COSC 320 – 001

Analysis of Algorithms

2022/2023 Winter Term 2

String Matching Project - Fourth Milestone

Rabin-Karp and LCS Implementations

Comparison Between the Three Algorithms

Group Members: Minh Bui, Neilansh Rajpal, Anamica Sunderaswara

Group Lead: Neilansh Rajpal

Project Report

Abstract

This report provides an analysis and overall summary of three foundational string matching algorithms: the Knuth-Morris-Pratt (KMP) algorithm, the Longest Common Subsequence (LCS) algorithm, and the Rabin-Karp algorithm

The KMP algorithm uses a precomputed prefix function to efficiently find incidents of a pattern in a text. The LCS algorithm works by identifying the longest common subsequence between two strings and using a matrix to match a pattern with a text. Lastly, the Rabin-Karp algorithm implements a hash function to search for a pattern in the text.

This paper summarizes the research done regarding each algorithm from the previous milestones, as well as compares and contrasts their performance when implemented on a database of our choice.

Summary

KMP

The Knuth-Morris-Pratt (KMP) algorithm is a string matching algorithm that finds instances of a pattern in a provided text by making use of a precomputed prefix. The prefix function refers to an array created that holds the lengths of the longest proper prefixes of the pattern being searched that double as suffixes for each of its substrings.

The prefix function is an array of size m (equivalent to the length of the pattern being searched) and is often denoted as $lps[1\ldots m]$ (longest prefix that is also a suffix) or $\Pi[1\ldots m]$.

The algorithm begins by iterating through the text being searched from left to right, checking at every instance to compare if the characters from the text match with those from the pattern. At the point of a mismatch, the prefix function comes into play to help determine where the algorithm should restart - which is done by providing the next position in the text containing the maximum number of characters that match the text thus far.

Through using the prefix function, you are avoiding needless comparisons, reducing the run time from the brute-force naive method of $O(nm)$ to $O(n+m)$ with n being the length of the text and m being the length of the pattern.

In general, the KMP algorithm is very efficient and powerful in the area of string matching.

LCS

The Longest Common Subsequence (LCS) algorithm finds the longest subsequence common to two texts. A subsequence is a succession of characters derived from a string by removing at least zero of its elements without changing its original order.

Its most common implementation uses Dynamic Programming (DP). Using this approach, we begin by creating a *n*X*m* matrix *DP*, where *n* and *m* are lengths of the *text* and *pattern* respectively. We compare the letters of the two strings and fill out the cells corresponding to their positions in the matrix *DP*. When characters match, the matrix's value at the current position *DP[i][j]* is assigned the sum of value in the left diagonal and 1. If they don't match, *DP[i][j]* is set to the bigger value between the cell above it and to the left of it.

After the matrix has been filled, the last element *DP[M][N]* gives us the length of the largest common subsequence occurring in both the *text* and *pattern*.

This memoization technique improves the algorithm and results in a runtime of *O(n * m)*, which is very efficient as compared to that of its naive approach, $O(n * 2^n)$.

Rabin-Karp

The Rabin-Karp algorithm is a frequently implemented algorithm used for identifying patterns in a given text. The algorithm forms a fingerprint value for the pattern being searched and creates a subsequent hash value for a substring of the text.

By using the rolling hash function, the algorithm verifies the subset's hash value compared to the formulated fingerprint to see if there is a match.

In the case that there is a match, the algorithm will notice the user of the duplicacy. If there isn't a match, the algorithm will change the hash value to represent the next available substring and re-begin checking for a potential match.

With this algorithm, the average time complexity reduces to *O*(*n*+*m*), with *n* representing the length of the text and *m* the length of the pattern being searched. Rabin-Karp is very effective as hash functions generally have a low collision probability and remain overall efficient.

Code

We decided to choose Jupyter Notebook to implement the algorithms, and make use of the python libraries such as **pandas** and **matplotlib** to extract fields from the dataset and create visuals.

The implementation for every algorithm can be found in the same .ipynb file. We also tested the algorithms with the dataset and created visuals for each (with increasing value of *n*, the text size).

Finally we ran a test to measure runtime for all algorithms and plotted them together for comparison.

The code can be found in the following GitHub repository here.

Running the code

1. Head to the GitHub repository through the link provided above, and clone the repository on your local machine. All the code is contained within the Jupyter Notebook (.ipynb) file.
2. Download the dataset. The link can be found under the **Dataset** section below.
3. Python would be needed to run the code.
4. After opening the code in an editor, the dependencies can be installed by running the very first command in the code.
5. The code is now ready to run.

Dataset

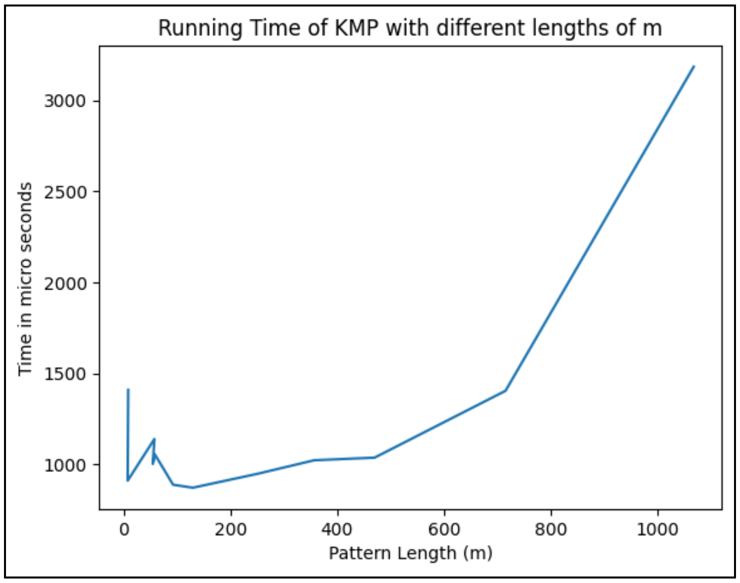
After having a discussion with the professor, due to the topic chosen, we were suggested to measure running time as a function of the length of the text rather than number of texts iterated through from the dataset.

As a result we prioritized a dataset having large texts (such as research papers, news articles) with differing lengths and (10000+ characters). Therefore, another dataset was chosen for Milestone 4.The chosen dataset consists of COVID-19 research papers, and can be found here.

Comparison

1. **Testing the effect of pattern length *m* on runtime**

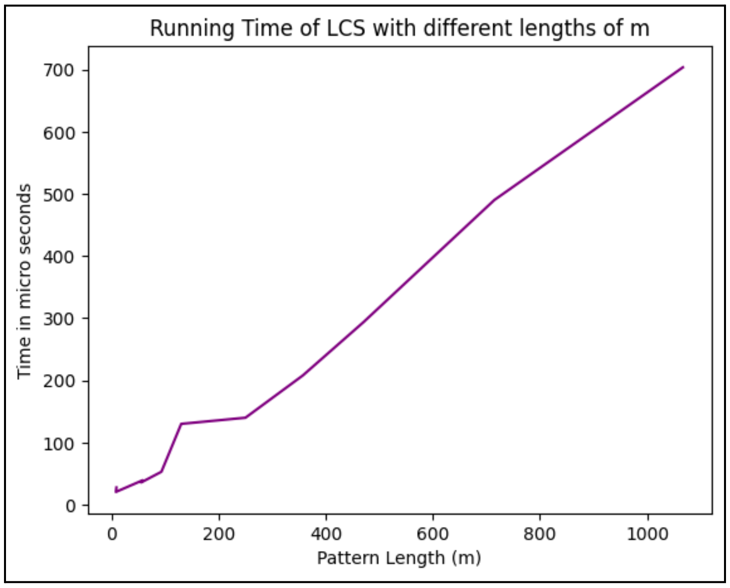
   For the 3 algorithms, we tested the impact of runtime with different lengths of pattern text. For an article text of fixed length *n*, we tested how different lengths *m* of pattern affected the runtime. The following plots were obtained:



The graph shows the relationship between Runtime and pattern text length *m* is not clear for the KMP algorithm.

A net increase in runtime is observed for *m > 600 characters* for the pattern text.

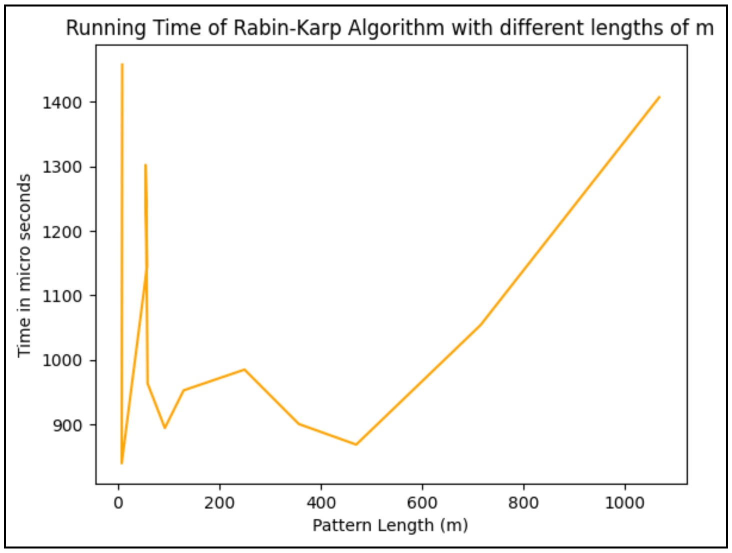
This also agrees with its complexity which is *O(n+m)* in the worst case.



In case of LCS, the runtime increases when the pattern text length *m* increases.

The association between the two can be considered linear.

This is in agreement with its complexity, which is *O(nm)*



A pattern similar to KPM is observed for Rabin-Karp.

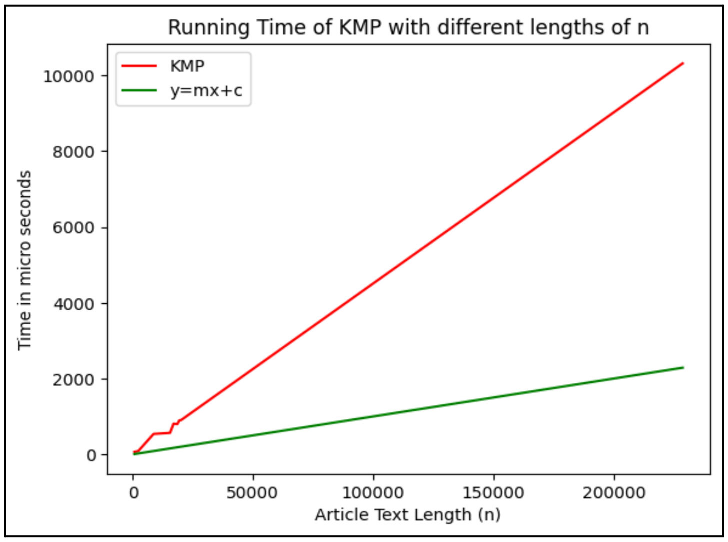
While a general pattern can't be established, the observations suggest that the runtime increases after pattern length of *m > 500 characters*.

2. **Testing the effect of <u>text length</u> *n* on runtime**

Similar tests were run for the 3 algorithms, but this time the pattern length *m* was kept constant, and the effect of article text length *n* on runtime was tested. Articles from the dataset were browsed through randomly and the plots (Runtime vs Text length *n*) for the algorithms were plotted. The tests were run separately. Here are the results:

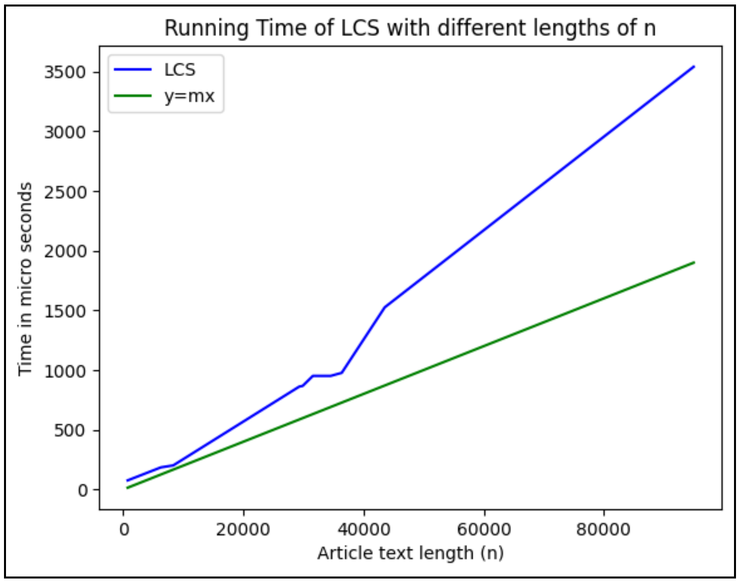
A green straight line was plotted in each plot for reference.

In the plots below, the values of n (on the x-axis) correspond to the number of characters in the article text.



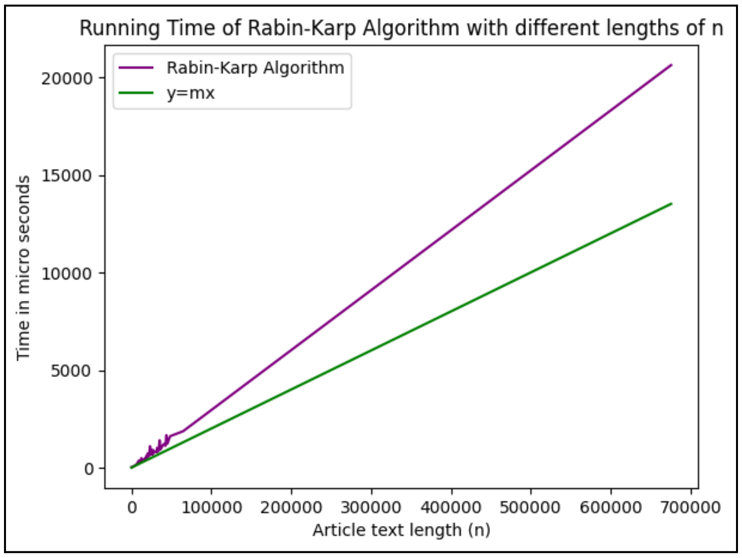
The plot for KMP suggests a linear relationship between text length *n* and runtime.

Even though the extra step of preprocessing is *O(m)*, seeing a linear correlation in this case suggests that the runtime may be largely influenced by text length *n*, and thus the complexity of KMP could be written as *O(n)* for *n>>m*.



A straight line is also obtained for LCS, which is consistent with the actual complexity of this algorithm, *O(n * m)*.

For *n >> m*, the complexity can be rewritten as *O(n)*.



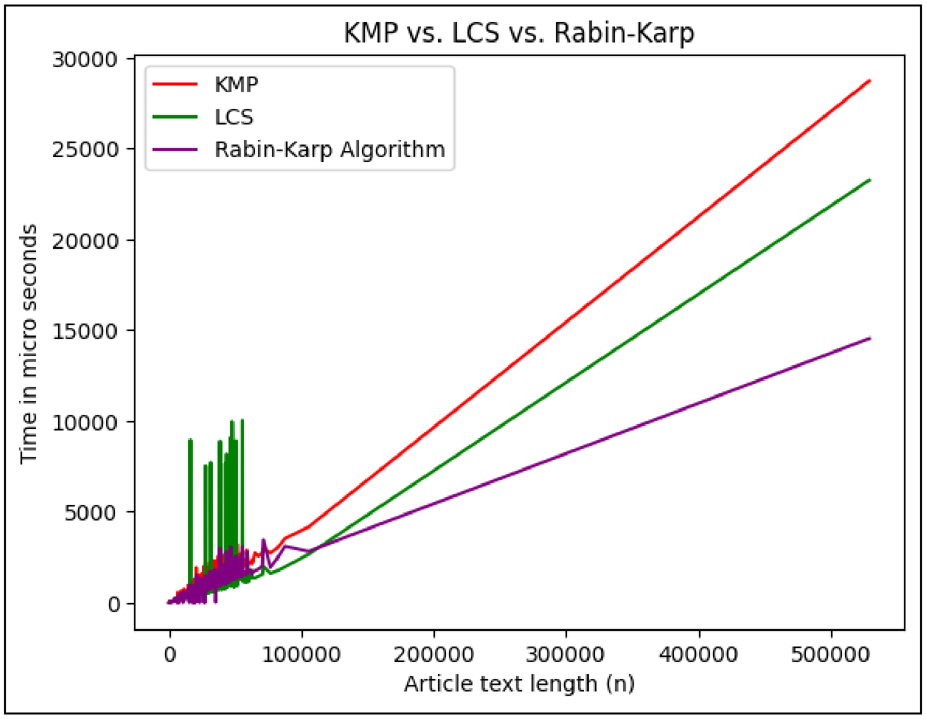
We can conclude that for the Rabin-Karp algorithm, there is also a linear correlation between text length *n* and the runtime.

This also agrees with our previous claim that the runtime is largely dictated by text length *n*.

Again, for *n >> m*, the complexity can be considered as *O(n)*.

3. Comparing the algorithms

Finally, to compare all the 3 algorithms, we iterated through 500 random articles from the dataset and plotted the graphs (Runtime vs text length *n*) in a single plot.



The values of *n* range from 100000 to 500000 and the plot depicts how the runtime for the algorithms increases with increase in *n*.

Evidently, the article text length *n* and the runtime have a linear relationship for all the 3 algorithms. In other words, the efficiency of the algorithms is largely influenced by the length of text *n*. However, the Rabin-Karp algorithm appears to grow slower with increase in *n* as compared to KMP and LCS. This could be attributed to the less number of comparisons made in the Rabin-Karp algorithm, since the pattern is only compared with the current window of the main text when their corresponding hash values match.

Task Separation

Neilansh: Created test methods for the algorithms, compared them and created visuals. Discussed LCS in this Milestone and the video.

Minh: Discussed the summary and analysis for the Rabin-Karp algorithm in the report, worked on its implementation (discussed after the report), edited and put together the video.

Anamica: Wrote the abstract, and the summary for the KMP algorithm, along with the Rabin-Karp algorithm in the report. Discussed the Overview and KMP algorithm in the video.

References

1. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

2. https://www.programiz.com/dsa/longest-common-subsequence
3. https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/
4. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/
5. https://www.programiz.com/dsa/rabin-karp-algorithm

Rabin-Karp - Implementation (Analyzed in Milestone 3)

Overview

This section is a continuation of the Rabin-Karp algorithm that we analyzed in Milestone 3. The following code is our implementation of this algorithm in Python.

1. **rabin_karp_search(*text, pattern*):** Function that performs the Rabin-Karp algorithm on the *text (length n)* and the *pattern (length m)*. It first calculates the hash value of the *pattern* and the first "window" of the *text*[1…*m*]. Then it iterates over the text and slides the window, comparing hash values of the *pattern* and the current window of the *text*. It only compares the characters in the current window of *text* and *pattern* if the hash values match. If all the characters match in the current window of *text* and *pattern*, a match is found and the method returns *true*. If the hash values don't match, it moves to the next window of *text* and compares its hash value to that of the *pattern*. End of the loop signifies that no match is found, which is when *false* is returned.

   We first tested our algorithm by performing it on the whole dataset for a pattern. Then we printed the results. To analyze the runtime, we created 2 test methods for different test cases which are presented in the next section.

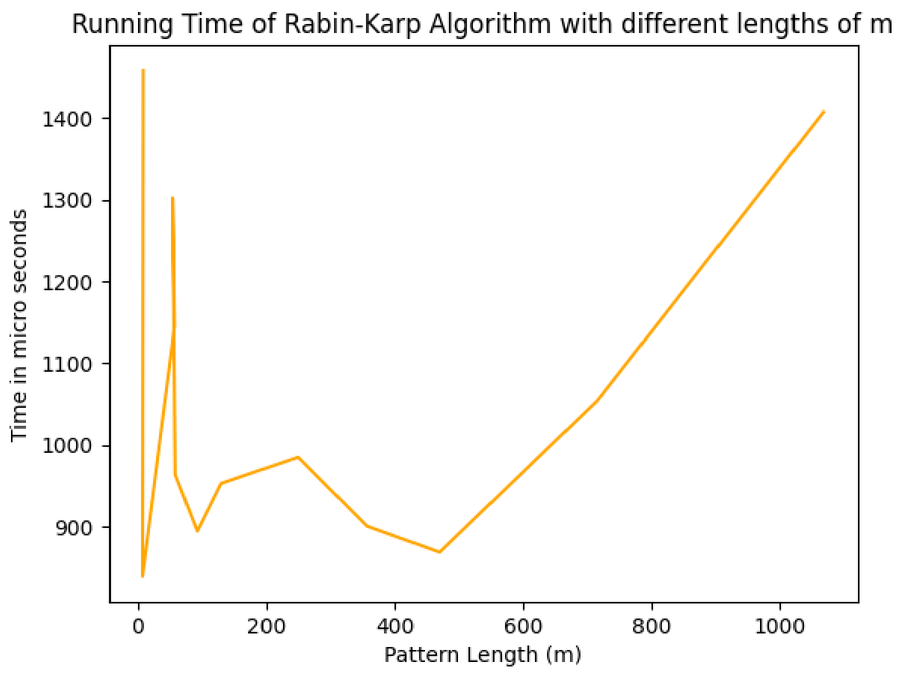
Results

Given a text of length *n* and pattern of length *m*, there are 2 possibilities to test for:

1. Effect of increase in *m* (pattern size) with a constant *n* (text size).
2. Effect of increase in *n* (text size) with a constant *m* (pattern size).

Case 1: Studying the effect of an increase in *m* (pattern size) with a constant *n* (text size)

We implemented a function **testRabinKarp1(*pattern)*** that performed the algorithm on text of fixed length *n* from the dataset for a given pattern of length *m*. To test the effect of pattern size *m*, we created 10 patterns of varying lengths and tested them. We chose an arbitrary text from the dataset which remained the same during the test. We calculated the running time for the KMP algorithm with patterns of varying lengths *m*, while using the same text of length *n*. Here are the results:

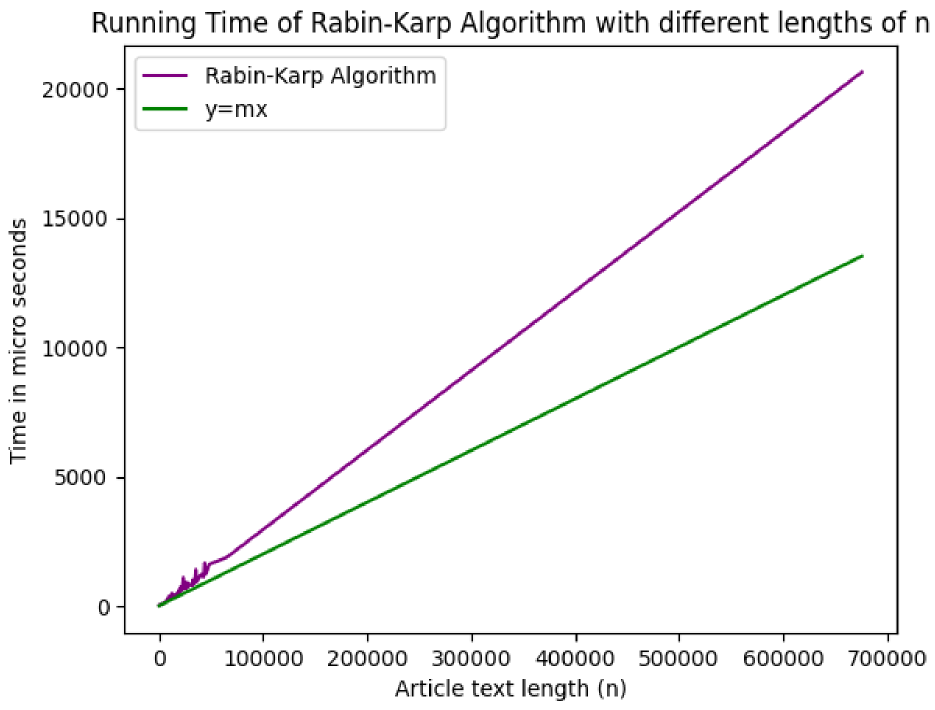


The above plot doesn't suggest a relationship between the pattern length *m* and runtime. However, the runtime seems to be increasing linearly for pattern lengths *m > 500*.

It is likely that the pattern may have been clearer with larger values of *m*. In this case, where the values of text length *n* are in order of hundred thousands, *m* appears to have little effect on the runtime.

Case 2: Studying the effect of an increase in *n* (text size) with a constant *m* (pattern size)

Another function **testRabinKarp2(*pattern, randomIndex*)** was created for this purpose. We selected a pattern of length *m*, which was kept fixed during the 10 trials. During each trial, a random text of length *n* was extracted from the dataset. This was followed by performing the algorithm for the fixed pattern. The results obtained are as follows:



The purple line represents the Rabin-Karp algorithm, and the green line is a simple straight line plotted for reference. The plot suggests a linear relationship between text length *n* and runtime since the runtime increases with increase in *n*.

The runtime for this algorithm is *O(n*m)* in the worst case and *O(n+m)* in the average case. In both cases, since we are still iterating through the whole text (length *n*), it appears to influence the runtime, especially in our case, where $n >> m$. Considering this, the results are consistent with the actual runtime of the algorithm.

Unexpected Cases/Difficulties

- Some of the texts (3%) in the datasets were incomplete and caused errors when patterns of larger lengths were used. This was, however, rare and we tweaked our methods to only choose texts which were complete and were bigger in size, ie., $n > m$.
- Since Python is case sensitive, we had to be careful with the patterns since the algorithm would sometimes report a "no match" if the case of the pattern and text did not match.

References:

1. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

2. https://www.programiz.com/dsa/rabin-karp-algorithm