Report & Journey

1.1 Boyer–Moore Implementation

As part of this assignment, I developed a complete implementation of the Boyer–Moore string matching algorithm. The implementation includes both the bad character rule and the good suffix rule.

The algorithm first handles the following edge cases:

- If the pattern is empty, all positions in the range 0..n are returned as results.
- If the pattern is longer than the text, the function immediately returns an empty result since no match is possible.
- If the pattern length is 1, the algorithm checks the text using a simple linear scan.

In the general case, the algorithm constructs two preprocessing tables:

- **Bad Character Table (buildBadCharacterMap)**
  It stores the last seen index for each character in the pattern.
  During a mismatch, the algorithm shifts the pattern based on the last occurrence of the mismatching character in the pattern; if the character does not exist in the pattern, a larger jump is performed.
- **Good Suffix Shift Table (buildGoodSuffixShift)**
  Using a standard border array, the algorithm computes how far the pattern can be shifted based on the previously matched suffix when a mismatch occurs.
  This table covers both the “matched suffix” and the “prefix of the pattern” cases.

In the search loop, the algorithm scans the pattern from right to left for each alignment:

- If a full match is found (j < 0), the index is recorded and the pattern is shifted by goodSuffix[0].
- If a mismatch occurs:

```
// Mismatch: Calculate the biggest possible jump.
int badCharShift = j - badChar.getOrDefault(t[i + j], -1);
int goodSuffixShift = goodSuffix[j + 1];
int shift = Math.max(badCharShift, goodSuffixShift);
i += (shift > 0) ? shift : 1;
```

and the pattern is shifted by the larger of these two values.
This approach directly follows the principle in the theoretical Boyer–Moore algorithm: "always perform the maximum safe shift."

As a result, our implementation is theoretically:

- **Preprocessing:** O(σ + m)
- **Search:** O(n) on average, and approximately O(n/m) in terms of character comparisons, meaning sublinear average running time.

It is a full Boyer–Moore implementation that provides significantly improved practical speed, especially for large alphabets and long patterns.

1.2 GoCrazy "Gated Bidirectional Search" Algorithm

In the creative algorithm section of the assignment, I developed a heuristic algorithm that I named **GoCrazy**. I defined this approach as a **"gated bidirectional search"**.

The core idea is to use the first and last characters of the pattern as filters (gates). In this way, many alignments can be eliminated at O(1) cost before performing a full match check.

Steps of the algorithm:

- For each window (alignment i):
    - Does the first character of the pattern match the first character of the window?
    - Does the last character of the pattern match the last character of the window?
    - If both "gates" are passed, the inner part of the pattern (1..m–2) is checked linearly.
    - The window is shifted 1 position to the right.

```
while (i <= n - m) {
    // 1. Check last and first characters first (the two "gates")
    if (text.charAt(i + m - 1) == lastChar && text.charAt(i) == firstChar) {

        // 2. If gates pass, check the rest of the pattern from inside
        boolean match = true;
        for (int j = 1; j < m - 1; j++) {
            if (text.charAt(i + j) != pattern.charAt(j)) {
                match = false;
                break;
```

The worst-case time complexity is still O(n·m) theoretically.
However, in practice – especially for large alphabets or random texts – the probability that the edge characters do not match is very high, so most windows are eliminated before performing the inner check. This yields a significant performance gain.

The experimental results confirm this:

- Across all 30 tests, GoCrazy has the lowest average running time (~2.773 µs).
- It is faster than the average execution times of Naive, KMP, Boyer–Moore, and Rabin–Karp.

```
SUMMARY STATISTICS:
================================================================================
Naive          : 30 passed, 0 failed | Avg: 3,505 µs, Min: 0,099 µs, Max: 20,500 µs
KMP            : 30 passed, 0 failed | Avg: 4,999 µs, Min: 0,241 µs, Max: 28,066 µs
GoCrazy        : 30 passed, 0 failed | Avg: 2,773 µs, Min: 0,133 µs, Max: 11,450 µs
BoyerMoore     : 30 passed, 0 failed | Avg: 5,872 µs, Min: 0,408 µs, Max: 34,474 µs
RabinKarp      : 30 passed, 0 failed | Avg: 5,180 µs, Min: 0,141 µs, Max: 28,142 µs
================================================================================
```

- In the detailed results table, GoCrazy appears as the “Winner” in many tests: Simple Match, No Match, Long Text Multiple Matches, Entire Text Match, DNA Sequence, Best Case for Boyer–Moore, etc..

DETAILED TEST RESULTS – Execution Time Comparison (Average of 5 runs)

Test Case	Naive (µs)	KMP (µs)	GoCrazy (µs)	BoyerMoore (µs)	RabinKarp (µs)	Winner
Simple Match	2,916	3,074	2,566	5,050	3,366	GoCrazy
No Match	0,716	1,275	0,508	3,625	1,058	GoCrazy
Single Character	6,299	6,066	7,650	6,283	7,692	KMP
Pattern at End	2,116	1,624	1,050	4,008	1,449	GoCrazy
Pattern at Beginning	1,083	1,858	1,008	4,166	1,525	GoCrazy
Overlapping Patterns	3,158	2,858	2,716	4,425	3,275	GoCrazy
Long Text Multiple Matches	4,258	5,300	2,625	9,958	5,658	GoCrazy
Pattern Longer Than Text	0,208	0,617	0,133	0,508	0,141	GoCrazy
Entire Text Match	0,800	2,441	0,849	3,991	1,350	Naive
Repeating Pattern	2,533	1,799	1,558	3,433	2,183	GoCrazy
Case Sensitive	1,550	2,150	1,241	4,250	2,025	GoCrazy
Numbers and Special Characters	1,975	2,483	1,825	5,525	2,758	GoCrazy
Unicode Characters	5,933	9,775	3,391	5,400	7,049	GoCrazy
Very Long Text	20,500	28,066	11,450	34,474	28,142	GoCrazy
Pattern with Spaces	2,425	2,666	1,566	3,058	2,725	GoCrazy
All Same Character	8,542	6,075	8,008	7,133	10,250	KMP
Alternating Pattern	12,300	21,325	10,658	20,333	25,566	GoCrazy
Long Pattern	5,391	7,858	4,358	5,441	8,416	GoCrazy
Pattern at Boundaries	0,841	2,150	0,966	2,258	2,100	Naive
Near Matches	0,474	1,616	0,650	1,983	1,450	Naive
Empty Pattern	0,541	0,791	0,691	2,550	0,708	Naive
Empty Text	0,099	0,325	0,133	0,408	0,175	Naive
Both Empty	0,133	0,241	0,216	0,483	0,224	Naive
Single Character Pattern	2,600	5,858	4,200	3,283	6,025	Naive
Complex Overlap	0,974	5,033	1,574	2,183	2,458	Naive
DNA Sequence	3,166	5,999	4,808	4,150	8,041	Naive
Palindrome Pattern	1,199	4,017	1,825	3,124	4,108	Naive
Worst Case for Naive	6,766	6,066	2,291	11,441	5,183	GoCrazy
Best Case for Boyer-Moore	1,158	4,750	1,916	4,458	4,625	Naive
KMP Advantage Case	4,483	5,800	0,766	8,791	5,683	GoCrazy

For this reason, despite its simplicity, GoCrazy has exhibited a much stronger general-purpose algorithmic performance than expected.

1.3 Pre-Analysis Strategy – “Smart Routing”

In the final part of the assignment, I developed a pre-analysis module that analyzes the text and the pattern to select the most suitable algorithm in advance. The name of my strategy is **“Intelligent Routing.”**

The strategy consists of a set of hierarchical rules.

Rule 0 – Basic States

```
// --- RULE 0: Trivial Cases (O(1) checks) ---
if (m == 0 || n == 0)
    return "Naive";
if (m > n)
    return "RabinKarp";
```

These have O(1) cost and handle cases such as empty pattern/text or impossible matches.

Rule 1 – Small Patterns

```
// --- RULE 1: Micro Patterns (O(1) check) ---
if (m <= 4)
    return "GoCrazy";
```

In this length range, the preprocessing cost of algorithms such as KMP and Boyer–Moore outweighs their benefits, so either GoCrazy or Naive — which have the lowest initialization cost — could be selected. Since our GoCrazy algorithm performs better on average, we chose to select GoCrazy.

Rule 2 – İ Content-Based Custom Patterns

```
// --- RULE 2: Specialized Content Heuristics (O(k) checks) ---
// These are strong signals for specific algorithms.
if (isHighlyRepetitive(pattern))
    return "KMP";
if (containsNonAscii(pattern))
    return "RabinKarp";
```

- **isHighlyRepetitive:** If the pattern consists of only 1 or 2 distinct characters within the first 64 characters (e.g., "AAAAAA" or "ABABAB"), KMP is the most effective algorithm for such patterns.
- **containsNonAscii:** If it contains Unicode characters, the alphabet is large, and hash-based methods (Rabin–Karp) behave more stably.

Rule 3 – General Purpose Heuristics

```
int distinctChars = countDistinctChars(pattern);
if (m > 20 && distinctChars > 15) {
    return "BoyerMoore";
}

// Heuristic 3b: High Pattern/Text Ratio -> GoCrazy
// If the pattern is very long relative to the text, the number of possible
// alignments is small. The setup cost of Boyer-Moore may not be worth it.
// GoCrazy has lower setup cost and is a good choice here.
if (n > 0 && m > n / 2) { // Check n > 0 to avoid division by zero
    return "GoCrazy";
```

- Uzun ve çok çeşitli karakterlere sahip pattern → Boyer–Moore'un optimum senaryosu.
- Pattern metnin yarısından uzun ise hizalama sayısı azdır → düşük setup maliyetli GoCrazy daha ekonomik hale gelir.

A long pattern with many diverse characters → the optimal scenario for Boyer–Moore.

If the pattern is longer than half of the text, the number of alignments is small → GoCrazy, which has a low setup cost, becomes more economical.

Rule 4 – Default Algorithm

return "GoCrazy";

If none of the rules trigger, GoCrazy is selected as it reliably handles all scenarios.
(It is also possible to choose Naive as the default, but since GoCrazy performs better than Naive in most cases and conditions, we chose GoCrazy.)

Overhead – Accuracy Balance

This strategy is designed to balance two important objectives:

- **Pre-analysis must be very fast**
  → It includes only fixed-length prefix scans and a few O(1) checks.
- **It must select the correct algorithm as often as possible**, because if the pre-analysis overhead becomes large, the analysis itself loses its purpose.
  → It contains rules simple enough to correctly identify scenarios where KMP, BM, or GoCrazy are strong.

All three content functions (isHighlyRepetitive, countDistinctChars, containsNonAscii) scan at most 64 characters, so the analysis time is almost constant, which helps achieve a short and efficient PreAnalysis step.

1.4 Experimental Evaluation of Pre-Analysis Results

The "Detailed Test Results – Execution Time Comparison" table shows the raw execution time of each algorithm. In this table, GoCrazy has by far the lowest average runtime.

In the "Pre-Analysis Performance Comparison" table:

- The total of **PreAnalysis time + the time of the selected algorithm**,
- The difference compared to the actual time of each algorithm

are computed.

Green values → PreAnalysis provided a time gain.
Red values → PreAnalysis was slower.

Overall:

- In many tests, the correct algorithm was selected and time was saved.
- Since the PreAnalysis overhead is low, the total cost is advantageous in most cases.
- In some tests (Pattern at End, Alternating Pattern, Long Pattern, KMP Advantage Case), incorrect selection or choosing algorithms with high setup cost such as Boyer–Moore caused only a very small extra cost.

This indicates that more complex analyses might be required to make even more accurate decisions; however, such complexity would slow down PreAnalysis and could harm overall performance.
Therefore, the chosen strategy successfully achieves the intended balance.

“Additionally, these outputs and results may vary depending on the current performance of the computer. To maintain stability, instead of expecting green output for all cases, we preferred to make a more general performance evaluation. For this reason, even if the result appears red in some cases, having a small overhead time was more important to us than being green, because we were trying to determine whether the most optimal result exists.”

Test Case	Choice	PreA+Choice (µs)	vs GoCrazy	vs Naive	vs KMP	vs BoyerMoore	vs RabinKarp
Simple Match	GoCrazy	1,93	N/A	+1,07 µs	+0,33 µs	+0,59 µs	+1,45 µs
No Match	GoCrazy	0,33	N/A	+0,10 µs	−0,88 µs	−0,62 µs	+0,07 µs
Single Character	GoCrazy	1,61	N/A	+0,87 µs	+0,06 µs	+0,63 µs	+0,80 µs
Pattern at End	Naive	4,12	+3,91 µs	N/A	+3,02 µs	+2,98 µs	+3,82 µs
Pattern at Beginning	Naive	1,30	+1,05 µs	N/A	+0,22 µs	+0,14 µs	+0,99 µs
Overlapping Patterns	GoCrazy	0,52	N/A	+0,09 µs	−1,67 µs	−0,60 µs	−0,21 µs
Long Text Multiple Matches	GoCrazy	0,78	N/A	−0,31 µs	−3,83 µs	−0,88 µs	−0,34 µs
Pattern Longer Than Text	RabinKarp	0,29	+0,20 µs	+0,20 µs	−0,28 µs	+0,04 µs	N/A
Entire Text Match	GoCrazy	1,57	N/A	+1,24 µs	+0,68 µs	−4,56 µs	+1,29 µs
Repeating Pattern	GoCrazy	1,14	N/A	+0,38 µs	−0,10 µs	−0,14 µs	+0,56 µs
Case Sensitive	GoCrazy	4,39	N/A	+3,92 µs	+2,85 µs	+3,19 µs	+3,88 µs
Numbers and Special Characters	Naive	1,75	+1,27 µs	N/A	−2,85 µs	+0,27 µs	+1,05 µs
Unicode Characters	GoCrazy	0,51	N/A	+0,02 µs	−1,44 µs	−1,32 µs	−0,58 µs
Very Long Text	GoCrazy	9,21	N/A	+2,04 µs	−15,13 µs	+2,98 µs	+2,16 µs
Pattern with Spaces	Naive	2,64	+1,55 µs	N/A	−1,70 µs	+0,92 µs	+1,93 µs
All Same Character	KMP	3,00	−0,09 µs	−0,53 µs	N/A	−0,06 µs	−0,67 µs
Alternating Pattern	GoCrazy	4,17	N/A	−0,33 µs	+0,41 µs	+0,86 µs	−0,94 µs
Long Pattern	BoyerMoore	6,11	+4,74 µs	+4,13 µs	+2,73 µs	N/A	+3,79 µs
Pattern at Boundaries	GoCrazy	0,65	N/A	+0,12 µs	−0,17 µs	−0,24 µs	+0,12 µs
Near Matches	GoCrazy	0,37	N/A	−0,09 µs	−0,30 µs	−0,49 µs	−0,03 µs
Empty Pattern	Naive	0,42	+0,11 µs	N/A	+0,11 µs	−0,12 µs	+0,15 µs
Empty Text	Naive	0,21	+0,12 µs	N/A	−0,04 µs	+0,13 µs	+0,15 µs
Both Empty	Naive	0,24	+0,02 µs	N/A	+0,11 µs	+0,11 µs	+0,12 µs
Single Character Pattern	GoCrazy	1,61	N/A	−0,16 µs	−0,56 µs	+0,23 µs	−0,38 µs
Complex Overlap	KMP	1,23	+0,69 µs	+0,57 µs	N/A	−0,00 µs	+0,26 µs
DNA Sequence	GoCrazy	1,88	N/A	−0,96 µs	−0,34 µs	+0,19 µs	−0,75 µs
Palindrome Pattern	Naive	3,72	+0,68 µs	N/A	+0,06 µs	+2,62 µs	+2,70 µs
Worst Case for Naive	KMP	3,08	+2,37 µs	−3,08 µs	N/A	+0,43 µs	+1,84 µs
Best Case for Boyer-Moore	GoCrazy	0,84	N/A	−0,24 µs	−0,91 µs	−0,15 µs	−0,31 µs
KMP Advantage Case	Naive	6,04	+5,33 µs	N/A	+3,36 µs	+3,10 µs	+4,77 µs

1.5 Research Section

“How did we use internet resources and LLM (Large Language Model) tools in this assignment?”

During the design, comparison of the algorithms used in this assignment, and the construction of the pre-analysis strategy, we used both classical sources and modern LLM tools together. Our goal was to produce a solution that is both theoretically correct and practically effective by leveraging different sources of information. Additionally, thanks to the instructor’s feedback, we better understood our strategy and what the assignment aimed to teach us.

My research process progressed as follows:

1. Theoretical Background Research

First, to more clearly understand how algorithms such as Boyer–Moore, KMP, and Rabin–Karp operate and under which conditions they exhibit optimal performance, I used the following sources:

- GeeksForGeeks – Pattern Searching section
- Wikipedia – Boyer–Moore String Search
- University lecture slides (String Matching notes)

These resources provided a solid foundation especially for:

- How the bad character and good suffix tables are constructed
- The logic behind KMP’s failure function
- The advantages and disadvantages of Rabin–Karp’s hash computations

2. LLM-Based Research (Using ChatGPT / Gemini)

In this assignment, we used LLMs primarily to validate our design decisions and to understand the performance analysis of complex scenarios.

Our use of LLMs included the following aspects:

a) Determining the Heuristics Needed for the Pre-Analysis Strategy

For example:

- “Which pattern characteristics make KMP more advantageous?”
- “In which scenarios does Boyer–Moore achieve the largest jumps?”
- “Why do hash-based algorithms behave more stably when the pattern contains Unicode?”

We asked such questions to the LLM to obtain different perspectives.

Thanks to these answers, we designed routing rules such as:

- repetitive pattern → KMP
- long + large alphabet → Boyer–Moore
- very short pattern → GoCrazy
- Unicode → Rabin–Karp

b) Designing the GoCrazy Algorithm

I initially designed GoCrazy based on the idea of a “two-gate” control, but the questions I asked the LLM included:

- “Does endpoint filtering provide practical speed gains in pattern matching?”
- “Even if the worst-case is O(n·m), how can it still run fast in practice?”
- “What is the statistical effect of edge-character matching in large-alphabet scenarios?”

These answers validated our design and allowed us to implement the algorithm confidently.

c) Optimizing PreAnalysis Overhead

From the LLM, we sought guidance on:

- the overhead of using a HashSet,
- choosing the prefix sampling length (the suggestion of 64 characters),
- the effect of O(1) or O(k) checks.

This allowed us to make the analysis module fast and constant-cost.

3. Experiments – Output Analysis – Improvements

Each time I ran my code, I shared the resulting green/red table with the LLM and asked:

- "Why am I losing in these tests?"
- "If I change this rule, what side effects will it cause?"
- "Why does Alternating Pattern produce a red result?"

Through this, I received improvement suggestions.

As a result of this refinement loop:

- unnecessary Boyer–Moore selections were removed,
- my repetitive-pattern analysis was corrected,
- Unicode detection was prioritized,
- the default selection strategy was refined.

General Conclusion

LLMs were highly useful not for writing the code itself, but as a guide during decision-making and optimization.

1.6 Journey Section

This assignment process was highly instructive for us and at the same time practically challenging. Since it was structured like a real job scenario, we did not remain only at the theoretical level; instead, we questioned how these theoretical concepts should guide real-world design decisions. I previously knew string matching algorithms theoretically, but comparing their performance within a real testing framework, understanding their strengths/weaknesses, and designing a pre-analysis system based on these insights was an entirely different experience.

During this process, I particularly learned the following:

1. The question of the "right algorithm" is meaningless on its own

The tests showed that:

- KMP is extremely fast in some scenarios but unnecessarily costly in others. Although we assume in theory that extra memory usage does not cause time loss, in the real world it introduces overhead.
- Boyer–Moore is theoretically very powerful, but in practice it is highly sensitive to the structure of the pattern and alphabet.
- Rabin–Karp works well with Unicode but is unnecessary for short patterns.
- Even Naive becomes a special-case winner for very short patterns.
- GoCrazy, with its simple logic, won many tests.

In other words, there is no single “best” algorithm; the right decision completely depends on the structure of the input.

Experiencing this firsthand was extremely valuable for us.

2. Designing PreAnalysis is harder than it seems

At first, I thought: “We analyze the pattern and choose the best algorithm,” but in practice:

- if the analysis cost is high, the benefit of choosing correctly disappears,
- if I add too many rules, it becomes overfitted,
- if I add too few rules, selection quality decreases.

In the end, we learned to design a structure that is lightweight yet effective.

3. Writing our own algorithm (GoCrazy) was the most enjoyable part

Developing an original approach, ensuring that it works correctly in edge cases, and seeing that it performs well in tests motivated us greatly. It was especially satisfying to see that the fast-rejection idea worked in practice.

4. Challenges We Encountered

- The good suffix implementation of Boyer–Moore was quite complicated at first.
- While implementing Boyer–Moore, in our first attempt, we allocated the Bad Character table with a size of Integer.MaxValue. Later, we realized that this caused a huge overhead, and by examining the average and test case results of

Boyer–Moore, we saw that it was taking much longer than expected, and it was not becoming the winner even in the cases where Boyer–Moore should perform exceptionally well.

Based on this, we revised the Bad Character table by using a HashMap, and we observed that Boyer–Moore performed faster in the scenarios where it was expected to work well, and its average execution time decreased as expected.

- It took time to interpret the test results correctly.
- While optimizing some red tests, we had to be careful not to drift toward an overly complex PreAnalysis.

With each iteration, we progressed by understanding the behavior of the algorithms more deeply.

5. Overall Evaluation

For us, this assignment was not only about writing code, but also about developing our abilities in:

- decision-making,
- algorithm selection,
- performance analysis,
- heuristic reasoning,
- and understanding how to make decisions not only theoretically but also in practice.

Enes Geldi 21050111035

Okan Rıdvan Gür 21050111022