CS 225 EC Project: Knuth-Morris-Pratt (KMP) Search Algorithm

Simone-Nicole Angelov and Archie Mucharla

Dataset Overview

We have five final datasets. The table below describes their approximate size in terms of n, where n is defined as the number of characters in the respective dataset.

Dataset Name	n	Description
1_spotify	1,641	Top songs on Spotify
2_horror_movie	22,007	Horror movie titles
3_starbucks_reviews	402,145	Reviews from Starbucks customers
4_makeup	581,486	Makeup products and related information
5_amazon	1,199,836	Products and descriptions from Amazon UK

Algorithm Accuracy and Correctness

Our test suite uses the Catch2 framework and can be divided into several distinct parts, each serving a specific purpose in validating different aspects of the algorithms:

1. **Preprocessing tests:**

The first section of our suite is dedicated to the preprocessing function of the KMP algorithm. It involves creating the Longest Prefix Suffix (LPS) table, essential for the algorithm's efficiency. This part of the suite includes tests for patterns of varying complexities, from single characters to complex repetitive sequences. By ensuring the LPS table is correctly computed, we validate the foundation upon which our KMP search algorithm operates.

Test Case Name	Pattern Tested	Expected LPS Array	Test Result

LPS Table - single char	" "	{0}	Pass
LPS Table - Non-repetitive Pattern	"ABCDEFG"	{0, 0, 0, 0, 0, 0, 0}	Pass
preprocess test	"ABABAC"	{0, 0, 1, 2, 3, 0}	Pass
preprocess test with repetitive pattern	"AAABAAA"	{0, 1, 2, 0, 1, 2, 3}	Pass
Complex Repetitive Pattern	"AABAACAABAAA"	{0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5, 2}	Pass

2. KMP Algorithm Correctness Tests

In this next section of our test suite, we test the correctness of the KMP search algorithm. Various scenarios are covered here, ranging from basic pattern matching to more challenging cases like repeating patterns and patterns longer than the text. Here, we also test for edge cases such as empty strings and patterns with repeated sub-patterns. These tests demonstrate the KMP algorithm's capability to accurately locate patterns. In order to verify correctness, we compared the algorithm’s results with a manual calculation of what index should be printed in that case.

The following is an overview of the test cases:

Test Case	Text	Pattern	Expected Outcome	Test Result
Simple Pattern	"abczzzzzzzzzz"	"abc"	index 0	Pass
Repeating Pattern	"abcabcabcabc"	"abc"	indices 0, 3, 6, 9	Pass
Pattern Longer Than Text	"abc"	"abcd"	No match	Pass
Pattern with Repeated Sub-Patterns	"ababcabcacab"	"abcac"	index 5	Pass

Empty String	""	"abc"	No match	Pass
Empty Pattern	"abcabc"	""	No match	Pass

3. Naive Algorithm Correctness Tests

In parallel with our KMP algorithm tests, we conducted the same correctness tests for the naive search algorithm. These tests mirrored the scenarios used in the KMP tests, encompassing a range of patterns and edge cases.

4. KMP vs. Naive Search

This last section of the test suite focuses on varying dataset sizes and comparative analyses between the KMP algorithm and a naive search approach. For instance, consider the following test case from our Spotify dataset:

```
TEST_CASE("Preprocessing and Single Match: Spotify (1)", "[1]") {

    std::string text = read_file_content("../data/1_spotify");

    std::string pattern = "peach";

    std::vector<int> lps(pattern.size(), 0);

    // Preprocess the pattern to generate the LPS array

    preprocess_pattern(pattern, lps);

    std::vector<int> expectedLPS = {0, 0, 0, 0, 0};

    REQUIRE(expectedLPS == lps);

    // Perform KMP search and check result

    KMPResult kmpResult = KMP_search(text, pattern, lps);

    std::vector<int> expectedKMPIndices = {53};

    REQUIRE(expectedKMPIndices == kmpResult.matchStartIndices);

    REQUIRE(kmpResult.totalComparisons >= (int) text.size());

    // Perform naive search and check result
```

```
   KMPResult naiveResult = naive_search(text, pattern);

   std::vector<int> expectedNaiveIndices = {53};

   REQUIRE(expectedNaiveIndices == naiveResult.matchStartIndices);

   // The total number of comparisons in KMP should be less than or equal to that in
the naive approach

 REQUIRE(kmpResult.totalComparisons <= naiveResult.totalComparisons);
```

Rationale Behind Test Input:

- The pattern ‘peach’ was chosen because it only appears once in the Spotify dataset. This allows us to precisely predict its location and validate the accuracy of our algorithm in finding singular matches.
- The preprocessing step computes the Longest Prefix Suffix (LPS) array for the pattern “peach”. The expected LPS array {0, 0, 0, 0, 0} correctly reflects the absence of repeating prefixes and suffixes in the aforementioned pattern.

For each of the five datasets, we ran three tests. A single match (like above), multiple match, and no match case.

A key metric in our analysis is the total number of character comparisons made by the KMP algorithm. This count offers insight into the algorithm’s efficiency. We check for reasonable bounds by ensuring that totalComparisons is never less than the text length, as each character must be examined at least once, and we also ensure it is not excessively high. Contrary to the random nature of naive search, KMP strategically leverages the LPS array to skip unnecessary comparisons, which potentially reduces the total number of character comparisons. This is evidenced when comparing the KMP’s totalComparisons against those of a naive search. In an optimized KMP algorithm, the number of comparisons should be less than or equal to that of the naive search.

Our tests delve deeper into the KMP Search’s accuracy by benchmarking its number of character comparisons against the naive approach. For a given pattern, the KMP algorithm’s efficiency is validated if it consistently uses fewer or equal comparisons than the naive method.

Benchmarking and Big-O Analysis

The benchmarking of our algorithm was conducted using five distinct datasets of increasing size, offering a depiction of how the algorithm scales with larger inputs. For each dataset, we utilized the Unix 'time' command to measure the execution time of our algorithm. This process involved executing the command ‘time ./test [#dataset number]’ for each specific dataset. Within each dataset, we recorded the execution times as comments within their respective test cases, allowing us to compare timings across different test cases.

To provide a visual representation of our findings, we plotted the runtimes for each dataset on a graph with labeled axes. The graph below illustrates the relationship between the size of each dataset and the corresponding execution time. For consistency, we focused on the single-match test case for each dataset. For example, in the 'Spotify' dataset (smallest), we used the test case "Preprocessing and Single Match: Spotify" to plot (n, runtime):

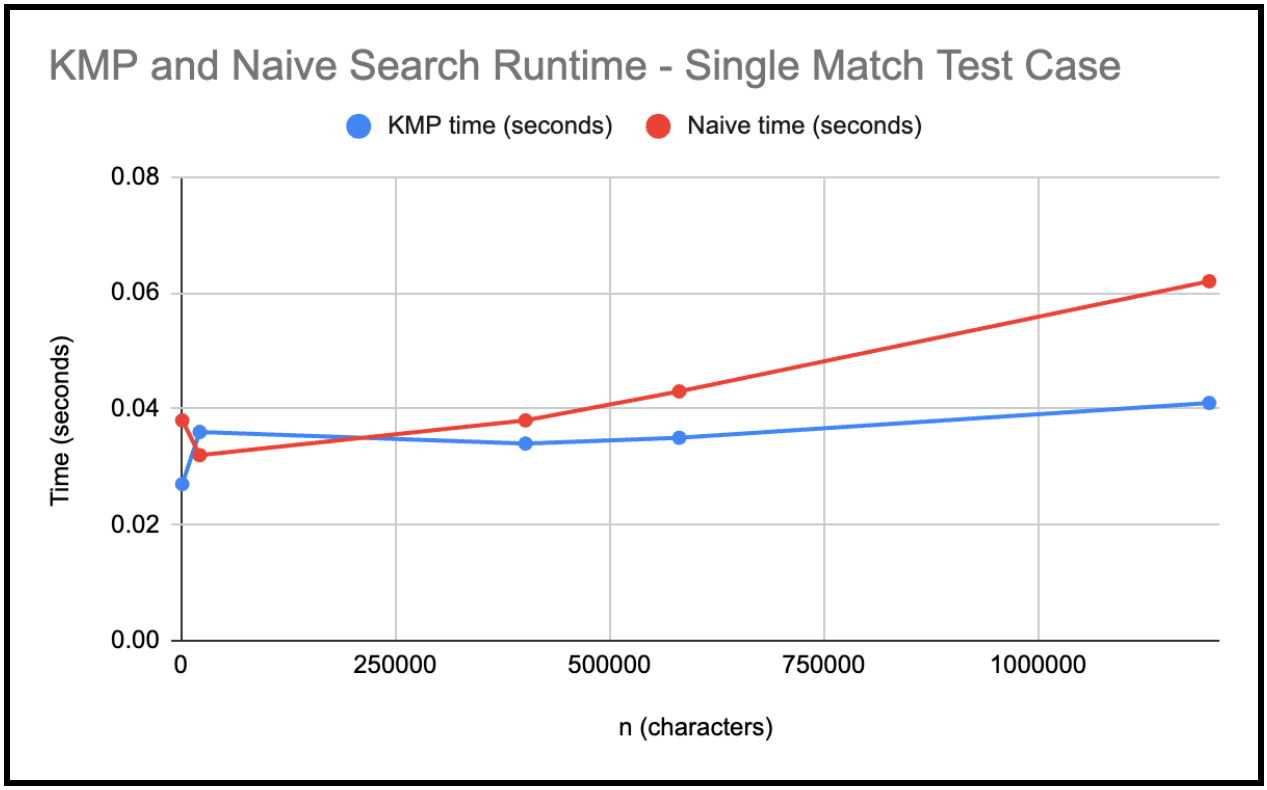
KMP: (1,641 characters, 0.027 seconds)

Naive: (1,641 characters, 0.038 seconds)

This approach was replicated for the same type of test case across all datasets, including 'Horror Movie', 'Starbucks Reviews', 'Makeup Reviews', and 'Amazon Reviews'. Using the same single-match test case for all the points allowed us to best assess the efficiency of our algorithm since the main varying factor was the size of the dataset.

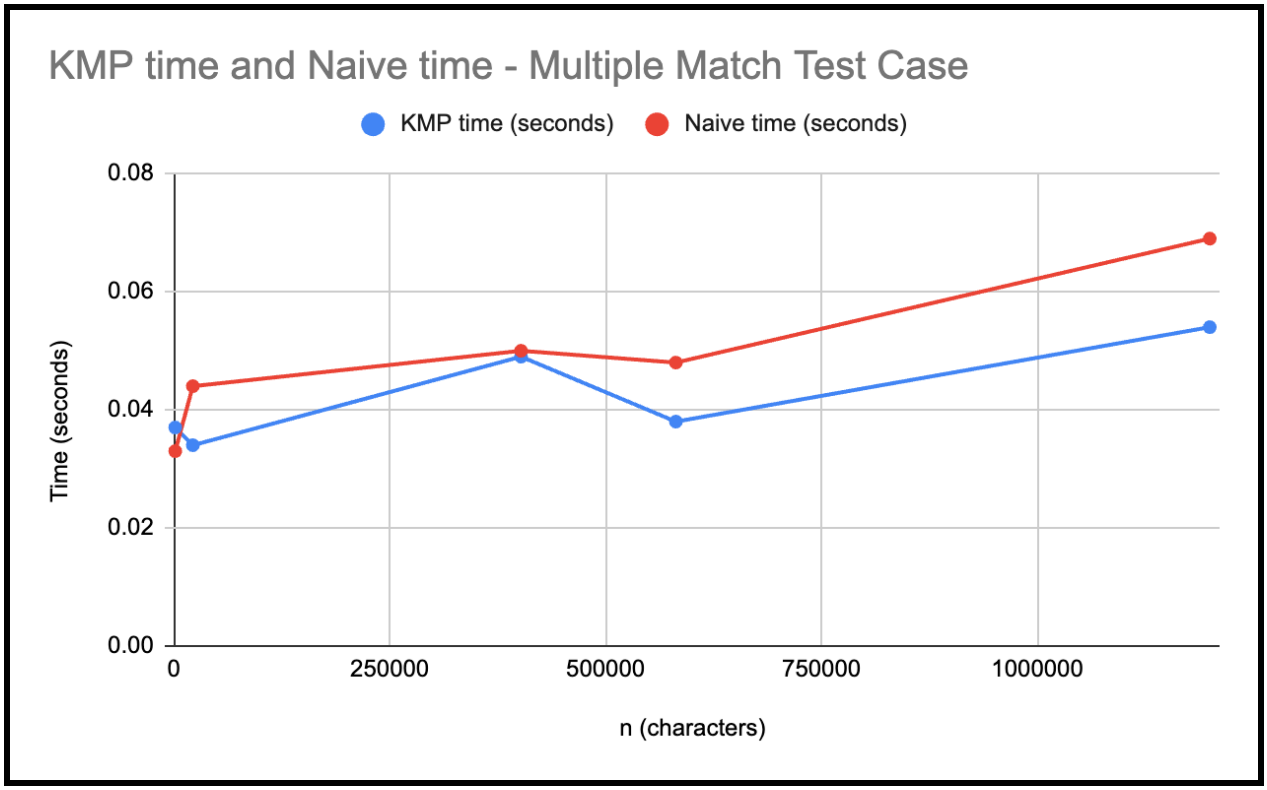
For the KMP algorithm, we anticipated a theoretical time complexity of O(n + m), where 'n' is the length of the text and 'm' is the length of the pattern. This expectation stems from the nature of the KMP algorithm, which preprocesses the pattern in O(m) time to create an array of longest proper prefixes and suffixes (LPS array) and then scans through the text in O(n) time. Our empirical data largely supported this theoretical complexity. As the dataset sizes increased, the runtimes showed a trend that was consistent with a linear growth pattern, indicative of the O(n + m) behavior. This alignment between empirical observations and theoretical expectations demonstrates the efficiency of the KMP algorithm, particularly in handling larger datasets where its optimized pattern-searching capabilities become most evident.

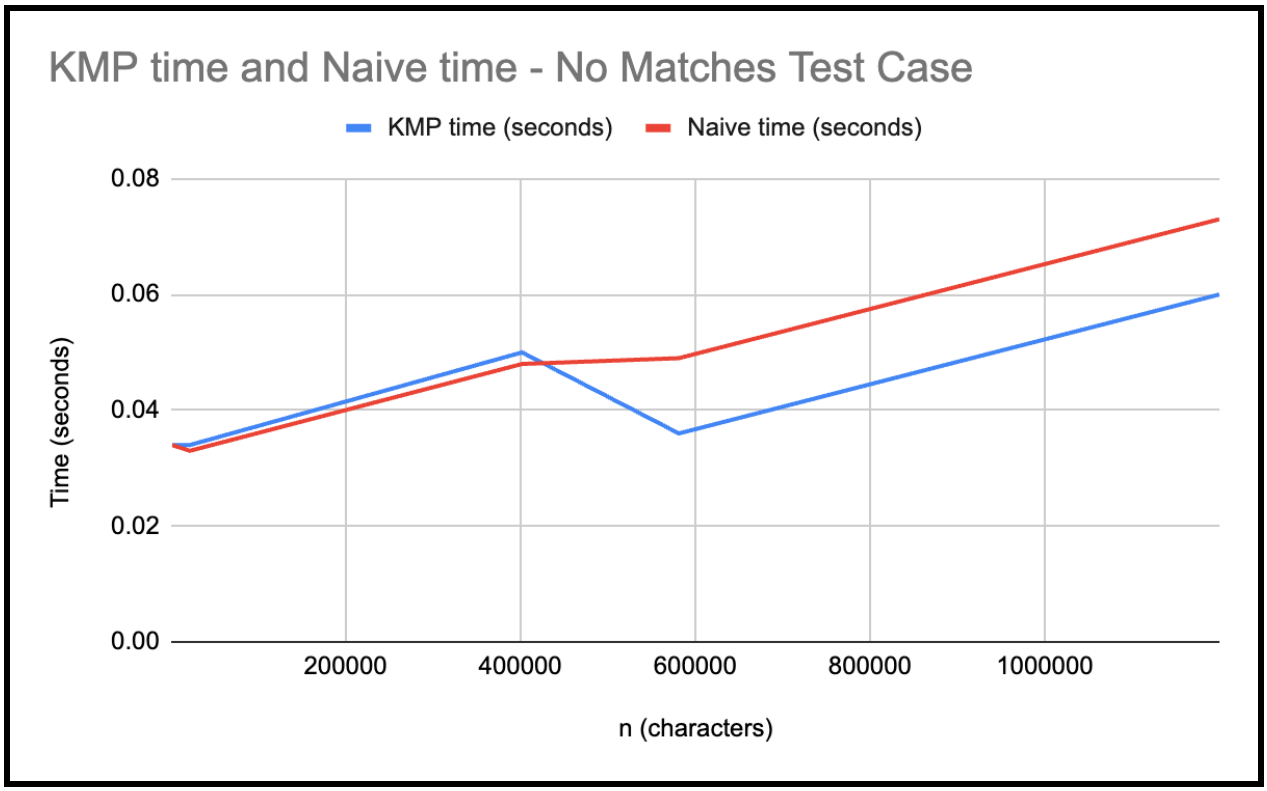
In contrast, the naive search algorithm, which was used as a baseline in our tests, exhibited a higher time complexity in the graph, aligning with its theoretical O(n*m) notation. The comparison between the naive algorithm and the KMP algorithm in our benchmarking highlighted improvements in efficiency achieved by the KMP algorithm.



It's important to note that the expected trend was more pronounced in larger datasets. In the case of smaller datasets, we observed some deviations from this expected linear pattern. It is worth considering that these deviations are potentially due to issues with repeatability and precision of runtime measurements in smaller datasets. Given the shorter durations involved, repeated runs of the same test can yield slightly different results. This inconsistency leads to the challenges in precisely measuring algorithm performance at a smaller scale, which is where the deviations in the graph may have come from.

Driven by curiosity, we extended our analysis to include the performance graphs for scenarios with multiple matches and no matches. These additional analyses reinforce our initial findings: as the data size increases, the runtime performance of both algorithms exhibits a linear growth pattern. Notably, the KMP algorithm consistently demonstrates superior efficiency over the naive method in larger data sets. The KMP search runtimes were, more often than not, shorter than those of the naive search. Only in smaller data sets did we observe any deviation from this trend, suggesting that the KMP algorithm's advantages are most noticeable when managing substantial volumes of data.


KMP time and Naive time - Multiple Match Test Case
KMP time (seconds)
Naive time (seconds)
0.08
0.06
0.04
0.02
0.00
Time (seconds)
0
250000
500000
750000
1000000
n (characters)


KMP time and Naive time - No Matches Test Case
KMP time (seconds)
Naive time (seconds)
0.08
0.06
0.04
0.02
0.00
Time (seconds)
200000
400000
600000
800000
1000000
n (characters)

In this final section, we present a detailed walkthrough of our KMP_search() function's pseudocode. This walkthrough emphasizes the algorithm's significant Big O runtimes. The input for KMP_search() is:

- text: The text in which the pattern is to be searched
- pattern: The pattern to be searched in the text
- lps: The Longest Prefix Suffix array

The output of KMP_search() is a KMPResult struct which consists of indices of pattern matches in the text and the total number of comparisons made. Below is the pseudocode walk through of the KMP_search() function.

1. Initialization
    a. Create a KMPResult struct with an empty list for match indices and set totalComparisons to 0.
    b. Initialize patternIndex and textIndex to 0.
2. Edge Case Handling **(O(1) Time Complexity)**
    a. If either the pattern or text is empty, immediately return the empty result struct.
3. Main Text Traversal Loop **(O(n) Time Complexity)**:
    a. Here, n represents the length of text (in characters)
    b. While textIndex is less than the length of text:
        i. Increment totalComparisons.
        ii. If pattern[patternIndex] matches text[textIndex]:
            1. Increment both patternIndex and textIndex.
            2. If patternIndex equals the length of pattern:
                a. Record this match by adding (textIndex - patternIndex) to the match indices.
                b. Update patternIndex using lps[patternIndex - 1].
        iii. If there's a mismatch and patternIndex is not 0:
            1. Update patternIndex using lps[patternIndex - 1]
        iv. If patternIndex is 0, simply increment textIndex
4. Return Results
    a. Return the KMPResult struct containing the match indices and the total number of comparisons made

The primary computational effort occurs in the main loop (Step 3), which traverses the text once, leading to an O(n) time complexity, where n is the text's length in characters. The character comparison (Step 3b) and the LPS array lookup (Steps 3b.ii and 3c.i) are executed in constant time, thus maintaining the overall linear time complexity. The initial edge case check (Step 2) is also a constant time operation, O(1). Consequently, the KMP search function operates with an overall time complexity of O(n). This efficiency is pivotal for string search operations,

particularly when dealing with large texts. The strategic use of the LPS array significantly reduces the number of comparisons, especially in scenarios of pattern mismatches, which enhances the algorithm's efficiency.