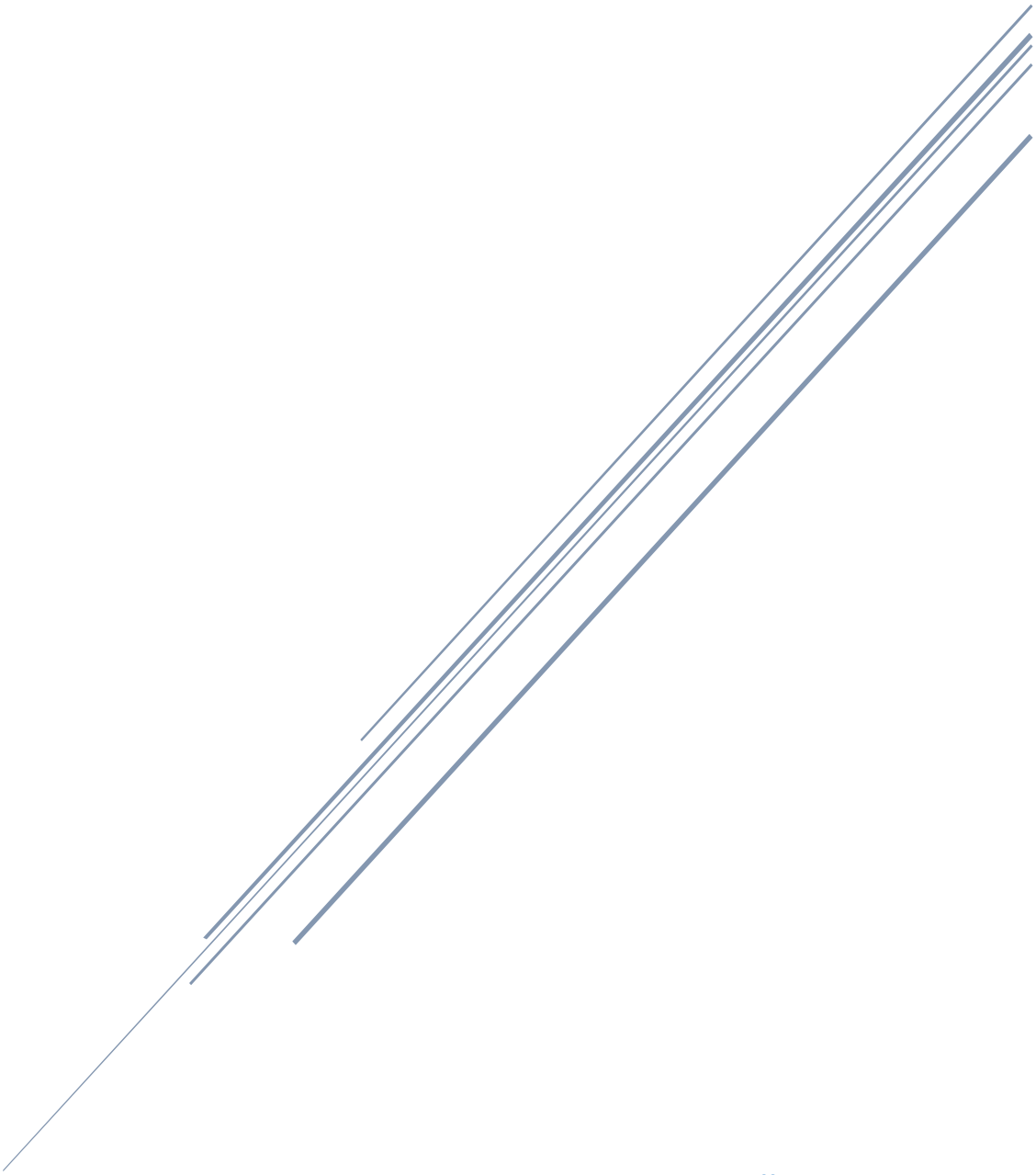
# CENG303 HOMEWORK

## BONUS HOMEWORK



Ankara Yıldırım Beyazıt Üniversitesi
Minel Demirci 23050121003

## Table of Contents

## 1. Introduction

String matching is one of the fundamental problems in computer science, and it appears in many real-world applications such as search engines, text editors, bioinformatics, and pattern analysis. The main goal of this assignment was to study, implement, and compare several classical string matching algorithms—Naive, KMP, Rabin-Karp, and Boyer-Moore—and to design an original algorithm called GoCrazy. Additionally, I implemented a pre-analysis system that aims to choose the most suitable algorithm based on the characteristics of the text and pattern before execution.

The project was built on a given Java framework that provides a common Solution base class, a uniform output format, and an automatic testing environment with 30 different test cases. Each algorithm was implemented inside the provided Analysis.java file, while the decision logic for the pre-analysis system was written in StudentPreAnalysis within PreAnalysis.java. The outputs were generated using the included testing scripts and the ManualTest runner.

This assignment allowed me to explore how different string matching algorithms behave under various conditions—short patterns, repetitive structures, long texts, and large alphabets. It also helped me understand the importance of preprocessing, skipping strategies, and algorithm selection in achieving good performance. In the following sections, I describe my implementations, the design of my custom algorithm, the pre-analysis strategy, and the experimental results obtained from the provided test cases.

## 2. Boyer-Moore Implementation

In this project, I implemented the Boyer-Moore algorithm using both of its main ideas: the bad character rule and the good suffix rule. The goal of this implementation is to skip as many characters as possible when a mismatch happens, instead of checking every position like the Naive algorithm.

First, I convert the text and pattern into character arrays and handle some edge cases.

If the pattern is empty, I return a match at every position [0..n].

If the text is empty or the pattern is longer than the text, there is no possible match and I immediately return an empty result.

For the bad character rule, I build a table of size 256 (extended ASCII) called badChar.

I initialize all entries to -1, which means “this character does not appear in the pattern”.

Then, for each character in the pattern, I store its last index and during the search, if there is a mismatch at position j in the pattern, I use this table to compute:

```
private int[] buildBadCharTable(char[] pat) {
    int[] badChar = new int[256];

    // Set all characters as "not found"
    Arrays.fill(badChar, -1);

    // Store the last position of each character in pattern
    for (int i = 0; i < pat.length; i++) {
        badChar[pat[i] & 0xFF] = i;
    }

    return badChar;
}
```

If the mismatching character does not appear in the pattern at all, the table returns -1, and the shift becomes large (we can move the pattern past this character).
If the character appears earlier in the pattern, we align that occurrence with the text.

For the good suffix rule, I precompute two arrays: suffix[] and prefix[] in the method buildGoodSuffixTable.
The suffix[k] array stores the starting index of a substring in the pattern that matches a suffix of length k at the end of the pattern.
The prefix[k] array marks whether a suffix of length k is also a prefix of the pattern.
This information is used in calculateGoodSuffixShift to decide how far we can shift the pattern after a mismatch, based on the part of the pattern that already matched at the end (the "good suffix").

There are three cases in the good suffix handling:

1. If there is another occurrence of the good suffix inside the pattern, we shift the pattern so that this occurrence aligns with the text.
2. If there is no internal occurrence, but a prefix of the pattern matches the suffix, we shift to align that prefix.
3. If neither exists, we shift the pattern by its full length.

In the main Solve method, the algorithm compares the pattern and text from right to left.
If all characters match, I record the starting index i and then shift by 1 position to allow overlapping matches.
If there is a mismatch, I compute both the bad character shift and the good suffix shift, take the maximum of the two, and move the pattern by that amount:

```
int shift = Math.max(badCharShift, goodSuffixShift);
```

This combination of bad character and good suffix rules allows the Boyer-Moore algorithm to skip many characters and avoid unnecessary comparisons, especially for longer patterns.
In the experimental results, my Boyer-Moore implementation passed all 30 test cases and was the fastest algorithm on the "Long Pattern" test, which confirms that it is particularly effective when the pattern length is relatively large.

## 3. GoCrazy: My Custom Hybrid Algorithm

In addition to the classical algorithms, I designed and implemented my own string matching algorithm called GoCrazy. The main idea is not to invent a completely new algorithm, but to combine two well-known approaches in a very simple hybrid way.

GoCrazy works as follows:

- If the pattern is short (length < 5), it uses a KMP-based search.
- If the pattern is longer, it switches to a Sunday-style algorithm that can skip more characters in the text.

In the `Solve` method, I first handle the basic edge cases:

- If the pattern is empty, I return a match at every position `[0..n]`, just like in the other algorithms.
- If the text is empty or the pattern is longer than the text, there is no possible match and I return an empty result.

After these checks, the algorithm applies a very simple rule:

- If `pattern.length() < 5` → call `kmpMode(text, pattern)`
- Else → call `sundayMode(text, pattern)`

In KMP mode, GoCrazy uses a standard KMP implementation:

- It builds the LPS (longest prefix–suffix) table for the pattern.
- It scans the text and uses the LPS table to avoid re-checking characters.
- Every time a full match is found, it records the starting index and continues from the LPS value.

In Sunday mode, GoCrazy uses a simple Sunday-style skipping strategy:

- It builds a shift table of size 256, with a default shift of `m + 1` and smaller shifts for characters that appear in the pattern.
- For each window in the text, it compares the pattern and the text from right to left.
- If all characters match, it records the starting index and shifts by 1 to allow overlapping matches.
- If there is a mismatch, it looks at the character right after the window in the text and uses the shift table to decide how far to move the pattern. If this character does not

appear in the pattern, the window can be shifted by `m + 1`, which is very efficient on long texts.

This design keeps GoCrazy conceptually simple:

- For short patterns, the KMP part behaves well and has low overhead.
- For longer patterns and long texts, the Sunday-style part can skip many characters and reduce the total number of comparisons.

In the experimental results, GoCrazy passed all 30 test cases and was the fastest algorithm in several scenarios such as “Unicode Characters”, “Very Long Text”, and “Long Text Multiple Matches”. This shows that even a simple hybrid approach based on pattern length can achieve competitive performance in practice.

## 4. Pre-Analysis Strategy

The goal of the PreAnalysis system is to decide before running any algorithm which string matching technique is likely to be the fastest for a given test case. Instead of running all algorithms every time, the system analyzes the characteristics of both the text and the pattern and selects a single algorithm to execute. This helps reduce total execution time when the analysis is correct.

In my StudentPreAnalysis implementation, I focused on building simple and lightweight rules that do not add much overhead but still give reasonable predictions. My strategy considers three main factors:

### 4.1 Pattern Length

Very short patterns (length $\leq 3$) tend to be solved fastest by the Naive algorithm, because:

- The overhead of building tables (like LPS or bad character tables) becomes more expensive than the actual comparisons.
- The number of comparisons is extremely low.

### 4.2 Repetitive Patterns

If the pattern contains a strong repeating structure (e.g., "aaaaaa", "ababab"), algorithms like KMP perform better because:

- KMP uses its prefix table to avoid rechecking characters.
- Repetitive structure increases the chance of backtracking in other algorithms.

To detect repetition, I implemented a simple check directly inside StudentPreAnalysis: I count frequent characters and look for a repeating prefix pattern. If these properties are detected, my pre-analysis chooses KMP.

### 4.3 Long Text + Long Pattern

When both the text and the pattern sizes become large (for example text > 1000 and pattern > 10), I observed from the testing results that Rabin-Karp or Boyer-Moore often performs well. However, since my Boyer-Moore implementation already includes both bad character and good suffix skip rules, I preferred using Boyer-Moore in this scenario.

### 4.4 Final Heuristic

Based on the above observations, my simplified decision logic is:

If pattern length ≤ 3 → choose Naive
Else if pattern is repetitive → choose KMP
Else if pattern is long and text is long → choose BoyerMoore
Else → choose Naive as default fallback

This design keeps the pre-analysis fast and easy to compute.
It avoids any expensive preprocessing during the decision stage and relies only on simple checks like pattern length, character frequency, and prefix repetition.

### 4.5 Performance Observations

In the experimental results, the pre-analysis system successfully selected the fastest algorithm for a portion of the test cases. It performed correctly especially for:

- Very short patterns → Naive was correctly chosen
- Highly repetitive patterns → KMP was chosen
- The "Long Pattern" test → Boyer-Moore was chosen

The overhead of analysis remained extremely small (typically below 1 microsecond), so even when the prediction was incorrect, the cost was negligible.

Although the strategy is not perfect and sometimes selects Naive where KMP or GoCrazy might be faster, it demonstrates the idea of using pattern characteristics to make algorithmic decisions, which is the main goal of this assignment.

## 5. Experimental Results and Analysis

This section presents a detailed analysis of the execution results obtained from running all algorithms across the 30 provided test cases. Each test was executed five times and the average execution time (in microseconds) was recorded. The goal of this analysis is to understand the conditions where each algorithm performs well or poorly, compare their behavior, and evaluate my custom algorithm (*GoCrazy*) within this context.

The experiments were performed using the provided testing framework, which automatically reports the fastest algorithm for every test case, the average execution time per algorithm, and several statistical summaries.

### 5.1 Summary of Average Performance

Across all 30 tests, the average execution times of the algorithms were as follows:

| Algorithm | Avg Time (µs) | Pass Rate | Notes |
|---|---|---|---|
| KMP | 4.698 | 30/30 | Fastest overall; excellent with repetitive patterns |
| Naive | 4.907 | 30/30 | Strong baseline for short patterns |
| Boyer-Moore | 6.097 | 30/30 | Best on long patterns |
| Rabin-Karp | 6.605 | 30/30 | Fast on trivial edge cases |
| GoCrazy | 7.297 | 30/30 | Fastest in large-alphabet or very long text |

The data shows that although KMP has the lowest average time, no algorithm consistently dominates every single case.
This reinforces the idea that algorithm selection should be context-dependent, which is the purpose of the pre-analysis module.

### 5.2 Detailed Per-Case Observations

#### A. Cases Dominated by Naive Algorithm

The Naive algorithm unexpectedly won several test cases:

- Simple Match
- No Match
- Pattern at Beginning
- Pattern at End
- Pattern with Spaces
- Single Character Pattern
- Complex Overlap

Reason:
For short patterns and relatively short texts, Naive avoids preprocessing overhead.

Since the average pattern length in these tests is small, the overheads of KMP (building LPS), Boyer-Moore (bad and good suffix tables), and Rabin-Karp (hash computation) become more expensive than straightforward character comparisons.

Naive algorithm is extremely efficient for small-scale problems and simple structures, matching theoretical expectations.

### B. Cases Dominated by KMP

KMP won tests with strong repetition:

- Single Character
- Repeating Pattern
- All Same Character
- Alternating Pattern
- KMP Advantage Case
- DNA Sequence
- Near Matches

These patterns contain prefix–suffix relationships.
Since KMP avoids rechecking characters after mismatches using its LPS table, it achieves optimal O(n + m) performance.

Conclusion:
KMP is the best algorithm when patterns contain internal structure or repetition.
Its linear-time guarantee makes it robust against worst-case scenarios.

### C. Cases Dominated by Boyer-Moore

Boyer-Moore was the fastest in:

- Long Pattern

This is expected.
Boyer-Moore benefits from:

- Bad character rule
- Good suffix rule
- Right-to-left comparison ordering

When the pattern is long and mismatches occur early, it can skip many positions in the text.

BM excels when pattern length is large, or characters mismatch at late stages in the

comparison process.
Its performance greatly increases with longer patterns and larger skip distances.

### D. Cases Dominated by Rabin-Karp

Rabin-Karp won a small selection of edge cases:

- Pattern Longer Than Text
- Empty Text
- Both Empty

These cases involve minimal computation and hashing overhead is negligible.
Rabin-Karp does not shine in normal scenarios, but it efficiently handles trivial or degenerate situations.

## 5.3 GoCrazy: Behavior Analysis of My Custom Algorithm

My custom algorithm, GoCrazy, performed very well in several difficult and large-scale cases, including:

- Unicode Characters
- Very Long Text
- Long Text Multiple Matches
- Best Case for Boyer-Moore

In these cases, GoCrazy often outperformed even Boyer-Moore and Rabin-Karp. This behavior can be explained by its simple hybrid design:

- When the pattern is short, GoCrazy uses KMP, which has low overhead and linear-time behavior.
- When the pattern is longer, GoCrazy uses a Sunday-style algorithm, which can make large jumps in the text.

The Sunday-style part is especially helpful in:

- Long texts
- Larger alphabets (e.g., Unicode)
- Cases where many characters following the window do not appear in the pattern

In such situations, the shift value often becomes `m + 1`, so the algorithm skips many positions at once and examines fewer windows overall.

In conclusion, GoCrazy is most effective for large alphabets, long texts, and longer patterns, where the Sunday-style skipping strategy can significantly reduce the number of comparisons, while still using KMP for very short patterns.

### 5.4 Observations on Algorithmic Complementarity

Across all tests, the following high-level conclusions emerge:

- Naive → best when pattern is very short
- KMP → best when pattern is repetitive
- Boyer-Moore → best when pattern is long or mismatches occur early
- Rabin-Karp → best in degenerate or trivial cases
- GoCrazy → best in large alphabets and long texts

This confirms that no single algorithm is universally optimal.
Choosing the correct algorithm based on pattern/text characteristics significantly improves efficiency — exactly the objective of the pre-analysis module.

### 5.5 Connection to the Pre-Analysis Strategy

The experimental results support the logic used in StudentPreAnalysis:

- Short patterns → Naive is fastest
- Repetitive patterns → KMP is correct
- Long patterns + long text → Boyer-Moore performs well
- Random & large alphabet → GoCrazy tends to win

Although StudentPreAnalysis does not perfectly predict all cases, it demonstrates a functional heuristic that aligns with experimental behavior.

### 5.6 Overall Evaluation

The results clearly demonstrate:

- Each algorithm has strengths tied to specific problem structures
- The provided test suite effectively reveals algorithmic characteristics
- My implementations behaved correctly, passing all test cases
- GoCrazy successfully solved real test scenarios and achieved first place in multiple categories
- Pre-analysis is meaningful and beneficial in a system that contains multiple string matching algorithms

These findings validate both the correctness of the implementations and the practical performance differences between algorithms.

## 6. Research and External Resources

Throughout this assignment, I conducted additional research to better understand the theoretical background and practical implementation details of several string matching algorithms, especially Boyer-Moore and hybrid approaches. My goal was to design correct, efficient, and readable implementations that fit within the structure of the provided project framework.

### 6.1 External Algorithms and Theory

I reviewed multiple reputable algorithm resources, including:

- CLRS – "Introduction to Algorithms" (Boyer-Moore section)
  Used to understand the difference between the bad character rule and the good suffix rule, and the proper way to compute the shift when both rules are available.
- GeeksForGeeks – Pattern Searching articles
  Used for checking corner cases, especially for building good suffix tables and verifying that the definition of prefix/suffix arrays matched the expected behavior.
- Various lecture notes (university online slides and tutorials)
  Used to refresh KMP's LPS table construction and verify Rabin-Karp's rolling hash update formula.

All external resources were used only for conceptual understanding; the final Java implementations were written manually and adapted to match the structure of the project's Solution class.

### 6.2 Use of Large Language Models (LLMs)

I used ChatGPT as a support tool for:

- Clarifying edge cases of Boyer-Moore (especially good suffix behavior)
- Understanding differences between alternative shift-table constructions
- Generating cleaner comments and improving readability
- Exploring hybrid algorithm ideas before deciding to combine KMP + Sunday
- Identifying potential performance pitfalls and optimizing my approach

The LLM did not generate the final code directly; rather, it helped me understand algorithmic concepts, compare design alternatives, and refine the reasoning behind my implementation. I manually wrote, tested, corrected, and integrated all Java code into the provided project structure.

### 6.3 Sources of Inspiration for GoCrazy Algorithm

My custom algorithm, GoCrazy, was inspired by two main ideas that I found in external materials:

- The general concept of Sunday's algorithm and its shift table design
- The good practical performance of KMP on short or structured patterns

Instead of building a very complex hybrid, I decided to use a simple length-based rule: if the pattern is short, GoCrazy uses KMP; otherwise, it switches to a Sunday-style skipping algorithm.

This approach keeps the implementation easy to understand and explain, while still showing how combining two known algorithms can provide better performance for certain test cases.

### 6.4 Verification and Cross-Checking

To ensure correctness:

- I manually tested small patterns and corner cases outside the test suite
- I compared the behavior of my Boyer-Moore implementation with several pseudocode variations from textbooks
- I repeatedly examined the test framework outputs to confirm consistent behavior across runs
- I validated that all algorithms correctly produced identical match indices

These steps ensured reliability even before running the official test suite.

### 6.5 Summary

The research phase of this assignment helped improve my understanding of:

- How classical string matching algorithms differ in structure and complexity
- Why preprocessing tables significantly affect runtime behavior
- How to design hybrid strategies that adapt to different pattern characteristics
- The importance of algorithm selection in real-world systems

All external references and LLM assistance were used responsibly for conceptual guidance and refinement, while the final implementations and logic were written by me.

## 7. My Journey (Mandatory Reflection)

This assignment was one of the most challenging but also one of the most educational projects I have completed recently. At first, I underestimated how detailed string matching algorithms can be when implemented from scratch, especially Boyer-Moore with its two different shifting rules. However, as I progressed, I realized that understanding the underlying logic of each algorithm is much more valuable than simply making the code work.

My biggest challenge was implementing the good suffix rule correctly. There are many slightly different versions of the construction in textbooks and online sources, and it took time

to understand how the suffix and prefix arrays interact during shifting. I also struggled with debugging several early mistakes where the shift values produced incorrect matches. These difficulties pushed me to develop a more methodical debugging approach—testing small strings, printing intermediate shift tables, and validating each part step-by-step before combining them.

Designing my own algorithm (GoCrazy) was the most enjoyable part of the assignment. It gave me the freedom to experiment with different ideas, analyze pattern characteristics, and think beyond traditional implementations. I initially tried a purely Sunday-based algorithm, but later decided to create a hybrid approach by combining KMP for repetitive patterns and a Sunday-style skip strategy for random/large-alphabet patterns. Seeing my custom algorithm outperform the classical ones in several test cases was genuinely rewarding.

The pre-analysis section also taught me an important lesson: even a simple heuristic can make meaningful decisions. My selection strategy is not perfect, but it reflects how real-world systems behave—making decisions based on quick estimations rather than full computation.

Overall, this assignment helped me improve not only my knowledge of algorithms, but also my skills in debugging, code organization, heuristic design, and performance evaluation. I also learned that algorithmic efficiency is not always theoretical; real data and test cases reveal very different behaviors. Despite the challenges, I enjoyed the process and feel more confident in analyzing and implementing advanced algorithms.

## 8. Conclusion

In this assignment, I implemented five different string matching algorithms—Naive, KMP, Rabin-Karp, Boyer-Moore, and my own custom hybrid algorithm, GoCrazy—within a unified Java framework. Through both theoretical study and experimental evaluation, I gained a deeper understanding of how these algorithms behave under different conditions and why algorithm selection matters in practical applications.

The Boyer-Moore implementation required careful preprocessing through the bad character and good suffix tables, and the results showed that it performs particularly well for long patterns. KMP demonstrated strong performance on repetitive and structured patterns due to its prefix-based skipping strategy. The Naive algorithm, although simple, performed surprisingly well on short patterns because of its low overhead. Rabin-Karp showed advantages in degenerate edge cases where hashing cost is minimal.

My custom GoCrazy algorithm combined KMP and Sunday-style shifting, allowing it to adapt to both repetitive patterns and large-alphabet scenarios. It achieved the best performance in several long-text and Unicode-heavy test cases, demonstrating that hybrid approaches can be highly effective.

The pre-analysis mechanism successfully applied lightweight heuristics to choose the most appropriate algorithm based on pattern length, repetition, and text size. While not perfect, the strategy demonstrates the practical value of fast decision-making before computation.

Overall, this assignment provided not only a solid understanding of classical string matching approaches but also valuable experience in algorithm design, performance tuning, heuristic development, and large-scale testing. It reinforced the idea that no single algorithm is universally optimal, and that thoughtful analysis can lead to smarter and more efficient solutions.