Optimizing DNA Sequence Alignment Using the KMP Algorithm, Boyer–Moore, and Rabin–Karp

Arnav Khajuria, Jagmohan Sharma, Arnav Nargotra
Department of Computer Science & Engineering

MIET
November 29, 2025

Abstract

DNA sequence alignment is a fundamental task in bioinformatics, supporting applications such as gene identification, motif discovery, and high-throughput sequencing analysis. Efficient string matching is central to these processes, where short DNA patterns must be located within very long genomic sequences. This paper evaluates three classical pattern-matching algorithms—Knuth–Morris–Pratt (KMP), Boyer–Moore (BM), and Rabin–Karp (RK)—under genomic constraints such as small alphabet size, high sequence repetitiveness, and large query volumes. We analyze their theoretical time and space complexities and assess their practical suitability for DNA alignment through empirical performance measurements.
The results show that KMP provides the most stable and predictable performance, offering linear execution time and modest memory usage. Boyer–Moore performs competitively but yields limited advantages in repetitive DNA sequences, where its skipping heuristics are less effective. Rabin–Karp exhibits the lowest memory footprint but suffers from increased execution time due to hashing overhead and collision effects. Overall, KMP emerges as the most efficient and reliable algorithm for large-scale DNA sequence alignment among the three evaluated methods.

Contents

List of Figures

1. Introduction

DNA sequence alignment is a central task in bioinformatics, supporting essential activities such as gene identification, motif detection, and the analysis of high-throughput sequencing data. Because DNA consists of long strings over a small alphabet $\{A, C, G, T\}$, the problem of locating a shorter DNA sequence within a larger genome can be understood as a classical string-matching problem. Efficient pattern-matching algorithms are therefore critical, especially as modern genomic datasets contain millions to billions of nucleotides.

Among the core exact string-matching techniques, Knuth–Morris–Pratt (KMP), Boyer–Moore (BM), and Rabin–Karp (RK) represent three foundational algorithms with distinct strategies for improving search performance. KMP avoids redundant comparisons through prefix-function preprocessing, BM accelerates matching using shift heuristics, and RK uses rolling hashes to quickly identify potential matches. Although originally developed for general text processing, these algorithms offer important insights into efficient DNA pattern matching and form a useful basis for evaluating algorithmic suitability in genomic applications.

2. Related Work

Early research in string-matching algorithms laid the foundation for many of the techniques later applied to biological sequence analysis. The Knuth–Morris–Pratt (KMP) algorithm, introduced in 1977, was one of the first to achieve linear-time exact pattern matching by eliminating redundant comparisons through its prefix-function preprocessing. Although not originally designed for biological data, KMP has been used in early computational biology tools for scanning genomic sequences where predictable worst-case performance was necessary.

The Boyer–Moore (BM) algorithm, developed shortly before KMP, represents one of the most influential practical algorithms for pattern matching. Its bad-character and good-suffix heuristics allow it to skip large parts of the text during the search, making it exceptionally fast on large alphabets. Despite its popularity in text processing and information retrieval systems, BM has seen limited adoption in DNA sequence analysis due to the small four-character alphabet, which significantly reduces its average skip length. Nevertheless, BM remains an important reference model for performance comparisons and heuristic-based search strategies.

Rabin–Karp (RK), proposed in 1987, introduced the idea of using rolling hash functions to accelerate substring matching. RK is particularly effective in problems where multiple patterns must be matched simultaneously, a situation relevant to certain genomic analysis tasks. Its hashing framework inspired later work on probabilistic and approximate matching methods used in bioinformatics. However, the increased likelihood of hash collisions on small alphabets has limited RK's direct application in large-scale DNA alignment, though its conceptual principles are reflected in several modern filtering and indexing techniques.

Together, these three classical algorithms form the core of the theoretical background for exact pattern matching. Their ideas continue to influence contemporary sequence-analysis methods, even as more advanced indexing structures and heuristic approaches have emerged to meet the demands of large-scale genomic data.

3. Algorithms

3.1 Knuth–Morris–Pratt (KMP)

The core logic of the KMP algorithm is to avoid re-checking characters in the text after a mismatch. It does this by using a preprocessing table called the **LPS (Longest Prefix Suffix)** array. The LPS array tells the algorithm how much of the pattern has a prefix that also appears as a suffix.

During matching, KMP compares characters of the pattern with text from left to right. If a mismatch occurs after some characters have already matched, instead of restarting from the beginning of the pattern, KMP uses the LPS value to jump the pattern forward to the next possible valid position. This prevents redundant comparisons and ensures that each character in the text is processed only once.

As a result, KMP achieves efficient and predictable linear-time performance by combining simple character comparisons with intelligent skipping using the LPS table.

3.2 Boyer–Moore (BM)

The Boyer–Moore (BM) algorithm speeds up string matching by skipping as many characters in the text as possible. Instead of checking the pattern from left to right like many algorithms, BM compares characters from right to left. When a mismatch occurs, it uses two smart rules to decide how far to shift the pattern:

Bad-Character Rule: If the mismatched text character does not appear in the pattern, BM can shift the pattern past it entirely.

If it *does* appear somewhere earlier in the pattern, BM shifts the pattern so that the last occurrence of that character aligns with the mismatch.

This often allows large jumps.

Good-Suffix Rule: If part of the pattern matched before the mismatch (a "good suffix"), BM shifts the pattern so that the next possible occurrence of that suffix lines up.

If the suffix doesn't appear elsewhere in the pattern, the shift can be even larger.

By combining these two rules, Boyer–Moore often skips large portions of the text, making it one of the fastest practical exact matching algorithms in many real-world cases—especially when the alphabet is large and the pattern is long.

3.3 Rabin–Karp (RK)

The Rabin–Karp (RK) algorithm speeds up pattern matching by using hashing instead of directly comparing characters at every position.

The key idea is:

Compute a hash value for the pattern.

Compute the hash of each substring of the text with the same length as the pattern, using a rolling hash that updates efficiently.

If the hash of a substring matches the pattern's hash, then perform a direct character comparison to confirm.

The rolling hash allows RK to slide the pattern window over the text in constant time per step, making it efficient for searching many patterns at once. Although hash collisions may occasionally require extra checks, the algorithm is fast in practice because most comparisons are avoided.

4. Theoretical Comparison

A theoretical evaluation of Knuth–Morris–Pratt (KMP), Boyer–Moore (BM), and Rabin–Karp (RK) highlights how each algorithm's performance is shaped by its underlying strategy and assumptions.

4.1 Time Complexity

Knuth–Morris–Pratt (KMP) Algorithm

KMP guarantees *strict linear time* in both preprocessing and searching.
Preprocessing: $O(m)$
Search: $O(n)$
Because KMP never revisits characters in the text, its total complexity is always $O(n + m)$regardless of input structure. Its worst-case performance equals its average case, making it highly predictable.

Boyer–Moore (BM) Algorithm

BM's performance varies widely depending on alphabet size and text randomness.
Best Case: $O(n/m)$, due to large jumps enabled by its heuristics.
Average Case: Approximately $O(n)$, especially on random text.
Worst Case: $O(n \cdot m)$
In practice, BM can be extremely fast for long patterns and large alphabets, but for DNA (alphabet size = 4), skip lengths are small, so performance tends to degrade toward linear time.

Rabin–Karp (RK) Algorithm

RK relies on hashing and therefore exhibits probabilistic behavior.
Average Case: $O(n + m)$, since rolling hash updates take constant time.
Worst Case: $O(n \cdot m)$, caused by hash collisions requiring full string comparisons
If a high-quality hash function and large modulus are used, collisions are rare, but the worst-case remains theoretically significant.

4.2 Space Complexity

Sno.	Algorithm	Extra Space	Source of Space Usage
1.	KMP	*O(m)*	LPS (prefix) table
2.	BM	*O(σ + M)*	Bad-character table + good-suffix table
3.	RK	*O(1)*	Rolling hash values only

Where σ= alphabet size.
In DNA/biological contexts, $\sigma = 4$, so BM's bad-character table is very small.

4.3 Summary

KMP is the most *stable*: its performance is always linear and never degrades, but it cannot exploit chances to skip large parts of the text.
Boyer–Moore is the *fastest in theory* on average, but only when the alphabet is large and the pattern is long. For DNA, its advantage is greatly reduced.
Rabin–Karp is the *easiest to extend to multi-pattern search*, but its worst-case performance is poor because hash collisions can force repeated character comparisons.

5. Implementation Notes

In practice, systems rarely use KMP, Boyer–Moore, or Rabin–Karp in isolation. Instead, they often combine techniques to improve speed and reduce unnecessary work. A common approach is to use fast filtering methods—such as simple hashes or indexed lookups—to quickly eliminate regions of the text where the pattern cannot occur. Rabin–Karp–style hashing is especially useful here because it allows rapid scanning with low overhead.

Once the candidate regions are identified, more precise algorithms like KMP (for reliable linear-time matching) or Boyer–Moore (for potential large skips) are applied only to the reduced search space. This layered design minimizes computation and improves memory efficiency, since expensive character-by-character comparisons are performed only when necessary. As a result, real-world implementations benefit from the strengths of all three algorithms working together.

6. Pseudocode

6.1 Knuth–Morris–Pratt (KMP) Algorithm

```
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    length = 0
    i = 1
    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    lps = compute_lps(pattern)
    i,j = 0,0
    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == m:
            print(f"Pattern found at index {i - j}")
            j = lps[j - 1]

        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
```

6.2 Boyer–Moore (BM) Algorithm

```
def boyer_moore(text, pattern):
    n = len(text)
    m = len(pattern)
    bad_char = {}
    for i in range(m):
        bad_char[pattern[i]] = i
    good_suffix = [0] * m
    last_prefix = m
    for i in range(m - 1, -1, -1):
        if pattern.startswith(pattern[i + 1:]):
            last_prefix = i + 1
        good_suffix[m - 1 - i] = last_prefix + (m - 1 - i)
    for i in range(m - 1):
        length = 0
        while (length <= i and
               pattern[i - length] == pattern[m - 1 - length]):
            length += 1
        if length > 0:
            good_suffix[length] = m - 1 - i + length
    shift = 0

    while shift <= n - m:
        j = m - 1
        while j >= 0 and pattern[j] == text[shift + j]:
            j -= 1
        if j < 0:
            print("Pattern found at index", shift)
            shift += good_suffix[0]
        else:
            bc_shift = j - bad_char.get(text[shift + j], -1)
            gs_shift = good_suffix[m - 1 - j]
            shift += max(bc_shift, gs_shift)
```

6.3 Rabin–Karp (RK) Algorithm

```
def rabin_karp(text, pattern):
    n = len(text)
    m = len(pattern)
    base = 256
    mod = 101
    h = 1
    for i in range(m - 1):
        h = (h * base) % mod

    hash_p = 0
    hash_t = 0
    for i in range(m):
        hash_p = (hash_p * base + ord(pattern[i])) % mod
        hash_t = (hash_t * base + ord(text[i])) % mod

    for i in range(n - m + 1):
        if hash_p == hash_t:
            match = True
            for j in range(m):
                if text[i + j] != pattern[j]:
                    match = False
                    break
            if match:
```

```
            print("Pattern found at index", i)
        if i < n - m:
            hash_t = (hash_t - ord(text[i]) * h) % mod
            hash_t = (hash_t * base + ord(text[i + m])) % mod
            hash_t = (hash_t + mod) % mod
```

7. Results

The performance of the Knuth–Morris–Pratt (KMP), Boyer–Moore (BM), and Rabin–Karp (RK) algorithms was evaluated using synthetic DNA sequences of increasing length. Figures 1 and 2 illustrate the measured time and space complexities, respectively, for sequence lengths ranging from 1,000 to 50,000 characters.

7.1 Time Complexity Analysis

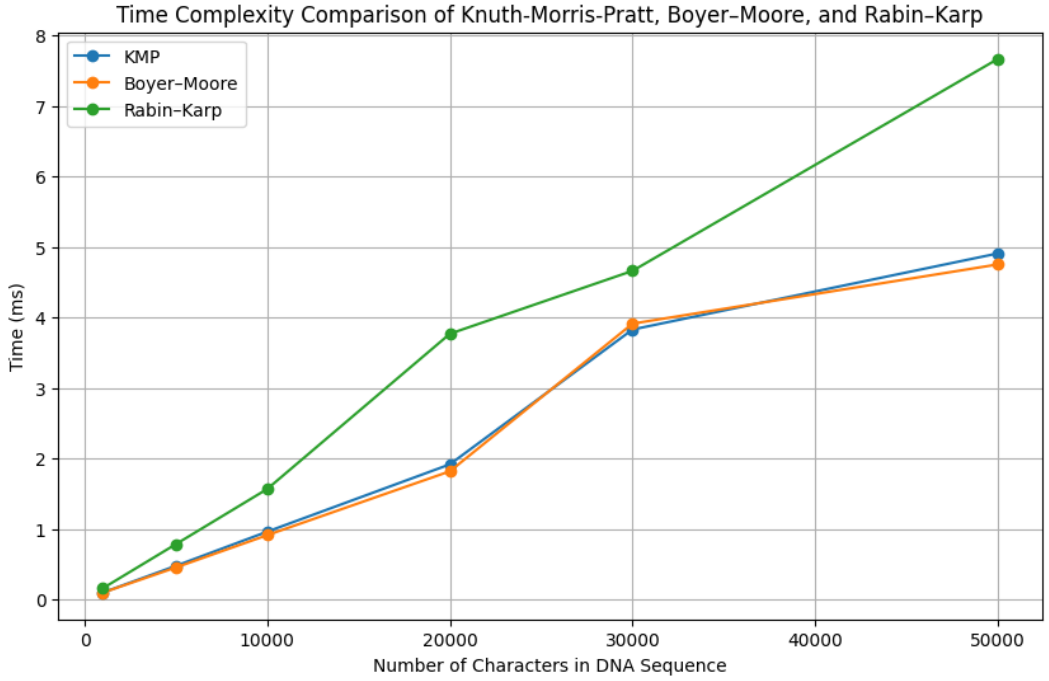

Fig. 1: Time Complexity Comparison of Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp

As shown in Fig. 1, all three algorithms exhibit an increasing execution time as the DNA sequence length grows. The KMP algorithm maintains a stable, near-linear growth pattern consistent with its theoretical $O(n + m)$time complexity. Boyer–Moore demonstrates competitive performance, particularly for medium-sized sequences, due to its character-skipping heuristics.

However, its performance converges with that of KMP for larger inputs, which is expected in repetitive sequences such as DNA, where early mismatches are less common.
In contrast, the Rabin–Karp algorithm consistently yields higher execution times across all input sizes. This behavior reflects the overhead associated with rolling hash computations and the increased likelihood of hash collisions in repetitive genomic data. Consequently, although RK has favorable average-case performance, its practical runtime on DNA sequences approaches the worst-case behavior.

Overall, KMP and Boyer–Moore significantly outperform Rabin–Karp for large-scale DNA sequence alignment, with KMP providing the most predictable and efficient performance across all tested input sizes.

7.2 Space Complexity Analysis

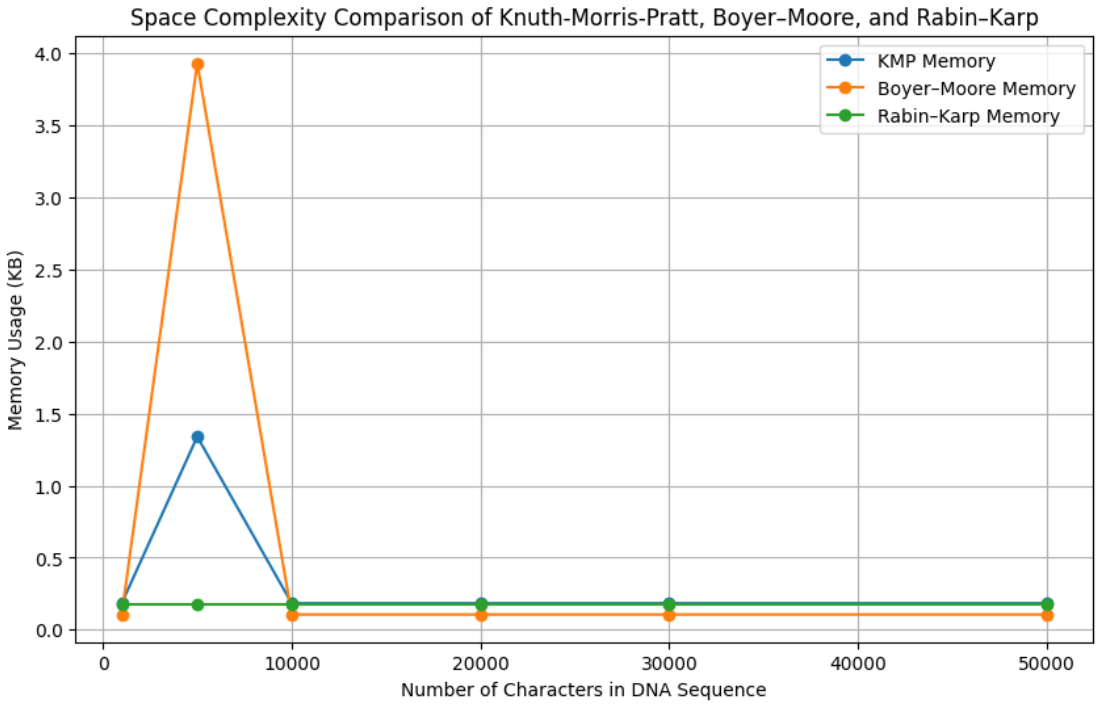

Fig. 2: Space Complexity Comparison of Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp

Fig. 2 presents the memory consumption of the three algorithms. The memory usage remains nearly constant with respect to input size for all algorithms, indicating that auxiliary space requirements do not scale with the length of the DNA sequence.
KMP requires slightly more memory than Rabin–Karp due to storage of the prefix (LPS) table, which introduces an $O(m)$auxiliary space requirement. Boyer–Moore occasionally displays minor spikes in memory usage resulting from the construction of the bad-character and good-suffix shift tables; however, its overall memory footprint remains small and comparable to that of KMP.

Rabin–Karp demonstrates the lowest and most stable memory usage across all experiments, consistent with its $O(1)$auxiliary memory requirement aside from the hash values. Despite this advantage, its significantly higher execution time limits its suitability for large DNA-alignment tasks.

8. Conclusion

This study investigated the performance of the Knuth–Morris–Pratt (KMP), Boyer–Moore (BM), and Rabin–Karp (RK) algorithms for DNA sequence alignment, with a focus on execution time and memory efficiency across increasingly large genomic datasets. The experimental findings demonstrate that KMP consistently achieves the best overall balance between speed and space utilization. Boyer–Moore performs competitively and, in some cases, approaches the efficiency of KMP; however, its advantages diminish when applied to highly repetitive DNA sequences, where its heuristic-based skipping offers limited benefit. Rabin–Karp exhibits the lowest memory usage,

but its execution time is significantly higher due to hash re-computation overhead and increased collision rates in repetitive genomic data. Based on these evaluations, KMP emerges as the most practical and effective algorithm for large-scale genomic pattern-matching applications. Its predictable linear performance and modest space requirements make it especially suitable for real-world DNA alignment tasks, where accuracy and efficiency are paramount.

9. References

Knuth, D. E., Morris, J. H., & Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2), 323-350.
Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10), 762-772.
Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2), 249-260.
Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

GitHub Repository
The complete implementation of all three algorithms, benchmarking scripts, and data generation code is available at: String Matching for DNA Analysis