Knuth–Morris–Pratt (KMP) Algorithm for DNA Pattern Matching

STARK DNA Pattern Matching Project

November 2025

Abstract

This report presents a complete overview of the Knuth–Morris–Pratt (KMP) exact string matching algorithm and its application to DNA sequence analysis. We cover the LPS (Longest Proper Prefix which is also Suffix) preprocessing, the linear-time search procedure, correctness, complexity, and empirical performance on both synthetic and real genomic datasets. Results and implementation follow the structure used for Boyer–Moore in this repository, ensuring consistent documentation across algorithms.

1 Introduction

String matching is central to bioinformatics tasks such as motif search, primer validation, and genome annotation. Given a text T of length n and a pattern P of length m, KMP finds all indices i such that $T[i..i+m-1] = P[0..m-1]$.

DNA sequences over the alphabet $\Sigma = \{A, C, G, T\}$ make KMP attractive due to its predictable $O(n)$ worst-case runtime and small memory overhead.

1.1 Problem Statement

Input: Text T (DNA), pattern P (DNA). Output: All start positions of P in T.

2 Algorithm Description

KMP avoids redundant comparisons by preprocessing P into an LPS array. When a mismatch occurs after matching a prefix of P, the LPS value indicates where to resume in P without re-checking characters in T.

2.1 LPS (Failure Function)

For each index i in P, LPS[i] is the length of the longest proper prefix of $P[0..i]$ that is also a suffix of $P[0..i]$. LPS can be computed in $O(m)$ time.

2.2 Search Procedure

We scan T left-to-right while maintaining an index j into P. On match, advance both indices; on mismatch after $j > 0$ matches, fall back to $j = \text{LPS}[j-1]$. If $j = m$, a match is reported and j falls back to $\text{LPS}[m-1]$ to allow overlapping matches.

3 Complexity Analysis

- Preprocessing (LPS): $O(m)$ time, $O(m)$ space.
- Search: $O(n)$ time in the worst case, $O(1)$ extra space beyond LPS and loop variables.
- Total: $O(n+m)$ time, $O(m)$ space.

4 Proof of Correctness

We provide a formal proof of correctness for the KMP algorithm by establishing a loop invariant for the search phase and proving the time complexity bound using amortized analysis.

4.1 Loop Invariant and Correctness

We define the invariant for the 'while' loop in the search procedure where i is the text index and j is the pattern index. Let π denote the failure function table (LPS).

Theorem 1 (Correctness Invariant)**.** *At the start of each iteration of the 'while' loop, $P[0 \dots j-1]$ is the longest prefix of P that is a suffix of $T[0 \dots i-1]$.*

Proof. We proceed by induction on the number of loop iterations.

- **Initialization:** Initially $i = 0, j = 0$. $P[0 \cdots -1]$ and $T[0 \cdots -1]$ are empty strings ϵ. ϵ is the longest prefix of P that is a suffix of ϵ. The invariant holds.
- **Maintenance:** Assume the invariant holds at the start of an iteration.
  1. **Case 1 ($T[i] == P[j]$):** We increment both i and j. Since $P[0 \dots j-1]$ is a suffix of $T[0 \dots i-1]$ (inductive hypothesis) and $P[j] = T[i]$, it follows that $P[0 \dots j]$ is a suffix of $T[0 \dots i]$. The length of the matching prefix becomes $j+1$, maintaining the invariant.
  2. **Case 2 ($T[i] \neq P[j]$ and $j > 0$):** We update $j \leftarrow \pi[j-1]$. By the definition of the failure function, the new $P[0 \dots \pi[j-1]-1]$ is the second longest prefix of P that matches the suffix of $T[0 \dots i-1]$. We do not increment i, so we re-test against $T[i]$ in the next iteration. This recursive reduction continues until a match is found or $j = 0$.
  3. **Case 3 ($T[i] \neq P[j]$ and $j == 0$):** No prefix of P matches $T[0 \dots i]$. We increment i. The longest matching prefix is empty (length 0). The invariant holds.

- **Termination:** The algorithm terminates when $i = n$. If $j = m$ at any point, a match is recorded, implying $P[0 \dots m-1]$ is a suffix of $T[0 \dots i-1]$.

□

4.2 Time Complexity Proof via Potential Function

To rigorously prove the $O(n)$ time complexity, we use an amortized analysis.

Proof. Define a potential function $\Phi = 2i - j$.

- When $T[i] = P[j]$: $i \to i+1, j \to j+1$. $\Delta\Phi = 2(i+1) - (j+1) - (2i - j) = 1$.
- When $T[i] \neq P[j], j > 0$: $i \to i, j \to \pi[j-1]$. Since $\pi[j-1] < j$, j decreases. $\Delta\Phi = 2i - \pi[j-1] - (2i - j) = j - \pi[j-1] \geq 1$.
- When $T[i] \neq P[j], j = 0$: $i \to i+1, j \to 0$. $\Delta\Phi = 2(i+1) - 0 - (2i - 0) = 2$.

In every step, Φ increases by at least 1. Since initially $\Phi = 0$ and finally $\Phi \approx 2n$ (as $j < m \leq n$), the total number of operations is bounded by $2n$. Thus, the time complexity is $O(n)$. □

5 Pseudocode

5.1 LPS Computation

Algorithm 1 ComputeLPS(P)

```
m ← |P|, LPS[0] ← 0, len ← 0
for i ← 1 to m − 1 do
    while len > 0 and P[i] ≠ P[len] do
        len ← LPS[len − 1]
    end while
    if P[i] = P[len] then
        len ← len + 1, LPS[i] ← len
    else
        LPS[i] ← 0
    end if
end for
return LPS
```

5.2 Search

Algorithm 2 KMP-Search(T, P)

$n \leftarrow |T|$, $m \leftarrow |P|$, LPS $\leftarrow$ ComputeLPS(P)
$i \leftarrow 0$, $j \leftarrow 0$, matches $\leftarrow [\,]$
while $i < n$ **do**
    if $T[i] = P[j]$ **then**
        $i \leftarrow i + 1$, $j \leftarrow j + 1$
        if $j = m$ **then**
            matches.append($i - j$); $j \leftarrow$ LPS$[m - 1]$
        end if
    else if $j > 0$ **then**
        $j \leftarrow$ LPS$[j - 1]$
    else
        $i \leftarrow i + 1$
    end if
end while
return matches

6 Edge Cases

- $m = 0$: invalid (we reject empty patterns).
- $m > n$: no matches.
- Highly repetitive patterns (e.g., "AAAA"): still linear due to LPS fallback.
- Overlapping matches: handled by resetting $j \leftarrow$ LPS$[m - 1]$ after a match.

7 Implementation Overview

We implement KMP in Python (see `kmp.py`) with an object-oriented wrapper exposing: `search`, `search_first`, `count_matches`, and `search_multiple_patterns`. The module normalizes input to uppercase and builds LPS once per pattern instance.

8 Experimental Setup

We evaluate on synthetic DNA (uniform random over $\{A, C, G, T\}$) and real genomes from NCBI (directory: `DnA_dataset/ncbi_dataset/data`). Benchmarks include:

1. Pattern length impact at fixed n.
2. Text length scaling at fixed m.
3. Multiple pattern search over the same text.
4. Per-genome runs across all available FASTA files (see notebook cell "Benchmark KMP across all genomes").

Outputs are stored in JSON (e.g., `KMP/benchmark_results.json`, `KMP/kmp_nb_results.json`).

9 Results

9.1 Synthetic Sequences

Across $n \in [10^4, 10^6]$ with moderate m (20–100), KMP shows near-constant throughput with total time growing linearly in n, consistent with $O(n)$.

9.2 Real Genomes

For bacterial-size genomes ($\tilde{4}$–6 Mbp), KMP processes sequences in linear time with speeds on the order of tens of Mbp/s on a typical laptop CPU. Aggregated per-genome plots and summaries are generated by the notebook.

Metric	Min	Median	Mean	Max
Speed (Mbp/s)	–	–	–	–
Time (ms)	–	–	–	–

Table 1: Summary placeholder; see notebook for concrete numbers.

10 Discussion

KMP provides predictable linear-time behavior independent of pattern content, in contrast to Boyer–Moore which benefits from larger shifts on average. For DNA alphabets, KMP is robust and memory-light, making it a great baseline for exact matching.

11 Conclusion and Future Work

KMP achieves $O(n + m)$ performance for exact DNA pattern matching with minimal memory overhead. Future work includes: multi-pattern automata (Aho–Corasick), approximate matching, SIMD acceleration, and parallelization.

Reproducibility

- Code: `KMP/kmp.py`, benchmarks in `KMP/benchmark.py`, notebook `KMP/kmp_nb.ipynb`.
- Datasets: `DnA_dataset/ncbi_dataset/data`.
- Environment: see `KMP/requirements.txt`.

References

[1] Knuth, D. E.; Morris, J. H.; Pratt, V. R. (1977). "Fast pattern matching in strings." SIAM Journal on Computing 6(2): 323–350.

[2] Gusfield, D. (1997). Algorithms on Strings, Trees, and Sequences. Cambridge University Press.

[3] GeeksforGeeks. KMP Algorithm for Pattern Searching. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

[4] NCBI Datasets. https://www.ncbi.nlm.nih.gov/datasets