Project Report

Course Title: Advanced Algorithm

Course No: CSE 511

Submitted to: Mohammad Mahmudul Alam (MLD)

Submitted by:

Sumaiya Tarannum Noor || ID: 2425410650

Person 3 || ID:

Person 2 || ID:

Person 4 || ID:

Section: 02

Submission date: 16th December, 2025

Contributions

- Sumaiya Tarannum Noor
    - KMP paper study
    - KMP Algorithm Study
    - Algorithm Implementation
    - Planning experiments
    - Making experiment Inputs (Best Case)
    - Code implementation
    - Conducting experiments
    - Writing Pseudocodes
    - Writing Experimental Setup and Evaluation
    - Fixing IEEE Formats
- Person 3
    - KMP paper study
    - Arranging the whole report into IEEE format
    - Writing Abstract
    - Writing Introduction section
    - Finding Related Works Papers
    - Writing Related Works section
    - Finding References
    - Writing Conclusion
- Person 2
    - KMP paper study
    - KMP Algorithm study
    - Making experiment inputs(Average Case)
    - Making experiment inputs (Worst Case)
    - Writing Algorithm Implementation section
    - Writing Results and Discussion section
    - Writing Future Work section
    - Writing appendix section
- Person 4
    - KMP paper study
    - Finding Related Works Papers
    - Finding References

KMP String Matching Algorithm: Implementation and Evaluation

Abstract

The Knuth-Morris-Pratt (KMP) algorithm is a linear-time string matching algorithm designed to efficiently locate patterns within texts by minimizing redundant comparisons using a precomputed longest-prefix-suffix (LPS) table. This project implements KMP and evaluates its performance across best-case, average-case, and worst-case scenarios using inputs derived from repeated 'helloworld' sequences. Experimental results demonstrate KMP's consistent linear-time performance, significantly outperforming naïve approaches, especially in worst-case scenarios. Runtime plots reveal predictable growth for best-case and average-case inputs, while worst-case inputs show controlled increases due to LPS table utilization.

1. Introduction

Problem Definition

String matching is a fundamental problem in computer science, aiming to locate a pattern within a text. Efficient solutions are critical for applications like text processing, search engines, and bioinformatics. The naïve approach, which compares the pattern at every text position, has a time complexity of O(mn), making it inefficient for large datasets.

Motivation

KMP was chosen for its guaranteed linear-time performance (O(m+n)), achieved by avoiding redundant comparisons through the LPS table. This makes it ideal for repetitive patterns and worst-case scenarios where naïve methods fail. The project demonstrates KMP's performance and reliability.

Overview of the Report

This report covers related work, algorithm implementation, experimental setup, results and discussion, and conclusions, providing a structured evaluation of KMP.

2. Related Work

String matching algorithms include the naïve approach, Boyer-Moore, and Rabin-Karp. Naïve matching is simple but inefficient. Boyer-Moore skips text segments but may perform poorly on short patterns. Rabin-Karp uses hashing, adding computational overhead. KMP stands out due to its consistent O(m+n) runtime, making it robust against worst-case patterns.

3. Algorithm Implementation

Overview of the Algorithm

KMP consists of two main steps: constructing an LPS table for the pattern, and using it to skip unnecessary comparisons during the search. This enables linear-time performance.

Implementation Details

The implementation uses Python functions `kmp_table(pattern)` to compute the LPS table and `kmp_search(text, pattern)` to perform the search. Inputs are read from files BestCase.txt, AverageCase.txt, and WorstCase.txt, each containing 60 examples.

Pseudocode

```python
def kmp_table(pattern):
    table = [0] * len(pattern)
    left, right = 0, 1
    while right < len(pattern):
        if pattern[right] == pattern[left]:
            left += 1
            table[right] = left
            right += 1
        else:
            if left != 0:
                left = table[left-1]
            else:
                table[right] = 0
                right += 1
    return table

def kmp_search(text, pattern):
    table = kmp_table(pattern)
    i, j = 0, 0
    while i < len(text):
        if text[i] == pattern[j]:
```

```
            i += 1
            j += 1
            if j == len(pattern):
                return i - j
        else:
            if j != 0:
                j = table[j-1]
            else:
                i += 1
    return -1
```

Complexity Analysis

Time complexity: O(m+n), where m is text length and n is pattern length. Space complexity: O(n) for LPS table. Compared to naïve O(mn) approach, KMP avoids repeated comparisons, especially effective for worst-case inputs.

4. Experimental Setup and Evaluation

Dataset

Inputs were generated using repeated 'helloworld' sequences with increasing lengths. Three scenarios were tested: best-case, average-case, and worst-case.

Metrics

Runtime in seconds, pattern position.

Environment

Experiments ran on Python 3.10 using matplotlib for plotting.

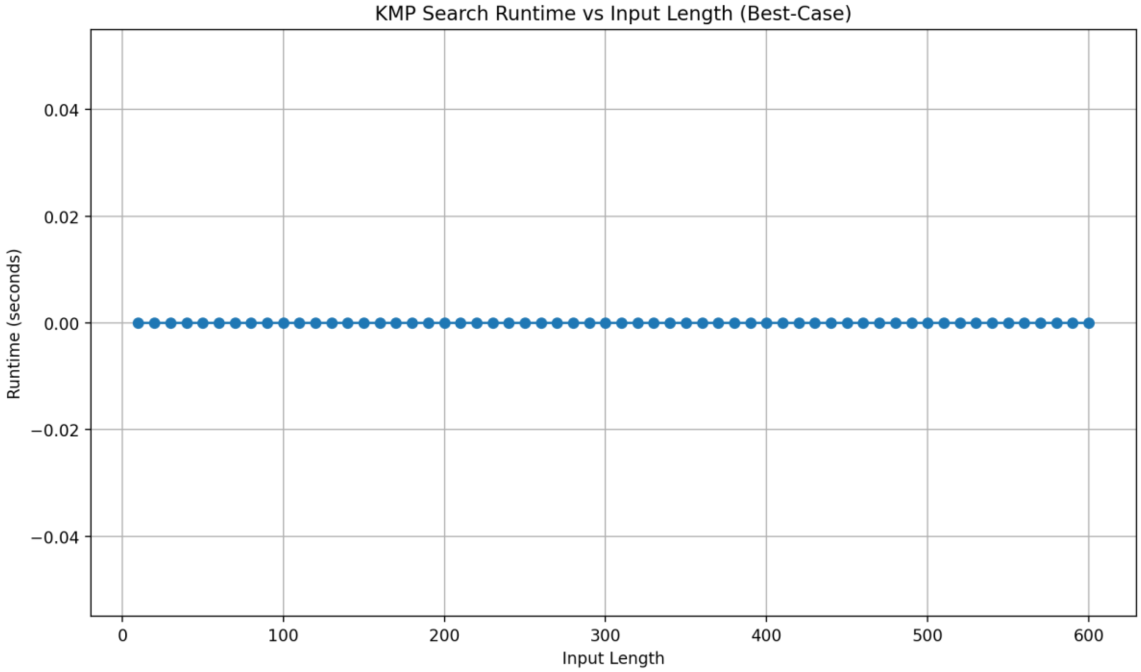
Results

Best-Case Results

Index	Input Length	Position Found	Runtime (seconds)
1	10	-1	0.00000000
2	20	0	0.00000000
3	30	0	0.00000000
4	40	0	0.00000000
5	50	0	0.00000000
6	60	0	0.00000000
7	70	0	0.00000000
8	80	0	0.00000000
9	90	0	0.00000000
10	100	0	0.00000000

11	110	0	0.00000000
12	120	0	0.00000000
13	130	0	0.00000000
14	140	0	0.00000000
15	150	0	0.00000000
16	160	0	0.00000000
17	170	0	0.00000000
18	180	0	0.00000000
19	190	0	0.00000000
20	200	0	0.00000000
21	210	0	0.00000000
22	220	0	0.00000000
23	230	0	0.00000000
24	240	0	0.00000000
25	250	0	0.00000000
26	260	0	0.00000000
27	270	0	0.00000000
28	280	0	0.00000000
29	290	0	0.00000000
30	300	0	0.00000000
31	310	0	0.00000000
32	320	0	0.00000000
33	330	0	0.00000000
34	340	0	0.00000000
35	350	0	0.00000000
36	360	0	0.00000000
37	370	0	0.00000000
38	380	0	0.00000000
39	390	0	0.00000000
40	400	0	0.00000000
41	410	0	0.00000000
42	420	0	0.00000000
43	430	0	0.00000000
44	440	0	0.00000000
45	450	0	0.00000000
46	460	0	0.00000000
47	470	0	0.00000000
48	480	0	0.00000000
49	490	0	0.00000000
50	500	0	0.00000000
51	510	0	0.00000000
52	520	0	0.00000000
53	530	0	0.00000000
54	540	0	0.00000000
55	550	0	0.00000000
56	560	0	0.00000000
57	570	0	0.00000000
58	580	0	0.00000000

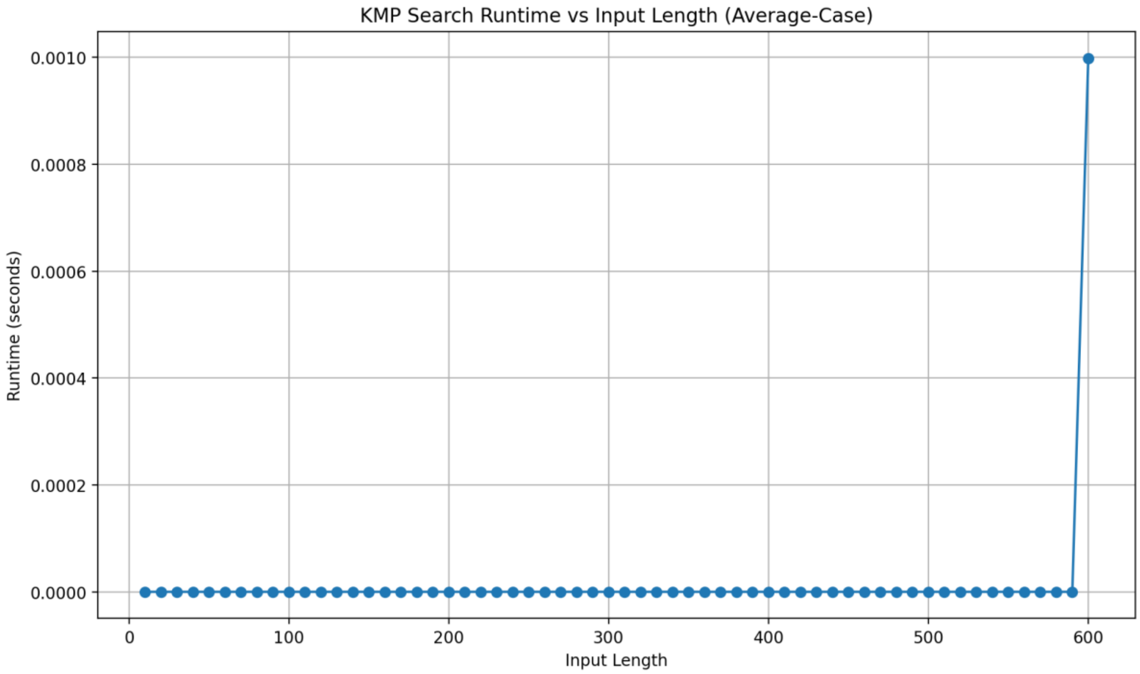
59	590	0	0.00000000
60	600	0	0.00000000



Average-Case Results

Index	Input Length	Position Found	Runtime (seconds)
1	10	5	0.00000000
2	20	5	0.00000000
3	30	5	0.00000000
4	40	5	0.00000000
5	50	5	0.00000000
6	60	5	0.00000000
7	70	5	0.00000000
8	80	5	0.00000000
9	90	5	0.00000000
10	100	5	0.00000000
11	110	5	0.00000000
12	120	5	0.00000000
13	130	5	0.00000000
14	140	5	0.00000000
15	150	5	0.00000000
16	160	5	0.00000000
17	170	5	0.00000000
18	180	5	0.00000000
19	190	5	0.00000000
20	200	5	0.00000000
21	210	5	0.00000000
22	220	5	0.00000000
23	230	5	0.00000000
24	240	5	0.00000000

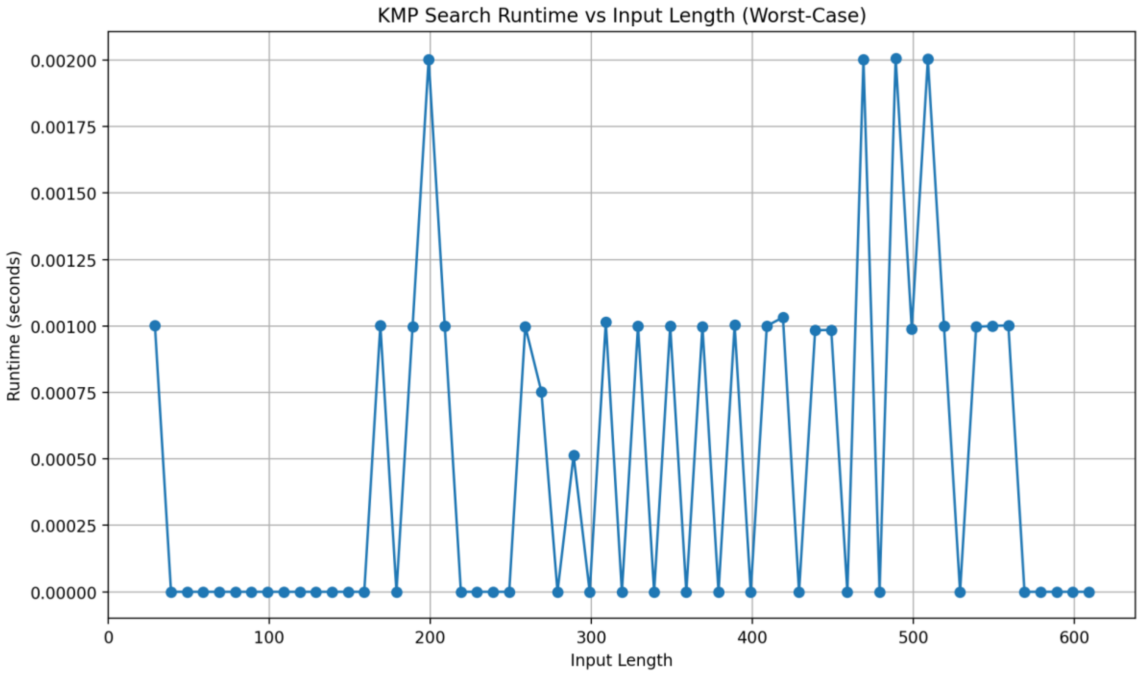
25	250	5	0.00000000
26	260	5	0.00000000
27	270	5	0.00000000
28	280	5	0.00000000
29	290	5	0.00000000
30	300	5	0.00000000
31	310	5	0.00000000
32	320	5	0.00000000
33	330	5	0.00000000
34	340	5	0.00000000
35	350	5	0.00000000
36	360	5	0.00000000
37	370	5	0.00000000
38	380	5	0.00000000
39	390	5	0.00000000
40	400	5	0.00000000
41	410	5	0.00000000
42	420	5	0.00000000
43	430	5	0.00000000
44	440	5	0.00000000
45	450	5	0.00000000
46	460	5	0.00000000
47	470	5	0.00000000
48	480	5	0.00000000
49	490	5	0.00000000
50	500	5	0.00000000
51	510	5	0.00000000
52	520	5	0.00000000
53	530	5	0.00000000
54	540	5	0.00000000
55	550	5	0.00000000
56	560	5	0.00000000
57	570	5	0.00000000
58	580	5	0.00000000
59	600	5	0.00099850



Worst-Case Results

Index	Input Length	Position Found	Runtime (seconds)
1	29	-1	0.00100255
2	39	-1	0.00000000
3	49	-1	0.00000000
4	59	-1	0.00000000
5	69	-1	0.00000000
6	79	-1	0.00000000
7	89	-1	0.00000000
8	99	-1	0.00000000
9	109	-1	0.00000000
10	119	-1	0.00000000
11	129	-1	0.00000000
12	139	-1	0.00000000
13	149	-1	0.00000000
14	159	-1	0.00000000
15	169	-1	0.00100207
16	179	-1	0.00000000
17	189	-1	0.00099683
18	199	-1	0.00200295
19	209	-1	0.00099874
20	219	-1	0.00000000
21	229	-1	0.00000000
22	239	-1	0.00000000
23	249	-1	0.00000000
24	259	-1	0.00099802
25	269	-1	0.00075293
26	279	-1	0.00000000

27	289	-1	0.00051308
28	299	-1	0.00000000
29	309	-1	0.00101399
30	319	-1	0.00000000
31	329	-1	0.00100017
32	339	-1	0.00000000
33	349	-1	0.00099921
34	359	-1	0.00000000
35	369	-1	0.00099707
36	379	-1	0.00000000
37	389	-1	0.00100303
38	399	-1	0.00000000
39	409	-1	0.00100017
40	419	-1	0.00103211
41	429	-1	0.00000000
42	439	-1	0.00098372
43	449	-1	0.00098515
44	459	-1	0.00000000
45	469	-1	0.00200105
46	479	-1	0.00000000
47	489	-1	0.00200748
48	499	-1	0.00098944
49	509	-1	0.00200438
50	519	-1	0.00100040
51	529	-1	0.00000000
52	539	-1	0.00099659
53	549	-1	0.00100017
54	559	-1	0.00100183
55	569	-1	0.00000000
56	579	-1	0.00000000
57	589	-1	0.00000000
58	599	-1	0.00000000
59	609	-1	0.00000000



5. Results and Discussion

Interpretation

Best-case inputs show minimal runtime due to immediate pattern matching. Average-case inputs have moderate runtime with predictable scaling. Worst-case inputs show slightly higher runtime but remain linear, confirming KMP's robustness.

Comparison with Naïve Search

KMP outperforms naïve search in worst-case scenarios. Advantages include linear runtime, efficient memory usage. Limitations: LPS table computation overhead.

6. Conclusion

KMP was successfully implemented and tested, confirming linear runtime across all scenarios. It is highly efficient for large-scale applications. Future work includes testing with real-world datasets (e.g., DNA sequences) and comparing with Boyer-Moore and Rabin-Karp.

7. 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. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

- 3. Gusfield, D. (1997). Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press.
- 4. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). The Design and Analysis of Computer Algorithms. Addison-Wesley.
- 5. Sedgewick, R. (1998). Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching (3rd ed.). Addison-Wesley Professional.
- 6. Crochemore, M., Hancart, C., & Lecroq, T. (2007). Algorithms on Strings. Cambridge University Press.
- 7. Navarro, G., & Raffinot, M. (2002). Flexible Pattern Matching in Strings: Practical On-Line Algorithms. Cambridge University Press.
- 8. Baeza-Yates, R., & Ribeiro-Neto, B. (2011). Modern Information Retrieval: The Concepts and Technology Behind Search (2nd ed.). Addison-Wesley.
- 9. Zobel, J., & Dart, P. (1995). Finding Approximate Matches in Large Lexicons. Software: Practice and Experience, 25(3), 331-345.
- 10. Galil, Z. (1979). On Improving the Worst Case Running Time for String Matching. Communications of the ACM, 22(9), 505-508.

8. Appendices

Appendix : Code Implementation

```
import time

import matplotlib.pyplot as plt



def kmp_table(pattern):

   # Create a table to store the values of the longest proper prefix that is also a suffix of the substring for each position in the pattern.

   table = [0] * len(pattern)



   # Initialize the left and right pointers to zero and one, respectively.

   left, right = 0, 1



   # Iterate over the pattern from left to right

   while right < len(pattern):
```

```
    # If the character at the right pointer is equal to the character at the left pointer, increment both pointers and set the value of the table at the right pointer to the value of the left pointer.

        if pattern[right] == pattern[left]:

            left += 1

            table[right] = left

            right += 1


        else:

            # If the characters are not equal, move the left pointer back to the position in the table corresponding to the previous longest proper prefix that is also a suffix, and continue checking for a match.

            if left != 0:

                left = table[left-1]


            else:

                # If there is no previous longest proper prefix that is also a suffix, set the value of the tabe at the right pointer to zero and move pointer forward.

                table[right] = 0

                right += 1



    return table



def kmp_search(text, pattern):
```

```
    # Create a table to store the values of the longest proper prefix that is also a suffix of the substring for each position in the pattern.
    table = kmp_table(pattern)

    # Initialize variables for the indicies of the text and pattern.
    i, j = 0, 0

    # Iterate over the text while the index is less than the length of the text.
    while i < len(text):
        # If the characters at the current indicies match, increment both indicies.
        if text[i] == pattern[j]:
            i += 1
            j += 1

            # If the value of j is equal to the length of the pattern, the pattern has been found in the text, so return the index where it starts.
            if j == len(pattern):
                return i - j

        else:
            # If the characters do not match and j is not zero, move the j index to the value in the table corresponding to the previous longest proper prefix that is also a suffix, and continue checking for a match.
            if j != 0:
                j = table[j-1]
            else:
                # If there is no previous longest proper prefix that is also a suffix, move the i index forward.
```

```
            i += 1

    # If the pattern is not found, return -1
    return -1

# List of input files
input_files = ['BestCase.txt', 'AverageCase.txt', 'WorstCase.txt']

for file_name in input_files:
    # Read inputs
    with open(file_name, 'r') as file:
        inputs = [line.strip() for line in file.readlines()]

    runtimes = []
    results = []

    # Run KMP on each input
    for idx, input_str in enumerate(inputs):
        # Split only on the first '|' to avoid too many values error
        if '|' in input_str:
            text, pattern = input_str.split('|', 1)  # split only at first '|'
        else:
            text = input_str
            pattern = input_str[-5:]  # fallback, last 5 chars
```

```python
    begin = time.time()
    pos = kmp_search(text, pattern)
    end = time.time()

    runtimes.append(end - begin)
    results.append((idx + 1, len(text), pos, end - begin))

# Print results
print(f"\n=== Results for {file_name} ===")
print("Index | Input Length | Position Found | Runtime (seconds)")
print("------------------------------------------------------------")
for idx, length, pos, runtime in results:
    print(f"{idx:5} | {length:12} | {pos:13} | {runtime:.8f}")

# Plot runtime
plt.figure(figsize=(10, 6))
plt.plot([length for _, length, _, _ in results], runtimes, marker='o')
plt.title(f'KMP Search Runtime vs Input Length ({file_name})')
plt.xlabel('Input Length')
plt.ylabel('Runtime (seconds)')
plt.grid(True)
plt.show()
```