

# String Matching Algorithms

Kris Mannino



CSC 3331 Analysis of Algorithms

---

Spring 2024

---

Project #1

---

Dr. Jinghua Zhang



**WINSTON-SALEM**  
STATE UNIVERSITY

## Introduction

In computer science, string matching (or string searching) is often referred to as the needle in the haystack problem. Given a large body of text ( $n$ ) or *haystack*, the task is to find a specific pattern of characters ( $m$ ) within that text, the *needle*. Many algorithms have been developed to solve this problem. In this paper I will give an overview of real world applications and implementations of a brute force (naive) strategy as well as three of the most common algorithms: Knuth-Morris-Pratt (KMP), Rabin-Karp, and Boyer-Moore:

**Brute Force Strategy:** The simplest string matching algorithm, using the basic knowledge that involves comparing the pattern with every substring of the text.

**Knuth-Morris-Pratt (KMP) Algorithm:** A more efficient algorithm that avoids unnecessary character comparisons by utilizing a prefix function to skip ahead in the text.

**Boyer-Moore Algorithm:** An approach by scanning the text from right to left, skipping comparisons based on a “bad character” rule and a “good suffix” rule (Fun, 2024)

Algorithm	Preprocessing time	Matching time	Space
Naïve algorithm	none	$\Theta(n+m)$ in average,	none
		$\Theta(mn)$	
Knuth–Morris–Pratt	$\Theta(m)$	$\Theta(n)$	$\Theta(m)$
Boyer–Moore	$\Theta(m + k)$	$\Omega(n/m)$ at best,	$\Theta(k)$
		$O(mn)$ at worst	

Fig1. Time Complexity (Wikipedia, 2022)

## Background

Relatively unnoticed string matching algorithms permeate almost all aspects of data driven computing. Their usage and implementations are widespread and varied.

- **Spell-checkers** use the concept of a “trie”, or a predefined set of patterns (Kumar Soni et al.) that continuously matches possibilities of what the correct string should be, based on characters already entered based on “fuzzy” or incorrect string matching.
- **Spam filters** use string matching to identify and discard spam emails by looking for “suspected signature patterns” (Kumar Soni et al., 2014) in the content and classifying the email as malicious or not.
- **Plagiarism** detectors use string tokens to compare similarities between a body of text to determine similarities.
- **Search engines and databases** have presorted large datasets that combine the inherited categorization and tokenization of data along with the previously mentioned techniques to return results quickly.
- **Natural Language processing (NLP)** utilizes “fuzzy”, or approximate string matching, to match strings based on inaccurate or additional input characters basing a match on and “Edit distance”. This Levenshtein Distance (Levenshtein, 1966) measures how far apart two words and returns a match.
- **DNA Sequencing** must be mentioned as a straightforward “needle” problem in an incomprehensibly large “haystack”. Gene sequencing and DNA analysis are so intertwined with string matching as to be indistinguishable.

## Algorithms

## I. Naïve string matching (Brute Force)

The naïve algorithm compares each substring of text (T) to text of pattern (P).

The *sliding window* of characters moves one character from left-to-right looking for a match.

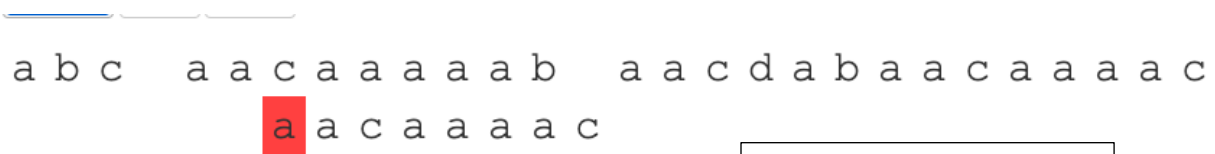
**Time Complexity:  $O(n^2)$**

**Auxiliary Space:  $O(1)$**

**Best Case:  $O(n)$ :** When the pattern is at the beginning of the text, it will make comparisons equal to the length of the pattern (**n**).

**Worst Case:  $O(n^2)$ :** The pattern appears at the end of the text, or the entire pattern is tested against for each character position in the text,  **$O((n-m+1)*m)$**  comparisons.

(GeeksforGeeks, 2011)



a b c a a c a a a a b a a c d a b a a c a a a c  
a c a a a a c

(Galles & Lucet, 2015)

Animation in Slide

**Pseudocode:**

```
for i := 0 to n-1 {  
  for j := 0 to m-1 {  
    if P[j] <> T[i+j] then break  
  }  
  if j = m then return i  
}
```

(Cornell University, 2002)

## II. Knuth-Morris-Pratt (KMP)

The KMP algorithm considers information from previous comparisons and does not compare these unnecessary characters. With preprocessing of the pattern, it can determine the number of characters to skip in the pattern whenever a match is not

found. This allows the algorithm to compare the two strings without “backing up”, making the time to check for matches “proportional to the sum of the lengths of the strings.” (Knuth et al., 1977)

**Time Complexity:  $O(n+m)$**

**Auxiliary Space:  $O(m)$**

KMP works because of its reliance on preprocessing and the presence of a “longest matching prefix that is also a suffix”. It uses the values derived for each index in the pattern to determine an integer array that will tell us how many characters can be skipped. “The idea is to build a graph representation that will provide information as to the amount of “slide” that will be necessary when a mismatch occurs.” (Miller & Ranum, 2011)

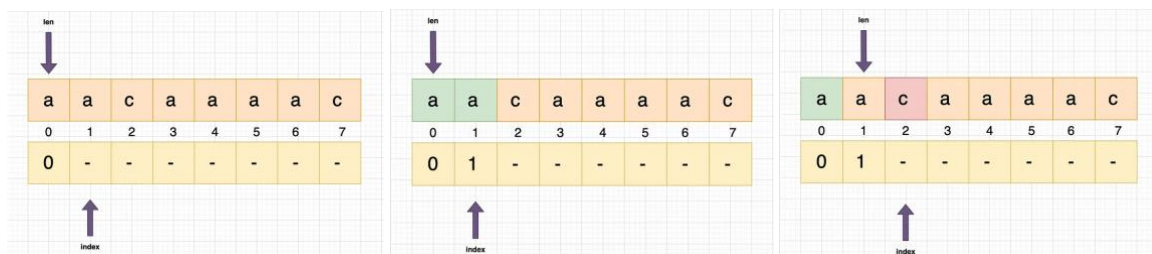
**Pseudocode: Longest Prefix that is a Suffix (lps)**

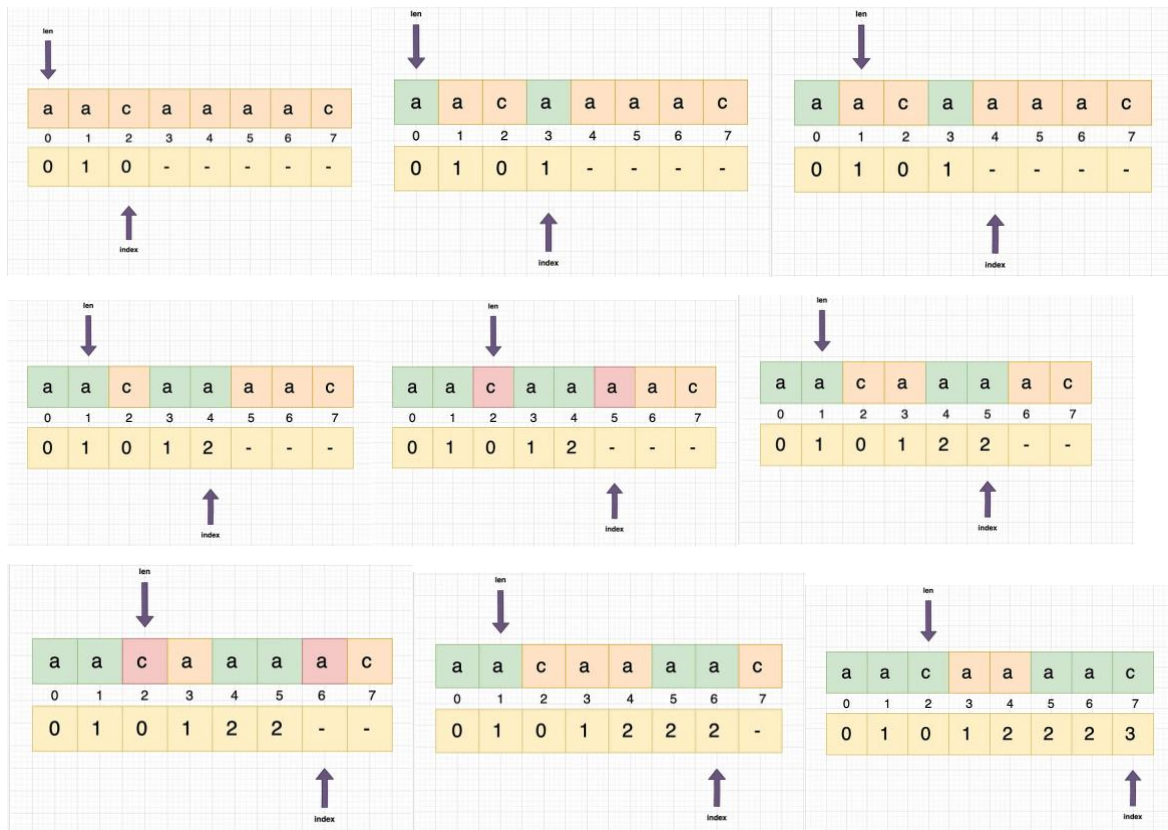
```

while index < len(pattern):
    if pattern[index] == pattern[length]:
        length += 1
        lps[index] = length
        index += 1
    else:
        if length == 0:
            // since a shorter prefix is not possible
            lps[index] = 0
            index += 1
        else:
            length = lps[length - 1]

return lps

```





(Das, 2021)

With the KMP algorithm instead of sliding the window by one character at a time, we slide it based on the value contained in the lps array, thus eliminating comparisons of characters that we know will not match.



**Lps Array**

a	a	c	a	a	a	a	c
0	1	0	1	2	2	2	3

(Galles & Lucet, 2015)  
Animation in Slide

**Pseudocode:**

```

j := 0
for i := 0 to n-1 {
  while (j > 0 and P[j] <> T[i]) {
    j := lps[j]
  }
  if P[j] = T[i] then j := j + 1
}

```

```
    if j = m then return i - m + 1
}
```

(Cornell University, 2002)

In this case we can increase the shift by the smallest amount that might allow a suffix of S to be a prefix of the new positioning of P, by setting j to lps[j]. This is repeated for mismatches by trying the next smaller prefix (which is also a suffix) to make sure no matches were missed. When j = m these searches have been exhausted and the pattern is checked from the beginning at the next index I (i-m+1). (Cornell University, 2002)

### III. Boyer-Moore

Boyer and Moore also extended the conditions of “what won’t match” and stated; “the pattern often allows the algorithm to proceed in large jumps through the text being searched.” This eliminates many of the possibilities for matches and greatly reduces the comparisons. “More generally, if the last occurrence of a char in pattern is so many positions from the right, then we know we can slide the pattern down that many positions without checking for matches.” (Boyer & Moore, 1977) As in the Knuth-Morris-Pratt algorithm, the only preprocessing is done on the pattern, allowing the algorithm to run at near real-time. The key feature of the Boyer is that it matches from the end of the pattern as opposed to the beginning. This method intrinsically leads to faster parsing of the text as the pattern grows larger.

**Time Complexity:**     *worst*  $O(n*m)$      |     *best*  $O(n/m)$

**Auxiliary Space:**  $O(k)$

Boyer Moore works *partly* due to the preprocessing to find the “bad character table”. With knowing the last instance of a char in the pattern we can slide the

window based on the bad character index of that char or by the entire length of the pattern for a character that does not appear in the pattern.

**Pseudocode: Bad Character Table (badChar)**

```
for all c, badChar[c] := 0
for j := 0 to m-1 {
  badChar[P[j]] := j
}
```

(Cornell University, 2002)

a b c    a a c a a a a b    a a c d a b a a c a a a c  
                                  a a c a **a** a a c

**Bad character**

d	a	b	*
0	1	2	3
index			len

(Galles & Lucet, 2015)  
Animation in Slide

**Pseudocode:**

```
s := 0 (* the current shift *)
while s <= n - m do {
  j := m - 1
  while P[j] = T[s+j] and j >= 0 do j := j - 1
  if j < 0 return s
  s := s + max(1, j - badChar[T[s+j]] + 1)
}
```

**Future**

The seismic shift in availability of Artificial Intelligence to the masses in due to the integration of natural language processing into user-friendly interfaces. NLP is what has made this possible, being “a subset of AI, NLP combines computational linguistics with statistical, machine learning, and deep learning models to understand and manipulate text” (Just, 2024; Hirschberg and Manning, 2015). This in turn has completely changed the way we interact with data and the possibilities of how “big data” can be consumed and processed in a new technological frontier. String



Matching algorithms such as the Levenshtein distance, Jaro similarity, and many more have improved this text analysis to which has been crucial in the implementation of these ever improving language models (Kalyanathaya et al., 2019).

Gene sequencing naturally has inherent commonalities with string matching. The growth of DNA identification and genetic research has run parallel with string matching techniques and often been the force behind its advancement. More target treatments are becoming available as effective sequencing and isolation are becoming possible. The most notable algorithm in this realm is the Smith-Waterman, which uses local sequence alignment to compare genes and proteins.

## References

- Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10), 762–772. <https://doi.org/10.1145/359842.359859>
- Cornell University. (n.d.). *Lecture 25: String Matching*. Wwww.cs.cornell.edu. <https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/lec25.htm>
- Das, A. (2021, August 31). *Preprocessing algorithm for KMP search (LPS array algorithm)*. Medium. <https://medium.com/@aakashjsr/preprocessing-algorithm-for-kmp-search-lps-array-algorithm-50e35b5bb3cb>
- Fun, D. (2024, January 9). *String Matching Algorithms : With the help of Java*. Medium. <https://www.codeutils.in/string-matching-algorithms-with-the-help-of-java-a07876c6ce28>
- Galles, D., & Lucet, Y. (2015). *Knuth-Morris-Pratt String Search Visualization*. Cmps-People.ok.ubc.ca. <https://cmps-people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html>
- GeeksforGeeks. (2011, April 1). *Naive algorithm for Pattern Searching*. GeeksforGeeks. <https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/>
- Hakak, S. I., Kamsin, A., Shivakumara, P., Gilkar, G. A., Khan, W. Z., & Imran, M. (2019). Exact String Matching Algorithms: Survey, Issues, and Future Research Directions. *IEEE Access*, 7, 69614–69637. <https://doi.org/10.1109/access.2019.2914071>
- Hirschberg, J., & Manning, C. D. (2015). Advances in natural language processing. *Science*, 349(6245), 261–266. <https://doi.org/10.1126/science.aaa8685>

Just, J. (2024). Natural language processing for innovation search – Reviewing an emerging non-human innovation intermediary. *Technovation*, 129, 102883. <https://doi.org/10.1016/j.technovation.2023.102883>

Kalyanathaya, K., Akila, D., & Suseendren, G. (2019). A Fuzzy Approach to Approximate String Matching for Text Retrieval in NLP A Fuzzy Approach to Approximate String Matching for Text Retrieval in NLP. *Journal of Computational Information Systems*, 15(3), 26–32.

Kamal, M., Alhendawi, & Suhaimi Baharudin, A. (2013). String Matching Algorithms (SMAs): Survey & Empirical Analysis. *Journal of Computer Sciences and Management*, 2637–2644.

Knuth, D., Morris, J., & Pratt, V. (1977). Fast Pattern Matching in Strings\*. *SIAM J. COMPUT*, 6(2).

Kumar Soni, K., Vyas, R., & Sinhal, A. (2014). Importance of String Matching in Real World Problems Importance of String Matching in Real World Problems. *International Journal of Engineering and Computer Science*, 3(6), 6371–6375.

Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics-Doklady*, 10, 707--710.

Miller, B. N., & Ranum, D. L. (2011). *Problem solving with algorithms and data structures using Python* (Second Edition). Franklin, Beedle & Associates.

*String-searching algorithm*. (2022, October 22). Wikipedia. [https://en.wikipedia.org/wiki/String-searching\\_algorithm](https://en.wikipedia.org/wiki/String-searching_algorithm)