**COSC 320 – 001**

***Analysis of Algorithms***

2022/2023 Winter Term 2

**Fourth Milestone**

**Project Topic Number: #2**

**String Matching for Plagiarism Detection**

**Group Lead:**

**Youssef Mahmoud**

**Group Members:**

**Esteban Martínez (22717805),**

**Youssef Mahmoud (37624970),**

**Khalid Mahmoud (28842458).**

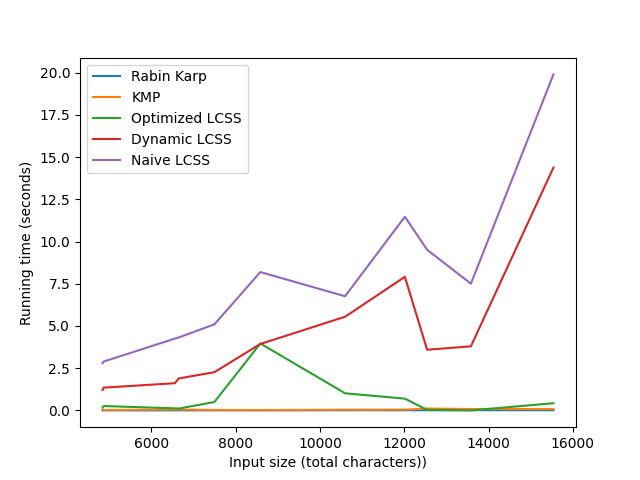
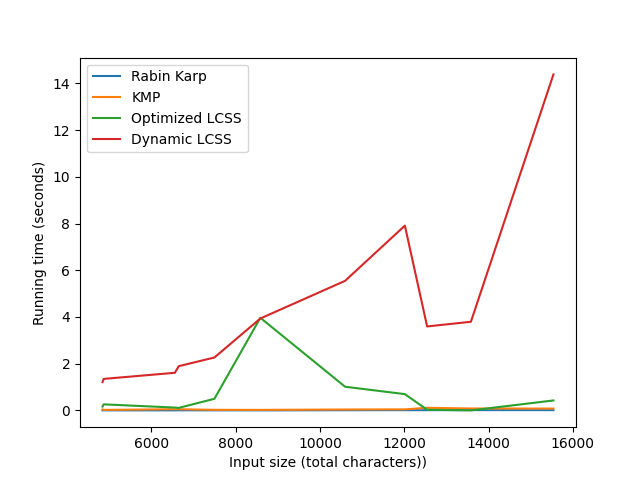
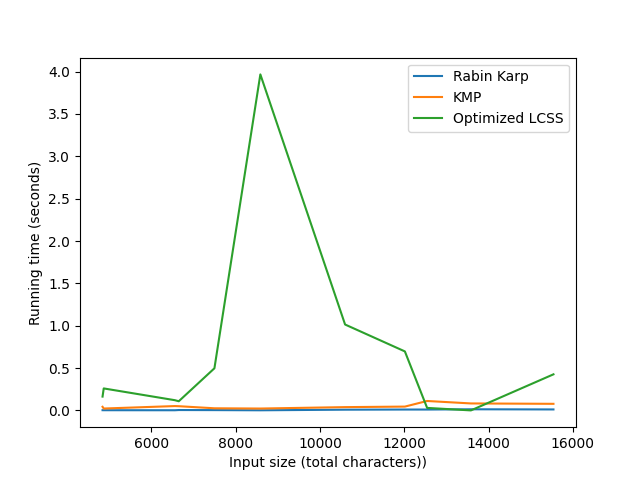
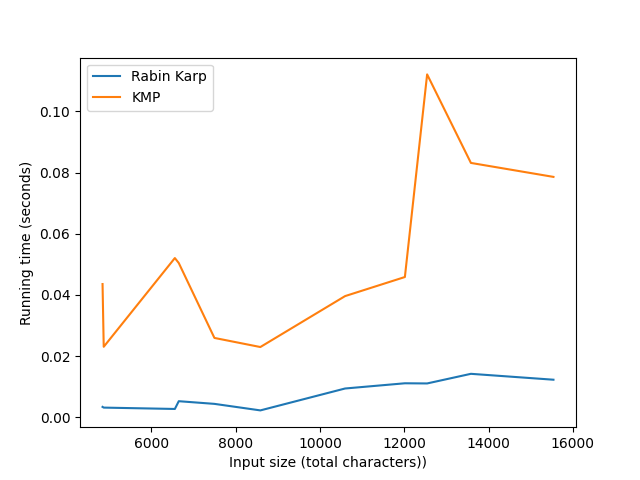
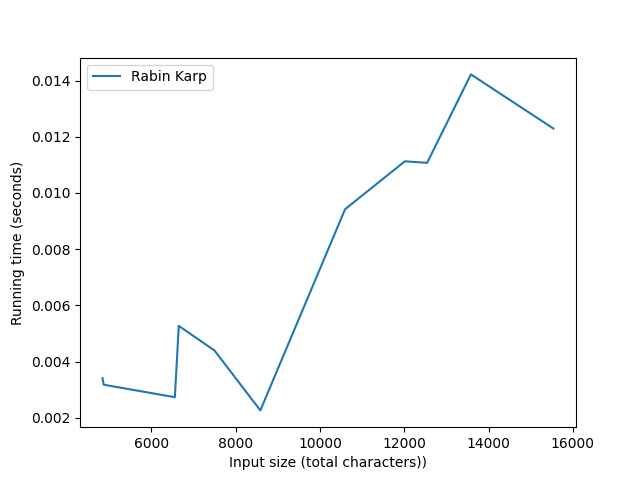
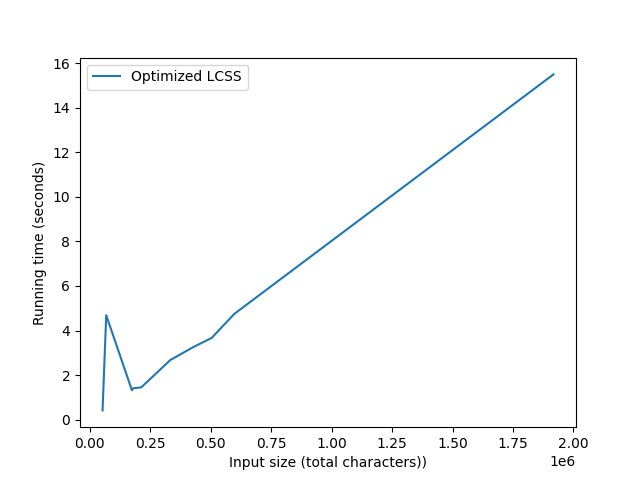
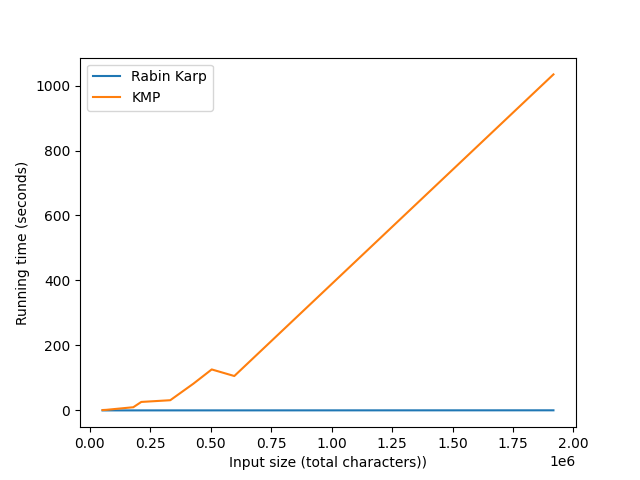
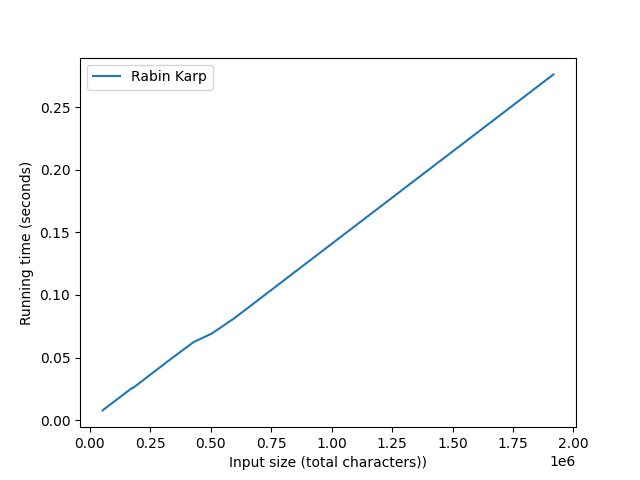
**Abstract:**

For this milestone, we implemented the final algorithm for the project, the Rabin Karp algorithm, and we created the final graphs for our report. As before, we implemented the algorithm in Python. We also expanded our previous testing dataset by adding four more plagiarised documents, as well as tested our algorithms on the expanded dataset sourced from the PAN Plagiarism Corpus 2011[[1]](#footnote-1). We made use of Google Cloud’s Compute Engine to perform the expensive calculations for our large dataset (2 million characters) and produced graphs for our final analysis. Our repo can be found here: <https://github.com/youssefM1999/COSC320-String-Plagiarism-Project>.

**Implementation:**

The Rabin-Karp Algorithm implemented in our project is a dynamic programming approach that uses a few methods to efficiently search for a pattern within a text. It first uses a rolling hash function, which updates the hash values of every substring as it slides across the text. As expected of a dynamic algorithm, the hash values of the previous substrings are used to calculate that of the current substring, along with the addition and subtraction of the values at index 0, and i-1, where i is the length of the substring. This makes the hash recalculation O(1). The algorithm then uses modular arithmetic to handle large hash values that can result from numeric values. For instance, the hash of a very large number would end up being that number. Here, modulating the hash value with a prime number solves this issue by compressing the range. Lastly, the algorithm compares the hash values of each substring to determine if they are a match. In the case of a hash match, it double-checks the match. This allows the algorithm to perform a filtration process before evaluating all values/characters against each other. Finally, the algorithm returns the indexes of all confirmed matches.

**Results:**

****

Overall, the general trend in running times of the algorithms was (generally) as expected.

The Rabin Karp algorithm was by far the fastest, even though it has a time complexity of O(mn), it is generally expected to only make O(n+m) comparisons. Although not as visible in the smaller dataset, this algorithm did exhibit a linear time growth in the graph as would be expected.

The second fastest was the KMP algorithm, which has a time complexity of O(n+m). Their slight difference in time complexity could very well be down to implementation details, such as the fact that our KMP algorithm tests for all possible subsequences in the text, and uses natural language processing to process sentences in the text. Again, the shape of its growth was not as apparent in the small dataset, but was evidently linear in the large dataset.

The third fastest (which was much slower than the previous two) was the memory optimized version of LCSS. This algorithm behaved strangely, and did not have a very defined shape in the small dataset, and looked practically linear in the large dataset. We are not too sure about the usefulness of this particular algorithm in the analysis, and some of the optimizations we made might have significantly changed the way we analyse LCSS in previous milestones.

Finally, the dynamic LCSS was the next fastest, followed by the naive implementation. These algorithms have a complexity of O(mn^2), and demonstrated a markedly quadratic shape even on a small dataset. We believe that this quadratic growth, along with the extreme memory growth of our implementation, made it impossible to test this implementation on larger datasets. This is regrettable, but we find this to be illustrative of the importance of the analysis of growth functions in the design of algorithms, as programs can quickly grow to be computationally infeasible even for extremely capable modern systems given a large enough input (but not one which we wouldn't find in the real world).

**Unexpected Cases/Difficulties:**

Due to the massive size of some of the files, the program can take an enormous amount of time to compute. For this reason, we decided to use a cloud computer using the Google Cloud Compute Engine API (AMD based, 8 core, 32 GB RAM) to do our large calculations.

This allowed us to produce graphs for the KMP and Rabin Karp algorithms with our large dataset, but we still ran into memory issues with our LCSS algorithms. This was because our LCSS implementations used a 2D array, as well as temporary storage for our files in the form of `src\_processed` and `sus\_processed`.

For this reason, we created another implementation, `optimized\_lcss.py` which instead iterates the files line by line to check for longest common subsequences and does not create a processed version of the file. It also does not create a 2D array and instead uses 2 different arrays.

With this new file we were able to produce a graph using our large dataset.

**Task Separation and Responsibilities:**

* Esteban
  + Added four new samples to small dataset
  + Created README documentation for repo
  + Created `main.py` for running algorithms and producing graphs
  + Setup Google Cloud VM for running graphs
  + Results/Unexpected Cases and Difficulties/Abstract
* Khalid
  + Implementation of Rabin Karp algorithm
  + Writeup for implementation
* Youssef
  + Sourcing dataset
  + Created Presentation Video

1. Potthast, Martin, Stein, Benno, Eiselt, Andreas, Barrón-Cedeño, Alberto, & Rosso, Paolo. (2011). PAN Plagiarism Corpus 2011 (PAN-PC-11) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.3250095 [↑](#footnote-ref-1)