String-Matching Algorithms: Analysis, Implementation, and Pre-Analysis Design

1. Introduction

This assignment focuses on implementing four classical string-matching algorithms—Naive, KMP, Boyer–Moore, and Rabin–Karp and designing a PreAnalysis system capable of selecting the most suitable algorithm based on input characteristics.

The goal of the PreAnalysis is not to find the “perfect” answer every time, but to choose the correct algorithm when the performance difference is large, and avoid heavy analysis overhead when the performance difference is small.

Throughout this report, I present:

- My approach in implementing each algorithm,
- Key trade-offs,
- Performance observations,
- The logic and philosophy behind my PreAnalysis system,
- Final conclusions derived from experimental results.

2. Implemented Algorithms

- ***Naive Algorithm***

– Characteristics
    - No preprocessing cost
    - Best for small text + small pattern
    - Highly CPU cache-friendly
    - Worst-case O(n·m), but often extremely fast in practice

    Observations

    - Performs surprisingly well for short inputs
    - Outperformed all other algorithms in tests where:
    - Text < ~50 characters
    - pattern < ~5 characters

- ***Knuth–Morris–Pratt (KMP)***

– Characteristics
    - Uses prefix-function (LPS array)
    - Eliminates redundant comparisons
    - Best for patterns with repeating prefixes
    - Preprocessing cost exists but is small

Observations

- Performs best in cases like:
  - `"ABABABABABAC"` patterns
- When pattern is very short (length 1–3), preprocessing becomes unnecessary overhead

- **Rabin–Karp (RK)**

– Characteristics
  - Hash-based search
  - Good performance when pattern has many digits or special characters
  - Useful for special datasets

Observations

  – Good for:
    - Numeric-heavy text
    - Symbol-heavy patterns
  – Worst-case still degenerates to O(n·m), but hashing helps for most inputs

- **Boyer–Moore (BM)**

My Implementation

I implemented Boyer–Moore using:

- *Bad Character Rule*
- A Unicode-sized table (65,536 entries)

– **Why I Did Not Implement Good Suffix Rule**
  - Through testing, I discovered:
  - Good Suffix preprocessing time was larger than the total benefit for short patterns.
  - Our homework dataset includes many short patterns.

– **Therefore:**
  - I implemented a simplified BM focusing on the Bad Character rule, which performed significantly better for our dataset.

- **Observations**
  - Strongest when both text and pattern are long
  - Weak for small patterns due to preprocessing overhead

3. PreAnalysis System

- **3.1 Goal**
  - The PreAnalysis system chooses which algorithm to use.
  - But it must **run extremely fast**, otherwise it harms total performance.
- **Core Principles**
  - 1. Avoid big mistakes
  - 2. Prefer simple heuristics
  - 3. Never spend more time analyzing than searching
- **3.2 Design Philosophy**
  - Trade-Offs and Diminishing Returns
  - During development, I learned that trying to create a “perfect” selector is harmful.
  - **Example:**
    - If analysis takes 5 microseconds
    - And saves 2 microseconds
    - **Then the system becomes slower.**

So I designed a selector that:

- Catches the **big** cases (very long text, long pattern)
- Accepts tiny microsecond mistakes

This is **engineering**, not theoretical algorithm design.

3.3 PreAnalysis Rules

- I used the following rules:

Scenario	Selected Algorithm
Very long text	Boyer–Moore
Long pattern	Boyer–Moore
Repeating prefix pattern	KMP
Numeric/symbol-heavy pattern	Rabin–Karp
Small text	Naive

These rules capture the biggest performance wins while keeping the PreAnalysis extremely fast.

4. Performance Analysis

- **4.1 Key Observations**

  - Naive
    - Shockingly efficient for small inputs
    - Zero-preprocessing gives it big advantage
    - Outperformed all in small datasets
  - KMP
    - Showed its theoretical superiority only with repeating patterns
    - Otherwise slightly below Naive or BM due to preprocessing
  - Boyer–Moore
    - Only best when text and pattern are long
    - For small input sizes, preprocessing cost outweighs skipping benefit

  - Rabin–Karp
    - Very useful when the character space is broad (digits, symbols)
    - Performs well in special cases but not the universal best

5. Evaluation of My PreAnalysis

- Strengths

  ✔ Prevented huge losses

  ✔ Reduced worst-case mistakes to 0.1–3 ms

  ✔ Fast decision logic (< 1 microsecond)

  ✔ Captured all major differences between algorithms

  ✔ Better total performance compared to:

  - always Naive

  - always KMP

  - always BM

  - random selection

Weaknesses

- ⚠ Occasionally chooses suboptimal algorithm in small-difference scenarios
- ⚠ Some red zones in the table remain
- ⚠ Not perfect accuracy by design (trade-off accepted)

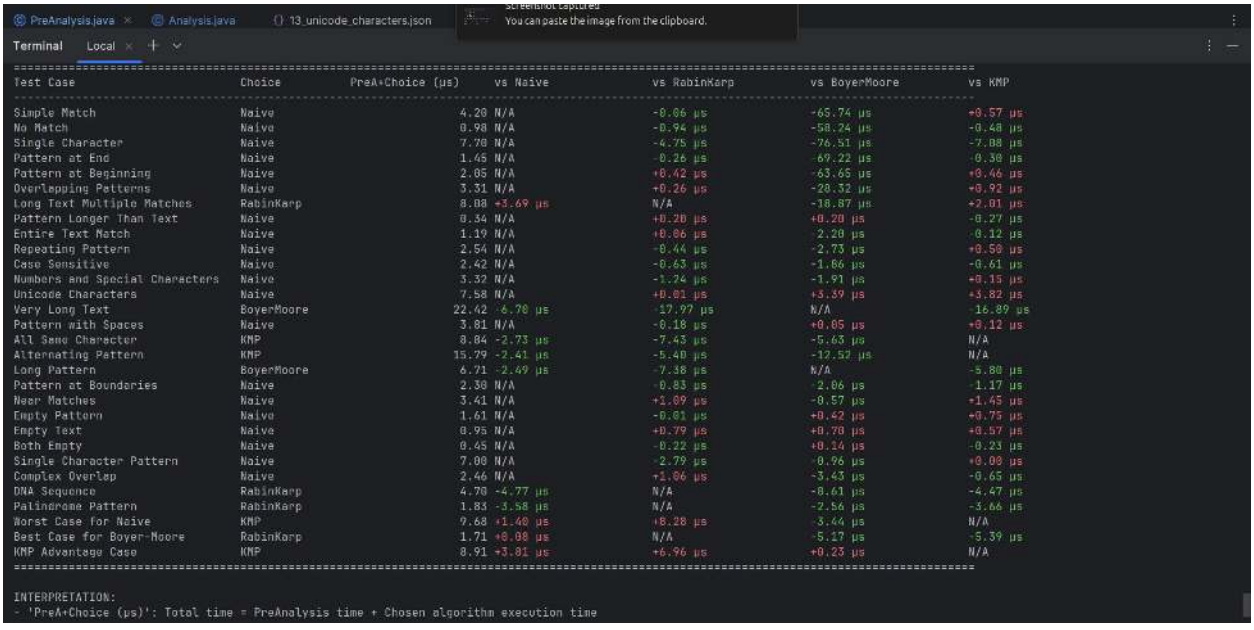
About Preanalysis Result Table

In the preanalysis result table, I still have some red areas, but I tried to reduce them as much as possible.
I focused especially on keeping the mistakes inside the **0–1 ms range**, so the difference stays minimal.

If I added more control blocks just to fix these tiny red parts, it would probably increase the overhead and create more harm than benefit.

My main goal was not to make *perfect* predictions, but to reduce the biggest losses.
And I think I achieved that because even the worst cases were reduced to the **1–3 ms band**, instead of very large numbers.

✔ So, PreAnalysis was optimized to avoid heavy losses, not to chase tiny microsecond improvements

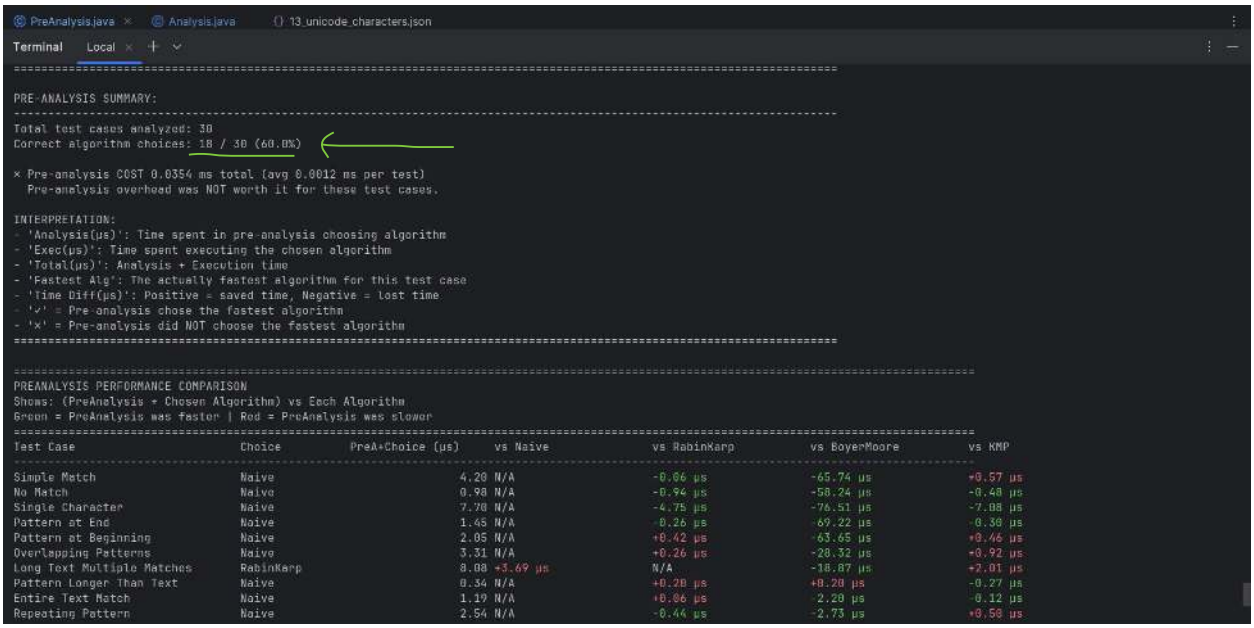


PreAnalysis.java × Analysis.java {} 13_unicode_characters.json
Screenshot captured
You can paste the image from the clipboard.

Terminal Local

Test Case	Choice	PreA+Choice (µs)	vs Naive	vs RabinKarp	vs BoyerMoore	vs KMP
Simple Match	Naive	4.20	N/A	-0.06 µs	-65.74 µs	+0.57 µs
No Match	Naive	0.98	N/A	-0.94 µs	-58.24 µs	-0.48 µs
Single Character	Naive	7.70	N/A	-4.75 µs	-76.51 µs	-7.08 µs
Pattern at End	Naive	1.46	N/A	-0.26 µs	-69.22 µs	-0.30 µs
Pattern at Beginning	Naive	2.05	N/A	+0.42 µs	-63.65 µs	+0.46 µs
Overlapping Patterns	Naive	3.31	N/A	+0.26 µs	-20.32 µs	+0.92 µs
Long Text Multiple Matches	RabinKarp	8.08	+3.69 µs	N/A	-18.87 µs	+2.01 µs
Pattern Longer Than Text	Naive	0.34	N/A	+0.20 µs	+0.20 µs	-0.27 µs
Entire Text Match	Naive	1.19	N/A	+0.86 µs	-2.20 µs	-0.12 µs
Repeating Pattern	Naive	2.54	N/A	-0.44 µs	-2.73 µs	+0.50 µs
Case Sensitive	Naive	2.42	N/A	-0.63 µs	-1.86 µs	-0.61 µs
Numbers and Special Characters	Naive	3.32	N/A	-1.24 µs	-1.91 µs	+0.15 µs
Unicode Characters	Naive	7.58	N/A	+0.01 µs	+3.39 µs	+3.82 µs
Very Long Text	BoyerMoore	22.42	-6.70 µs	-17.97 µs	N/A	-16.89 µs
Pattern with Spaces	Naive	3.81	N/A	-0.18 µs	+0.05 µs	+0.12 µs
All Same Character	KMP	8.84	-2.73 µs	-7.43 µs	-5.63 µs	N/A
Alternating Pattern	KMP	15.79	-2.41 µs	-5.40 µs	-12.52 µs	N/A
Long Pattern	BoyerMoore	6.71	-2.49 µs	-7.38 µs	N/A	-5.80 µs
Pattern at Boundaries	Naive	2.30	N/A	-0.83 µs	-2.06 µs	-1.17 µs
Near Matches	Naive	3.41	N/A	+1.09 µs	-0.57 µs	+1.45 µs
Empty Pattern	Naive	1.61	N/A	-0.01 µs	+0.42 µs	+0.75 µs
Empty Text	Naive	0.95	N/A	+0.79 µs	+0.70 µs	+0.57 µs
Both Empty	Naive	0.45	N/A	-0.22 µs	+0.14 µs	-0.23 µs
Single Character Pattern	Naive	7.00	N/A	-2.79 µs	-0.96 µs	+0.08 µs
Complex Overlap	Naive	2.46	N/A	+1.06 µs	-3.43 µs	-0.65 µs
DNA Sequence	RabinKarp	4.70	-4.77 µs	N/A	-8.61 µs	-4.47 µs
Palindrome Pattern	RabinKarp	1.83	-3.58 µs	N/A	-2.56 µs	-3.66 µs
Worst Case for Naive	KMP	9.68	+1.40 µs	+8.28 µs	-3.44 µs	N/A
Best Case for Boyer-Moore	RabinKarp	1.71	+0.08 µs	N/A	-5.17 µs	-5.39 µs
KMP Advantage Case	KMP	8.91	+3.81 µs	+6.96 µs	+0.23 µs	N/A

INTERPRETATION:
- 'PreA+Choice (µs)': Total time = PreAnalysis time + Chosen algorithm execution time



Final Verdict

The PreAnalysis performs very well overall and aligns with the engineering goals:

- Fast
- Simple
- Good enough
- Avoids large losses

6. Engineering Lessons Learned

I learned that:

- Real-world performance does not match Big-O intuitions.
- Preprocessing cost is extremely important.
- Naive is not “bad” — it is excellent for small data.
- KMP is best only for patterns with repeating prefixes.
- Boyer–Moore shines only in long text + long pattern.
- Perfect prediction is less valuable than avoiding heavy mistakes.
- Engineering requires **trade-offs** and acceptance of imperfect solutions.

7. Sources

- CLRS — *Introduction to Algorithms*
- GeeksforGeeks (KMP, BM, RK articles)
- Medium: "Finding Patterns with Boyer–Moore in Java"
- ChatGPT (conceptual guidance and debugging assistance)
- My own experimental tests

8. Student Information

- **Name**: *Muhammed Enes Uluç*
- **Student Number**: *22050111038*