Pattern Matching Algorithms: **Knuth–Morris–Pratt (KMP)** and **Boyer–Moore (BM)**

Presented by:

Raag Arya (A125002)
Manav Mantry (A125009)
Priyanshu Mishra (A125016)

Department of CSE, IIIT Bhubaneswar

November 11, 2025

Outline

The String-Searching Problem: Formal Definition

Problem Definition

Given:

- A **Text** T of length n: $T = T[0 \dots n-1]$
- A **Pattern** P of length m: $P = P[0 \dots m-1]$

Objective: Find all occurrences of P in T, i.e., all shifts s such that $0 \leq s \leq n-m$ and

$$T[s+i] = P[i] \quad \text{for all } i \in [0 \dots m-1].$$

Real-World Applications

- **Text Editors:** Quick search functionality (Ctrl+F)
- **Bioinformatics:** DNA/protein sequence alignment
- **Network Security:** Signature-based intrusion detection
- **Plagiarism Detection:** Matching copied text in documents

The Naïve (Brute-Force) Approach

Motivation: Why We Need Better Algorithms

Algorithm Idea

Try every possible shift s from 0 to $n - m$:

- Compare $P[0 \ldots m-1]$ with $T[s \ldots s+m-1]$ character by character.
- If a mismatch occurs, stop and move to the next shift $s + 1$.

Worst-Case Example

Text T: `AAAAAAAAAAAAAAAAB` ($n = 17$)
Pattern P: `AAAAAB` ($m = 6$)

- $s = 0$: Compare 6 chars $\rightarrow$ mismatch at last char
- $s = 1$: Compare 6 chars $\rightarrow$ mismatch at last char
- ...
- $s = 11$: Compare 6 chars $\rightarrow$ mismatch at last char

Time Complexity of Naïve Matching

Worst-Case Complexity

For each of the $n - m + 1$ shifts, up to m character comparisons may occur.

$$\textbf{Time Complexity} = O((n - m + 1) \cdot m) = \boxed{O(nm)}$$

Interpretation

Extremely inefficient when:

- Pattern is repetitive
- Text is repetitive

$\rightarrow$ This motivates smarter algorithms like KMP and Boyer–Moore.

KMP: The Core Idea

- The Naïve algorithm is slow because it re-checks characters already matched.
- After a mismatch, we already know a prefix of the pattern matched.
- **KMP uses this knowledge to skip ahead intelligently.**
- KMP never moves the Text pointer i backward.
- **Example:**
  Text: `ABCABD...`
  Pattern: `ABCABC`
- At mismatch ('D' vs 'C') — we had matched `"ABCAB"`.
- Naïve would shift by 1 $\rightarrow$ re-check redundant characters.
- KMP knows prefix `"AB"` = suffix of matched segment `"ABCAB"`.
- So KMP shifts pattern directly to align that prefix and continues immediately.

KMP Preprocessing: The LPS Table

LPS (Longest Proper Prefix = Proper Suffix)

We precompute an array π (also called *lps*) of size m.
$\pi[q]$ = length of the longest proper prefix of $P[0..q]$ that is also a suffix of $P[0..q]$.

Example: $P = \texttt{ABABACA}$

q	0	1	2	3	4	5	6
$P[q]$	A	B	A	B	A	C	A
$\pi[q]$	0	0	1	2	3	0	1

$\Rightarrow$ prefix "AB" repeats $\rightarrow$ no re-comparisons later.

KMP Search Algorithm: Core Operational Logic

- Maintain 2 indices: i over Text T, j over Pattern P.
- **Match:** $T[i] == P[j]$
  - advance both $\rightarrow i{+}{+}, j{+}{+}$
- **Mismatch:** $T[i] \neq P[j]$
  - if $j > 0 \rightarrow$ use LPS $\rightarrow$ jump pattern back: $j \leftarrow \pi[j-1]$
  - if $j = 0 \rightarrow$ no prefix help $\rightarrow$ just move $i{+}{+}$
- **Full match:** $j = m$
  - occurrence at index $(i - j)$
  - continue search by setting $j = \pi[j-1]$

KMP Complexity: LPS Preprocessing

Why $O(m)$?

- We scan pattern once left$\rightarrow$right.
- If mismatch occurs $\rightarrow$ we do not restart; we jump using π.
- Total pointer movement on $q + k \leq 2m$.

$k \leftarrow 0$, $\pi[0] \leftarrow 0$
for $q \leftarrow 1$ to $m - 1$ **do**
    while $k > 0$ and $P[k] \neq P[q]$ **do**
        $k \leftarrow \pi[k-1]$
    end while
    if $P[k] == P[q]$ **then**
        $k \leftarrow k + 1$
    end if
    $\pi[q] \leftarrow k$
end for

KMP Search Complexity (Idea)

Search runs in $O(n)$

- Text pointer i only moves forward $\rightarrow$ max n increments.
- Pattern pointer j increases on matches, decreases on mismatch via π.
- Every decrement of j corresponds to some earlier increment.
- Total increments $+$ decrements of $j \leq 2n$.

Final Result

$$\text{Preprocessing} = O(m), \qquad \text{Search} = O(n)$$

$$\boxed{O(n+m)}$$

Boyer-Moore: The Core Idea

- KMP scans **left-to-right**.
- **Boyer–Moore (BM)** scans **right-to-left**.
- On a mismatch, BM uses two heuristics to skip ahead by large steps:
  1. **Bad Character Rule**
  2. **Good Suffix Rule**
- **Example:**

  Text: `...THIS_IS_A_SIMPLE_EXAMPLE...`

  Pattern: `.EXAMPLE...`

BM Heuristic 1: Bad Character Rule

Preprocessing & Example

Preprocessing — The Bad Character Table

Build a table (or hash map) λ for all characters in the alphabet.

$\lambda[c]$ = index of the **last occurrence** of character c in $P[0 \dots m-1]$.
If c does not appear in P, then $\lambda[c] = -1$.

Example: $P = \texttt{EGAMPLE}$ $(m = 7)$

Character c	$\lambda[c]$
E	6
G	1
A	2
M	3
P	4
L	5

BM Heuristic 1: Bad Character Rule

Shift Rule

Shift Calculation

On a mismatch at pattern index j where the text character is $T[i] = c$, compute:

$$\text{Shift} \;=\; \max\left(1,\; j - \lambda[c]\right).$$

If $\lambda[c] = -1$ then the pattern moves completely past the text character (shift $\geq m$ in practice).

Worked Example

Suppose $P = \texttt{EGAMPLE}$ ($m = 7$). If we mismatch at pattern index $j = 5$ with text char $c = \texttt{X}$, then

$$\lambda[\texttt{X}] = -1 \quad \Rightarrow \quad \text{Shift} = \max(1, 5 - (-1)) = 6.$$

So we shift the pattern 6 positions (i.e. effectively past the bad character).

BM Heuristic 2: Good Suffix Rule

Motivation & Intuition

The Problem with Only Bad Character Rule

What if the Bad Character rule gives a tiny (or zero) shift?

Example:

Text: `...ABCDE...`
Pattern: `...ZYCDE...`

- `'E'` and `'D'` match.
- Mismatch at `'C'` vs `'Y'` ($j = 2$).
- Bad Character gives $\lambda[\texttt{'C'}] = 2 \Rightarrow$ shift $= 2 - 2 = 0$ (useless).

Hence, we need another heuristic to improve shifting efficiency.

BM Heuristic 2: Good Suffix Rule

The Idea Behind It

Idea — The Good Suffix Rule (Intuition)

Let $t = P[j+1 \dots m-1]$ be the matched suffix (for example, "DE").

The goal is to use this matched suffix t to decide a smarter shift:

- Look for the **rightmost other occurrence** of t in P.
- Ensure that occurrence is **not preceded by the same mismatching character**.
- If it exists $\rightarrow$ shift to align them.
- Otherwise $\rightarrow$ shift by m (move pattern past mismatch).

Example

$P = \texttt{ABCDABD}$, mismatch at $j = 3$.
Matched suffix $t = \texttt{"ABD"}$.
Rightmost reoccurrence of t starts at position 4 $\rightarrow$ shift to align.

BM Heuristic 2: Good Suffix Rule

Combined Shift Rule

At Runtime: Combine Both Heuristics

When mismatch occurs at position j with text character c:

$$\text{BadCharShift} = \max(1, j - \lambda[c])$$

$$\text{GoodSuffixShift} = \beta[j]$$

Then, the actual shift used by Boyer–Moore is:

$$\boxed{\text{Shift} = \max(1, \text{BadCharShift}, \text{GoodSuffixShift})}$$

Intuition

Boyer–Moore always takes the largest safe shift from both heuristics, allowing it to skip large sections of the text efficiently.

BM Search Algorithm

1. Initialize shift $s = 0$.
2. While $s \leq n - m$:
   1. Set $j = m - 1$.
   2. **Compare right-to-left:**
      While $j \geq 0$ and $P[j] == T[s + j]$, do $j \leftarrow j - 1$.
   3. **If $j < 0$:**
      - Match found at position s.
      - Shift by Good Suffix rule (or by 1 if unavailable).
   4. **Else (Mismatch at $P[j]$):**
      - Compute Bad Character shift: $\text{shift}_{bc} = j - \lambda[T[s + j]]$
      - Compute Good Suffix shift: shift_{gs} (from precomputed table)
      - Update: $s \leftarrow s + \max(1, \text{shift}_{bc}, \text{shift}_{gs})$

BM Complexity Analysis

Preprocessing: $O(m+k)$

- **Bad Character Table:** $O(k)$ to initialize, $O(m)$ to fill.
- **Good Suffix Table:** Computed in $O(m)$.
- Total preprocessing time: $O(m+k)$.

Best Case: $O(n/m)$

BM can skip large portions of text.

- Example: $T = \texttt{THIS\_IS\_A\_TEXT}$, $P = \texttt{ZYXWVU}$.
- Each mismatch at pattern end $\rightarrow$ shift by m.
- $\Rightarrow$ One comparison per shift, $\approx n/m$ comparisons total.

Worst Case: $O(n)$

- Without Good Suffix $\rightarrow$ $O(nm)$ (e.g., repetitive text).

KMP vs Boyer-Moore

Feature	KMP	Boyer-Moore
Match Direction	Left$\rightarrow$Right	Right$\rightarrow$Left
Best Case	$O(n)$	$O(n/m)$
Worst Case	$O(n)$	$O(n)$
Typical Use	Small alphabets	Large alphabets

Key Takeaway

Boyer–Moore is typically faster in practice for natural language text due to large alphabet size and its sub-linear $O(n/m)$ average performance.

Conclusion

- Naïve search runs in $O(nm)$ — too slow for large data.
- Both **KMP** and **BM** achieve efficient **linear-time** behavior via preprocessing.
- **KMP Insight:** Uses prefix-suffix overlap (LPS table) to avoid redundant comparisons.
- **BM Insight:** Scans right-to-left and uses Bad Character & Good Suffix rules to make large jumps.
- **Algorithm choice:**
  - KMP — best for small alphabets or streaming text.
  - BM — best for large alphabets (e.g., natural language, code).

References

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 3rd Edition, 2009.
- D. E. Knuth, J. H. Morris, and V. R. Pratt, *Fast Pattern Matching in Strings*, SIAM Journal on Computing, 1977.
- R. S. Boyer and J. S. Moore, *A Fast String Searching Algorithm*, Communications of the ACM, Vol. 20, No. 10, 1977.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

Thank You!