

Gökmen Balci
22050111046
My Journey and Project Report

1- How I Write BoyerMoore

Firstly i use this site for boyermooore implementation

<https://www.geeksforgeeks.org/dsa/boyer-moore-algorithm-for-pattern-searching/>

then i write boyermooore algorithm with this site but it giving errors in some cases like unicode characters so i use map for Bad Char Heuristic because map allow us to handle special characters without a huge array

2- How I Construct PreAnalysis

Firstly i ask gemini

can you explain to me which string matching algorithm is more efficient for these cases

Simple Match

No Match

Single Character

Pattern at End

Pattern at Beginning

Overlapping Patterns

Long Text Multiple Matches

Pattern Longer Than Text

Entire Text Match

Repeating Pattern

Then gemini give me this answers

Scenario	Winner	Why?
Simple Match	Naive	Zero preprocessing overhead; fastest for short strings.
No Match	Boyer-Moore	Skips large sections of text (sub-linear time).

Single Character	Naive	Simple linear scan; no complex tables needed.
Pattern at End	Boyer-Moor e	Jumps through the beginning of the text rapidly.
Pattern at Beginning	Naive	Matches immediately without calculating lookup tables.
Overlapping Patterns	KMP	Uses the LPS (Longest Prefix Suffix) table to slide correctly without rescanning.
Long Text Multiple Matches	Boyer-Moor e	Generally the fastest in practice due to character skipping.
Pattern Longer Than Text	All	All should include a trivial $O(1)$ length check before starting.
Entire Text Match	Naive / Rabin-Karp	Naive is a simple loop. Rabin-Karp is fast if hashes are pre-calculated.
Repeating Pattern	KMP	Guaranteed $O(N)$ time; never rescans text characters (unlike Naive/BM).

Detailed Case Analysis

1. Naive Algorithm

Best for: Simple Match, Single Character, Pattern at Beginning, Entire Text Match.

The Naive approach simply checks every index.

- **Single Character / Simple Match:** Complex algorithms like KMP or Boyer-Moore require time to build "jump tables" or "hash values" before they even start scanning. If the text is short or the pattern is just one letter, this setup time is a waste. The Naive loop starts immediately.
- **Pattern at Beginning:** Naive finds the match in $\$O(M)$ (length of pattern) immediately. KMP/BM would also find it in $\$O(M)$, but they would waste time preprocessing the pattern first.

2. Boyer-Moore (BM)

Best for: No Match, Pattern at End, Long Text Multiple Matches.

Boyer-Moore is famous for the "Bad Character Heuristic." It scans the pattern from **right to left**.

- **No Match / Pattern at End:** If the algorithm looks at a character in the text that doesn't exist in the pattern at all, it can **skip** the entire length of the pattern forward.
 - *Example:* Searching for "ZEBRA" in "APPLE BANANA..."
 - It looks at 'E' (from APPLE). 'E' is in ZEBRA, so it shifts alignment.
 - If it hits a character like 'X', it knows 'X' isn't in "ZEBRA" and jumps 5 spots instantly.
- **Efficiency:** In these cases, it can run in $\$O(N/M)$ time (sub-linear), making it significantly faster than KMP or Naive.

3. Knuth-Morris-Pratt (KMP)

Best for: Repeating Pattern, Overlapping Patterns.

KMP is the "safe" option. It guarantees linear time $\$O(N)$ in the worst-case scenarios where others fail.

- **Repeating Pattern:** Imagine searching for AAAAB inside AAAAAAAAAAAAB.
 - **Naive:** Will check AAAA... fail, shift 1, check AAAA... fail. This creates $\$O(N \cdot M)$ complexity (very slow).

- **Boyer-Moore:** Can also degrade to $O(N \cdot M)$ in specific repeating cases (though improved versions avoid this).
- **KMP:** Uses a prefix table to know exactly how far to shift. It never re-scans a character it has already matched. It stays strictly $O(N)$.
- **Overlapping:** KMP is designed to report a match and immediately know the correct "reset" position to find the next overlapping match without backing up.

4. Rabin-Karp (RK)

Best for: *Multiple Patterns (Plagiarism detection), Rolling Hash.*

While not the top winner for single-string searches, RK is unique.

- **Long Text Multiple Matches (Special Case):** If you are searching for **many different patterns** (e.g., checking a document for a list of 50 banned words), Rabin-Karp is superior. It can check for all 50 patterns simultaneously using hashing.
- **Entire Text Match:** If you interpret the text and pattern as numbers (hashes), checking if they are identical is very fast, provided you don't encounter hash collisions.

Algorithm	Worst Case Time	Best Case Time	Space Complexity
Naive	$O(N \cdot M)$	$O(N)$	$O(1)$
Boyer-Moore	$O(N \cdot M)$ (or $O(N)$ optimized)	$O(N/M)$	$O(M + \sigma)$
KMP	$O(N)$	$O(N)$	$O(M)$
Rabin-Karp	$O(N \cdot M)$	$O(N+M)$	$O(1)$

Based on these answers and the results of the compiled code, I wrote a pre-analysis code. However, I also requested code from Gemini, but the code it provided didn't give me the desired results and always returned a pre-analysis code that selected the naive algorithm. So, using the previous information, the example sent by our teacher, and the compiler results, I wrote my own code.

3-How I Construct My Cases In PreAnalysis

Case 1: I choose Naive algorithm for small pattern and text length because naive algorithm is giving best results for this cases.

Case 2: I choose RobinKarp algorithm for zero length pattern or text length because RobinKarp giving best results for these cases as a result of my experiments

Case 3: I choose Kmp algorithm for pattern length is 1 or pattern has a repeating patterns. Because Kmp is best for this cases and i write a helper method for check the repeating patterns.

Case 4: I choose BoyerMoore for default case this case is for long pattern or long text lengths. Because Bayermoore is very good for this cases.

HelperMethod isRepeating: This helper method is for Kmp. Purpose of this method is checking the pattern for any repeating patterns.

My Journey

I didn't have much difficulty with the Boyer-Moore implementation; I both knew how to use it and I used a website where I could easily get help online. However, it was giving errors in some test cases. To fix this, I used a map instead of an array. This way, I didn't need to use a large array and I got rid of the errors.

I really struggled with the pre-analysis phase because I couldn't achieve the desired results. I even used AI to help with these issues, but the AI's suggested code always led to the naive algorithm, which I didn't want because then the assignment would be pointless. So, I created my own algorithm, similar to the example code, by asking the AI which algorithm worked better in which cases. I ran many tests while creating the algorithm, but the naive algorithm always came out on top, which was due to the shorter test length. However, using only the naive algorithm was pointless. By testing different cases, I reached the best possible result, but I only managed about 60%. Although I didn't achieve complete success, I understood which string matching algorithms work better in which situations.

In conclusion, I believe this assignment has helped me improve my skills in string matching algorithms and general algorithm generation.

Sources:

<https://www.geeksforgeeks.org/dsa/boyer-moore-algorithm-for-pattern-searching/>

Gemini