COSC 320 – 001
Analysis of Algorithms
2022/2023 Winter Term 2

Project Topic Number: 2
Milestone 4

Group Members: Samuel S, Sam G., Jimmy J.

Summary of the Algorithms
Both LSC and KMP algorithms are used for string matching, but they have different approaches and are suited for different use cases.

The LSC algorithm is used to find the longest common subsequence between two strings. It has a time complexity of O(mn), where m and n are the lengths of the strings, and uses dynamic programming to build a matrix of solutions to subproblems. The space complexity is also O(mn) due to the matrix. LSC is useful for comparing two long texts, where exact matching is not required, but rather the degree of similarity between them.

The KMP algorithm is used to find all occurrences of a pattern in a text. It has a time complexity of O(n+m), where n is the length of the text and m is the length of the pattern, and uses a preprocessing step to build a lookup table of the pattern. The space complexity is O(m) due to the table. KMP is useful for comparing short patterns to long texts, where exact matching is required.

The Rabin-Karp algorithm is another algorithm used for string matching, similar to KMP. It has a time complexity of O(nm), where n is the length of the text and m is the length of the pattern, but has an average-case time complexity of O(n+m) due to its use of hash functions. The algorithm uses a rolling hash function to compute the hash value of the current substring and compares it to the hash value of the pattern. If the hash values match, it checks for exact matching of the strings. If not, it moves to the next substring by rolling the hash function

Compared to KMP, Rabin-Karp is faster for very large patterns, as it avoids preprocessing and building lookup tables. However, it can have collisions in hash values, leading to false positives. Additionally, the time complexity can be worse than KMP for some inputs, as it depends on the quality of the hash function.

In terms of algorithmic analysis, KMP has a lower time complexity than Rabin-Karp in the worst-case scenario, making it faster for large inputs. However, Rabin-Karp can be faster in average-case scenarios due to its use of hash functions. LSC, on the other hand, has a higher time complexity than both KMP and Rabin-Karp, but is useful for measuring similarity between long texts.

Therefore, the choice between LSC, KMP, and Rabin-Karp depends on the specific use case and input sizes. If we want to compare a long text to a corpus of documents to detect plagiarism, LSC may be more appropriate due to its ability to handle large inputs and measure similarity. On the other hand, if we want to find all occurrences of a short pattern in a large text and have limited memory, Rabin-Karp may be more efficient due to its average-case time complexity. If we want to find all occurrences of a short pattern in a small text, KMP may be more appropriate due to its lower time complexity in worst-case scenarios.

Dataset. Include the details of the dataset.

https://docs.google.com/document/d/1ITbNNOqqp3W39R5hJJsBu_87kOjblLezIKjNMcGPuoY/edit?usp=sharing

Original Data:
https://ubcca-my.sharepoint.com/personal/fatemeh_fard_ubc_ca/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Ffatemeh%5Ffard%5Fubc%5Fca%2FDocuments%2FCourses%2FCOSC%20320%2D2022%2D2023%20%28Jan%202023%29%2FProject%2DDataset&ga=1

Data after being cleaned
https://drive.google.com/drive/folders/1Z2GeIZ4iYjUoJ_GummzPsRVixuNMAKij?usp=sharing
(the way the program is made for KMP and LCSs the program will load a max of)

The first link is a modified dataset from the second link
basically we put the data into a data frame in python and then we selected the contents out of the data and then we used a very easy to use delimiter ;;; to easily separate the data in java

ImplementationSimilar to Milestone 2 and the implementation of Rabin-Karp, Python is used first to clean up the csv files extracting the main text and save to individual text files.

LCS Implementation:
The Algorithm was implemented in Java. We have the LCSS_main class for this algorithm. The Code contains two methods:

The docs() method takes a string parameter called location, which is the path to a directory containing one or more CSV files. It reads the contents of each file and returns an array of strings. Each element of the array contains a string consisting of all the comments from a file, with each comment separated by three semicolons (";;;"). The docs() method uses a nested for loop to initialize a two-dimensional string array, strings, where each row corresponds to a file and each column corresponds to a comment within that file. It then reads the contents of each file, separates the comments into separate strings, and stores them in strings.

The LCS() method takes two strings ppwd and corp, and returns an integer representing the length of the longest common subsequence between the two strings. It uses dynamic programming to fill in a 2D array (lcs) which stores the length between ppwd and corp at each pair of indices (i,j). This method then makes a recursive call to itself when it comes across matching characters in ppwd and corp. This method returns the length of lcs at the end.

Then we have our main() method, It reads comments using the docs method, preprocesses the query and finally calculates the similarity score between ppwd and each comment, and outputs the highest similarity found using the LCS() method. The similarity score is then divided by the length of the longer string to get a normalized similarity score. The comment with the highest score is printed as the most similar comment.

KMP Implementation:
The Algorithm was implemented in Java as well. The code has two main functions: PREFIX_FUNCTION and KMP_MATCHER

The PREFIX_FUNCTION takes a string ppwd and returns a 2D array containing the prefix function values for ppwd. This is used by the KMP algorithm to avoid unnecessary character comparisons.

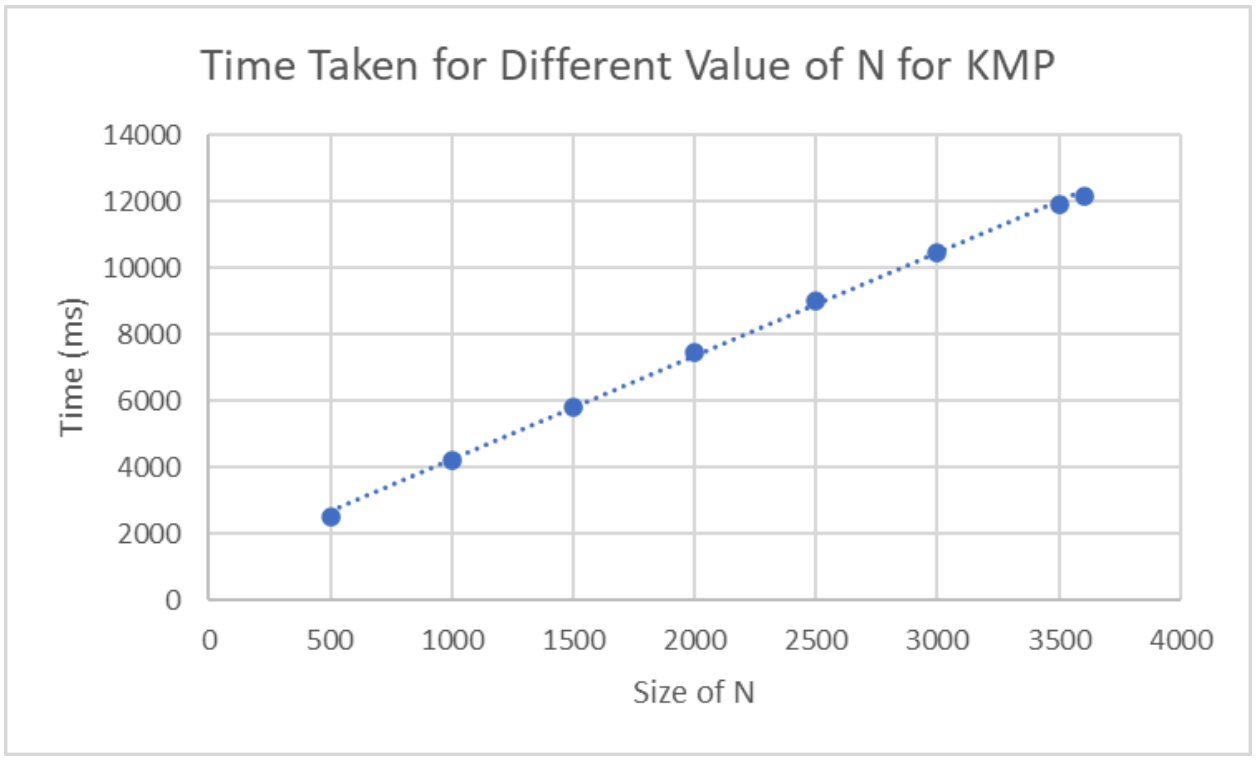
The KMP_MATCHER function takes two arguments: a comment and ppwd, and returns an integer that is the number of matches found in comment. It compares characters of ppwd to those of comment. The function also takes the array e from the previous function as an argument.

The code also has a docs() method which has the same function as in the LCSS algorithm implementation.

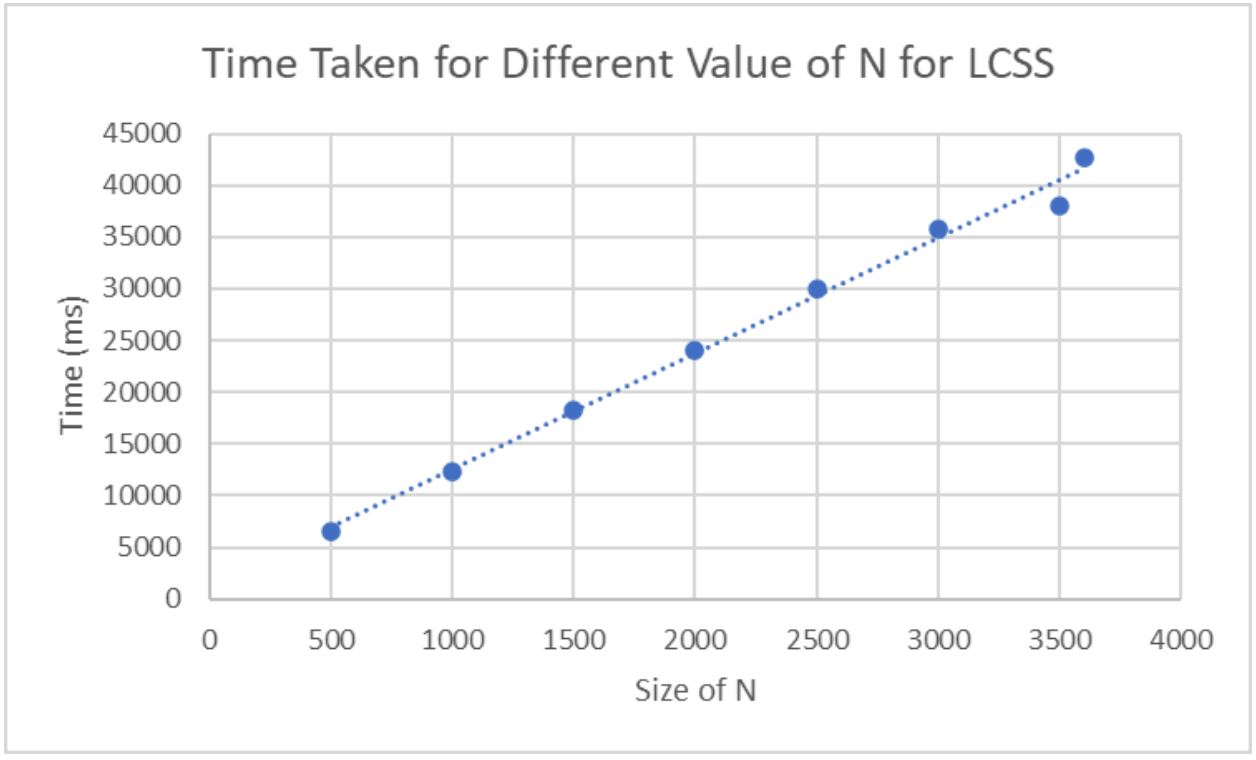
The main method calls the docs method to read in text files, and then iterates through each comment in each file, passing them to the KMP_MATCHER method along with a pattern to match against. It then prints out the total number of matches found in all comments for each file.

RKF Implementation:
RKF from milestone 2 is used nothing was change except some additional code to measure the time

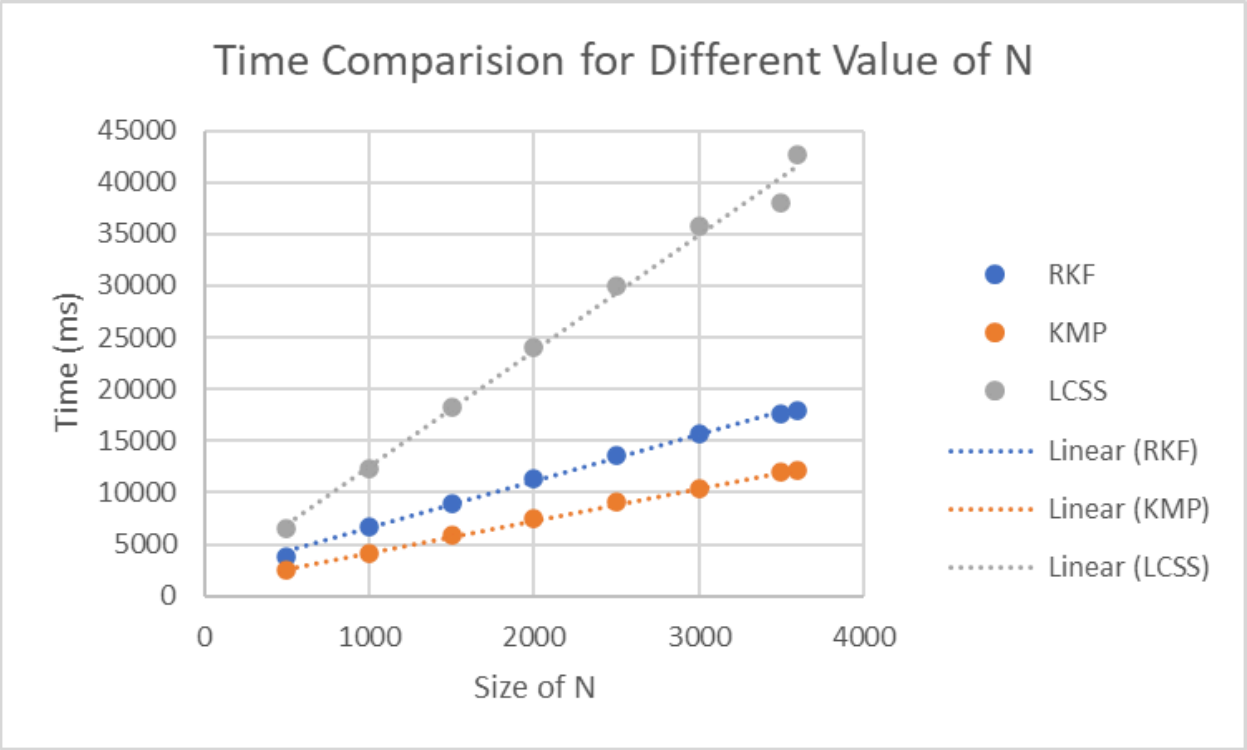
Results
The expected running time and time complexity is O(mn) for LCS algorithm and O(m+n) for the KMP algorithm where n and m are the length of the test input and the number of the documents to compare to, respectively.



The above plot is of the KMP algorithm. The plot shows a dotted line which is the ideal time complexity of the KMP algorithm which is O(m+n). Because the input m is constant, so the output should be linear. As seen in the graph the experiment data does closely matches the ideal value



A similar plot is generated for the LCSS, which has the time complexity of O(mn). Similarly, m is kept constant, so the idea growth is linear. As the graph shows the experimental does match the ideal line.



The performance of all three is compared as well. This plot shows that KMP is the fastest, RKF is the second, and LCSS is the slowest. Data of RKF is from milestone two.

The codes can be found here: https://github.com/DancingUnicorn047/COSC320Milestone4
And the video are here:
https://drive.google.com/file/d/1LBQXOQd7JUBsFeQTXrFAdNxpcS00d2oZ/view?usp=sharing

Unexpected Cases/Difficulties
KMP:

- because we are only counting words towards plagiarism only if there are 3 words in a row we had to add an additional π type function in order to see how many spaces are in each prefix that is also a suffix.

LCSS

- LCS: Because LCS does not look for specific strings this presents a problem for what we defined to be plagiarism, to remedy this particular problem we will redefine plagiarism for this method so a plagiarized document will be one that has the longest common subsequence that is equal to the largest value obtained from taking the max{lcs[i]/(min{length(ppwd), length(corpus[i])}, 25} ($\forall i \in \mathbb{N} \in$ corpus.length) where lcs[i] is defined as the longest common subsequence between the document being checked for plagiarism and the i^{th} document in the corpus and min{length(ppwd),

length(corpus[i]} = the length of the shorter of the 2 documents, this method was taken from https://reader.elsevier.com

both

- in implementation had to make sure that words are still counted the word is at the end of a comment but is in both the comment and document being checked
- tabs, lines within comment and other spaces that were not single spaces could present a problem so they were all removed

Test files:

- it was quite difficult to get the comments to all load into Java, I think this must have been due to poor formatting of the comments (by which I mean people who made comments would put all sorts of spaces and other things that were a pain in the neck to remove)

Task Separation and Responsibilities. Who did what for this milestone? Explicitly mention the name of group members and their responsibilities.

Samuel S: All the code for both the LCS and KMP algorithm. Wrote 100% of the code for the implementation of the Algorithms. As mentioned in Milestone 2, the dataset was modified and converted by Samuel S.

Sam G.: The summary of algorithms in the report, comparison of the 3 algorithms, explanation of how the code was implemented, and Results for KMP.

Jimmy: Modified code of LCS and KMP to read files faster and generated plots, made the video