String Matching Homework Report

Mohammad Ahmed Abdullah Ismail

December 12, 2025

1 Boyer–Moore Implementation Approach

My implementation for Boyer-Moore code follows the classic version that uses the bad character rule.

The approach is described below.

1. Before searching, scan the pattern once and build a table that detects for each character, at which index it appears last in the pattern.
2. During the search, align the pattern with the text and compare characters from the end of the pattern towards the start.
3. If all characters match, record the current starting index as a match and then move the pattern forward.
4. If there is a mismatch, look at the character in the text that caused the mismatch. With the table, check where this character appears in the pattern. This tells how far we can shift the pattern so that it does not miss any possible match but also do not move only one step every time.

I also handle two edge cases in the code:

1. If the pattern is empty, I treat it as matching at every position in the text, as the other algorithms in the project do.
2. If the pattern is longer than the text, I return an empty result immediately, because a match is impossible.

The method returns all starting positions as a comma separated string, in the same format as Naive, KMP and RabinKarp. When I ran the test suite, Boyer–Moore produced correct results on all shared test cases.

2 GoCrazy Algorithm Design and Rationale

I did not implement GoCrazy and left it in the default state that throws an exception.

3 Pre-Analysis Strategy

My strategy for the pre-analysis uses the length of the text, the length of the pattern, and a simple check for repeated characters at the start of the pattern. The rules i implemented are:

1. If the pattern return Naive. Any algorithm is fine in this case, and no cost to set up.

2. If the pattern is longer than the text also return Naive, because there can be no matches and Naive will stop quickly.
3. If the pattern length is at most three, I use Naive. For very short patterns, using adavanced algorithms is usually not worth it.
4. detect a repeating prefix with a helper method. It checks how often the first character appears in the first few positions of the pattern. If it appears at least three times in this small prefix, I treat the pattern as having a strong repetition at the start and choose KMP, which is good for patterns with repeated prefixes.
5. If the pattern length is greater than ten and the text length is greater than one thousand, I choose BoyerMoore. For large inputs, the ability to skip ahead by more than one position is helpful.
6. If the pattern length is greater than five and the text length is greater than one hundred, and the previous rule did not trigger, I choose RabinKarp
7. In all other cases, I choose Naive as a default. On the given tests, Naive is very competitive and often the fastest because it has very low overhead.

The helper method that detects a repeating prefix only looks at the first few characters of the pattern and counts how many of them are equal to the first character. The whole pre-analysis method runs quickly because it only does simple checks and a short scan of the pattern.

4 Analysis of Results

All four algorithms (Naive, KMP, RabinKarp and my BoyerMoore implementation) passed the 30 shared test cases. There were no wrong answers or crashes, including on edge cases such as empty text, empty pattern and pattern longer than text.

For the Pre Analysis with my implementation the chosen algorithm is the fastest one in 19 out of 30 shared test cases. This is about 63 percent accuracy on this test set.

The same tool also shows the time spent in pre-analysis. In all tests, the extra time added by pre-analysis was small compared to the time spent by the matching algorithms themselves. The differences are usually only a few microseconds which i think is acceptable

5 Research and use of internet and LLMs

For this homework I used a mix of course material, an online article, and ChatGPT . I started with the lecture notes on string matching to review the basic ideas behind the Naive, KMP, Rabin–Karp and Boyer–Moore algorithms.

For Boyer–Moore, I first read an online tutorial on GeeksforGeeks about the Boyer–Moore algorithm. That article explained the bad character rule and showed how the pattern is scanned from right to left and then shifted based on the mismatched character. Using that explanation, I wrote my first version of the `BoyerMoore` class in `Analysis.java` and tried to follow the same structure: build a last-occurrence table for the pattern and compute the shift after each mismatch.

My first implementation did not work correctly on all tests. Some matches were missing and some of the reported indices were wrong. At that point I used ChatGPT to help understand what was wrong. I described how my code worked and what the incorrect outputs looked like.

ChatGPT pointed out problems in how I calculated the shift. Based on its feedback I rewrote parts of the `Solve` method, simplified the shift logic and made the behaviour consistent with the other algorithms in the project. After these changes, Boyer–Moore passed all of the shared test cases.

For the pre-analysis strategy, I first tried to implement it on my own using the tips from the assignment and my general understanding of the algorithms. I wrote a simple version of the PreAnalysis that looked at the pattern length and text length and made basic decisions, and then I measured its accuracy and this first attempt selected the fastest algorithm in roughly 35% of the shared test cases.

After that, I asked ChatGPT more specifically about when each algorithm is best suited. Using this information, I redesigned my rules so that they matched these use cases more closely. Naive for very short patterns , KMP for patterns with a repeating prefix, Boyer–Moore for long patterns in long texts, and Rabin–Karp a middle ground. With this strategy, the accuracy increased to about 63% on the shared test cases.

After reaching that point, I tried to improve the accuracy even more. I experimented with more complicated conditions, such as counting distinct characters in the pattern and adding extra special cases. However, when I tested these more complex versions, they surprisingly gave lower accuracy than the simpler strategy. At that point I also asked ChatGPT to give me its own full implementation of the pre-analysis logic, but that version shockingly performed worse on the tests. Because of this, I decided to go back to the simpler algorithm

6 Your Journey

When I opened the homework for the first time, I felt completely overwhelmed; there were a lot of Java files and a long wall of explanation. While the explanation of what needed to be implemented was clear, trying to understand how the tests are run and what each class does took me some time.

On top of that, this homework was placed between several other project deadlines from different courses, which made this experience much more stressful.

The Boyer–Moore part wasn't exactly easy, but it was straightforward because with the test cases you can clearly see whether your implementation is correct or it's not. So even though it took some debugging to work, this segment wasn't a problem at all.

The preanalysis part, however, was by far the most painful part of the whole homework. When I first read the requirements I expected it to be pretty easy because I thought all I would need here is some theoretical knowledge and implementing some if/else cases. Unfortunately, I was wrong, as this is the part I spent most of my time on. Trying to push the accuracy of the results beyond 63% was so exhausting, and when I finally gave up and asked ChatGPT for its own full implementation and ended up with worse accuracy results, I decided to give up.

The GoCrazy part was disappointing for me because I spent almost an hour trying to come up with an original algorithm but no ideas came to mind. This made me realise that I have issues coming up with algorithm solutions for problems that aren't textbook questions. On a positive note, I did finish the assignment with a much better understanding of string matching algorithms, both in theory and in actual implementation, which makes this whole experience feel like it was worth the struggle.

Student Name:Mohammad Ahmed Abdullah Ismail Student Number: 23050141007