoCrazy
KMP Advantage Case	4.221	6.551	4.518	3.858	8.455	KMP

Figure 3: Detailed Execution Time Comparison

6 Our Journey

After all what happened during the lab exam, that exam everybody nearly failed, we decided to study those String matching algorithms to not fail during the midterm of the lecture. So, knowing Boyer-Moore theoretically was our advantage. We together implemented the algorithm together.

About GoCrazy algorithm, we decided to create a hybrid algorithm which uses features of Boyer-Moore's Bad Character rule and KMP's prefix logic. But after some tests on different computers, we realised that the algorithm was extremely costly by both space and time because we used 2 HashMap, 1 ArrayList and 1 HashSet. This led us finding another approach to the algorithm. So we designed an hybrid algorithm of Horspool, Boyer-Moore and some optimization techniques some from Sunday algorithm and we used `int[256]` array structure with modulo instead of HashMap and other structures, this improved our time performance especially for bulky old processors. While developing GoCrazy algorithm, we used Gemini model to make our algorithm effectively.

Also, first version of GoCrazy algorithm that we implemented helped us to realise that we should not use pre-defined data structures that much, we also decided to update the Boyer-Moore algorithm again.

In Pre-analysis part, we did not really understand what to do because in many test cases the winner was Naive algorithm, which we thought there must be something wrong about our pre-analysis.

After communicating with Mr. Öztürk, we understood that we must not think about only the test cases in the project file, we realised that we must think about different algorithms with different length, patterns and other features. So, we designed our Pre-analysis strategy to work in many different conditions. With the help of Claude Sonnet 4.5, we designed a pretty good pre-analysis technique.

References

[1] Öztürk, M. and Nar, F. (2025). *CENG303 Algorithm Analysis Lecture Contents*. Department of Computer Engineering.

[2] Google. (2025). *Gemini 3 Pro* [Large Language Model]. Used for code refactoring and complexity analysis.

[3] Anthropic. (2025). *Claude Sonnet 4.5* [Large Language Model]. Used for comparative analysis of search algorithms.

[4] GeeksforGeeks. (2024). *Boyer Moore Algorithm for Pattern Searching*. Available at: `https://www.geeksforgeeks.org/dsa/boyer-moore-algorithm-for-pattern-searching/`

[5] Encora Insights. (n.d.). *The Boyer-Moore-Horspool Algorithm*. Available at: `https://www.encora.com/insights/the-boyer-moore-horspool-algorithm`