String Matching: Plagiarism Detection

SYSTEM DESIGN: (Modules and Interaction)

The four algorithms that we have implemented are:

1. **Naive String Search**: This algorithm slides the pattern over the file text one by one and checks for the suitable match. When match found at particular index a pattern_occured Boolean value is returned and then slides by one to check for sub-sequence matches. This is rather a simple traversal on the given sequence of text or a blob. There is no Preprocessing time and Space complexity. Time Complexity is Θ (nm).

2. **Least Common Subsequence(LCSS)**: Problem of finding the longest subsequence common to all the sequences for a given set of strings. The algorithm we have implemented finds the length of the sequence that is common from the given corpora text(i.e, file). We have dynamic programming to implement this algorithm. Time Complexity is O(nm) and Space Complexity is O(min(n,m)).

3. **KMP**: This string matching algorithm matches a pattern having same sub patterns appearing more than once, this is achieved by avoiding matching of the characters which will match anyway whenever a mismatch is detected. Also pre-processing the pattern to be matched is taken care by pre_processing() in the algorithm we have implemented which returns an auxiliary pattern array which is used to skip characters while matching. The Preprocessing time and Space complexity is Θ (m) and the Time Complexity is Θ (n).

4. **Boyer Moore algorithm**:String matching algorithm which is practically used when pattern length is too long.The algorithm that we have implemented uses “bad character shift rule” which escapes repeating unsuccessful comparisons against a target character. At every step, it slides the pattern by max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern. The Preprocessing time is Θ (m+k), Space complexity is Θ (k) and the Time Complexity is best case is Omega(n/m), Worst Case is O(mn) and O(n) in Average case.

Modules:

The entire program is written in Java. This program contains four modules for each algorithm which is to be implemented. Each module's class instance is created in the main java class "PlagiarismDetector".

BoyerMoore.java, KMP.java, LeastCommonSubsequence.java and NaiveSearch.java are the four modules containing pattern matching function.

Python script file(graph_plotter.py) is written to plot the graphs with input size as x-axis and time as y-axis. This script takes in the output of the string matching java program as it's input and outputs the corresponding graphs depicting the run time of each algorithm against one another.

Input: filename.txt (plagiarized file to be searched against a list of files for patterns or strings in common)

Folder Directory (containing list of files or documents).

Output:

Naive String Search:

txt[] = "AABAACAADAABAABA"
pat[] = "AABA"

Pattern found at index 0.

Least Common Sub Sequence(LCSS):

txt[] = "AABAACAADAABAABA"
pat[] = "AABA"

Pattern found at index 0.

KMP:

txt[] = "AABAACAADAABAABA"
pat[] = "AABA"

KMP Found Pattern at index: 0.

Boyer Moore algorithm:
txt[] = "AABDBAABA"
pat[] = "AABA"

Pattern occurred at shift= 0.

<u>Data Structures</u>(used in our program):

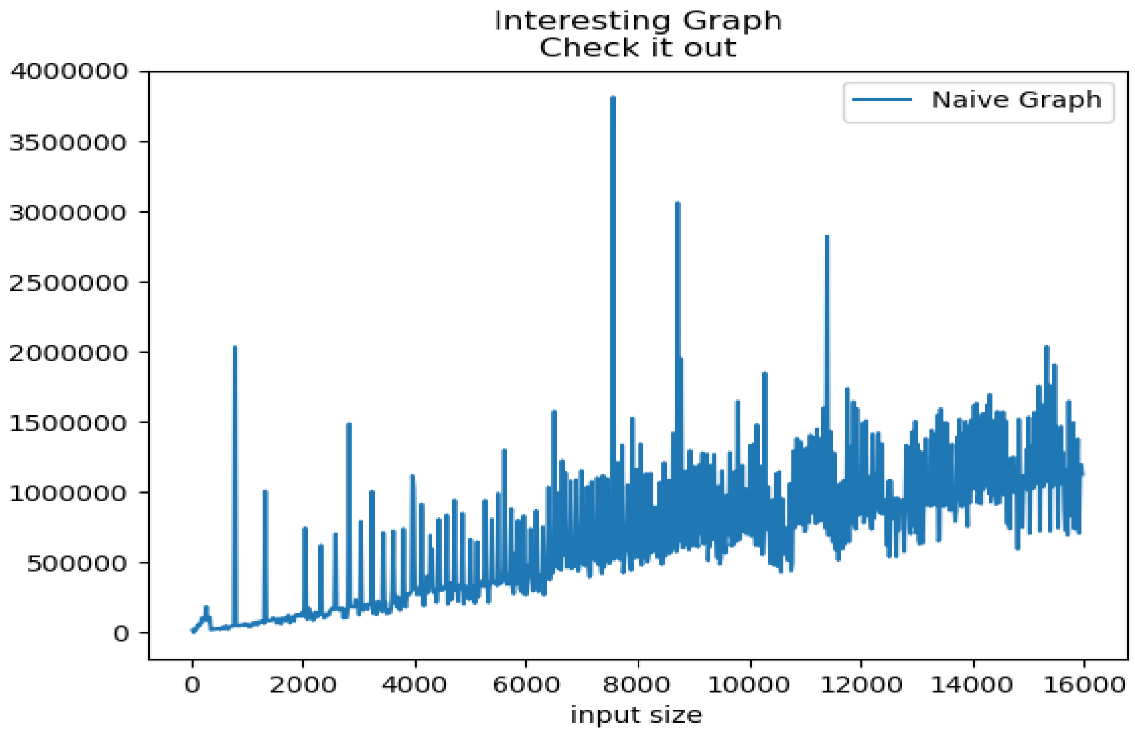
HashMap (key, value)- used to store the pre-processed pattern which tells us the count of characters to be skipped (KMP Algorithm).

StringBuilder a java object used to manipulate the strings (like an ArrayList).

<u>Experiments and results:</u>

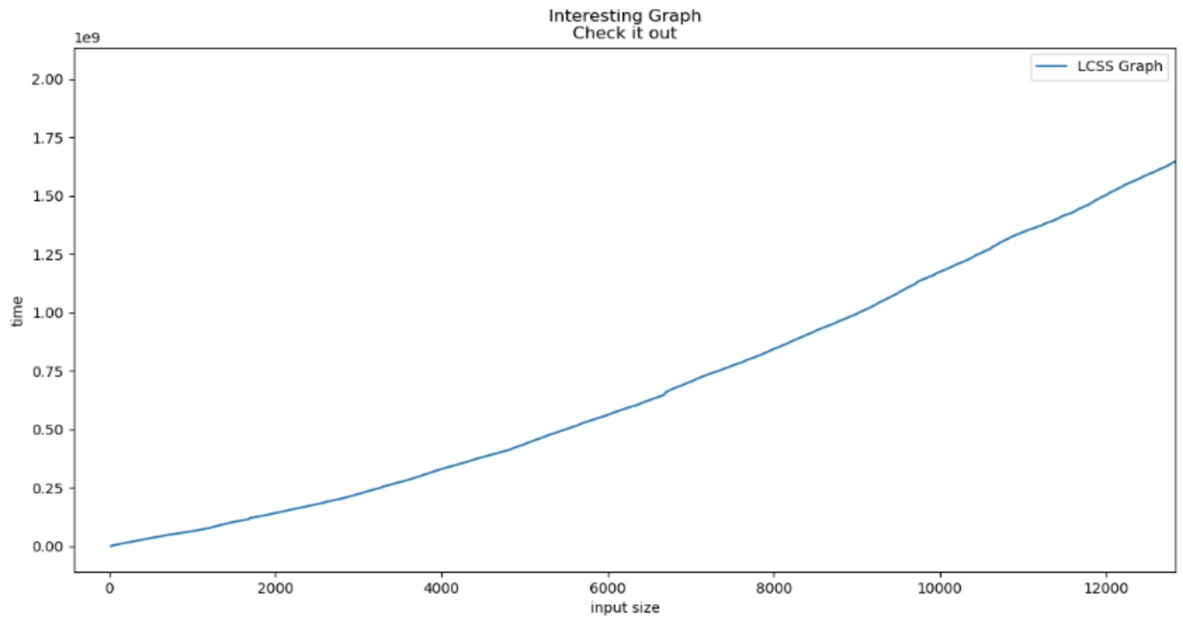
<u>Naïve String Search - Observation:</u>

The Naive Search algorithm slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches. This algorithm takes roughly Θ((n − m + 1)m).



Least Common Sub Sequence(LCSS) - Observation:

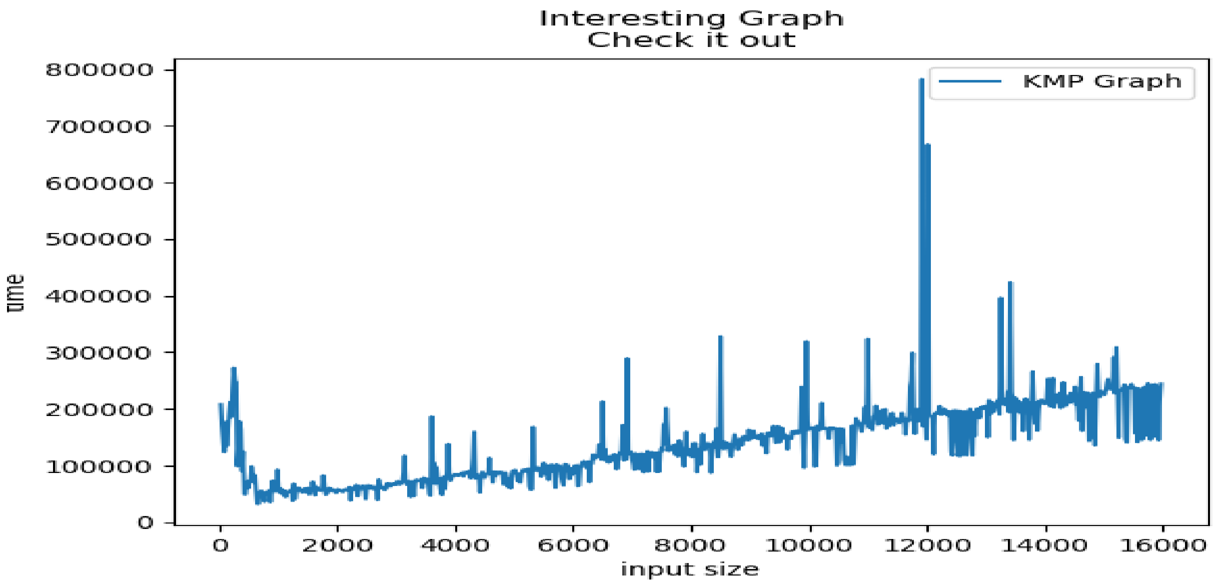
This algorithm takes roughly the same amount of time as O(mn). This graph below shows the time vs input size as exponential growth. It takes more time as it matches character by character.



KMP - Observation:

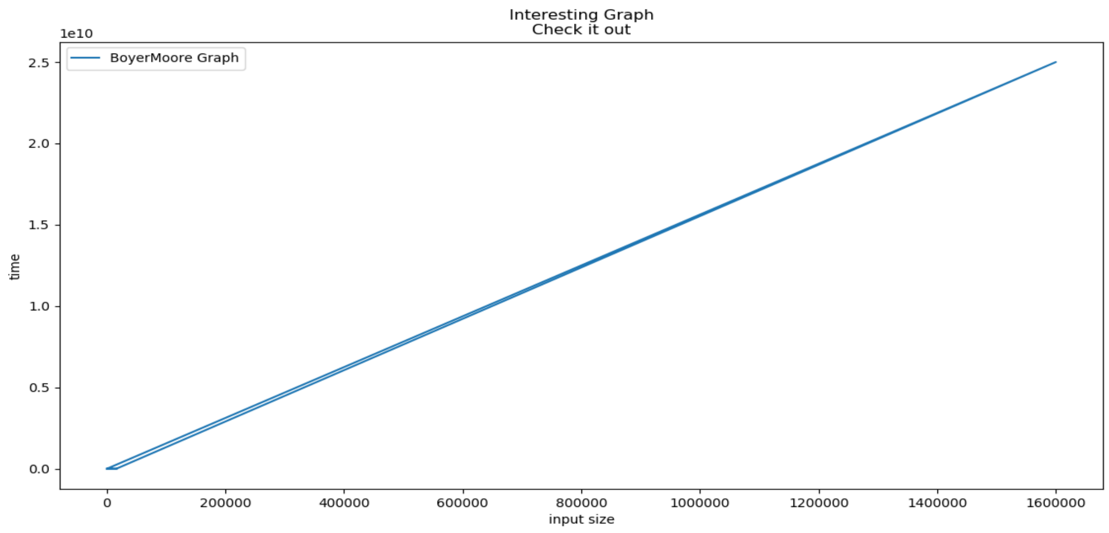
This algorithm needs some time and space for preprocessing. Thus the preprocessing of the pattern can be done in O(m), where m is the length of the pattern, while the search itself needs O(m+n). We can do the preprocessing only once and then perform the search as many times.

We have plotted the results in the graph with input size against the execution time for the running time of the KMP algorithm.



<u>Boyer Moore algorithm - Observation:</u>

Boyer-Moore needs both good-suffix and bad-character shifts in order to speed up searching performance. After a mismatch the maximum of both is considered in order to move the pattern to the right and its performance is plotted below.



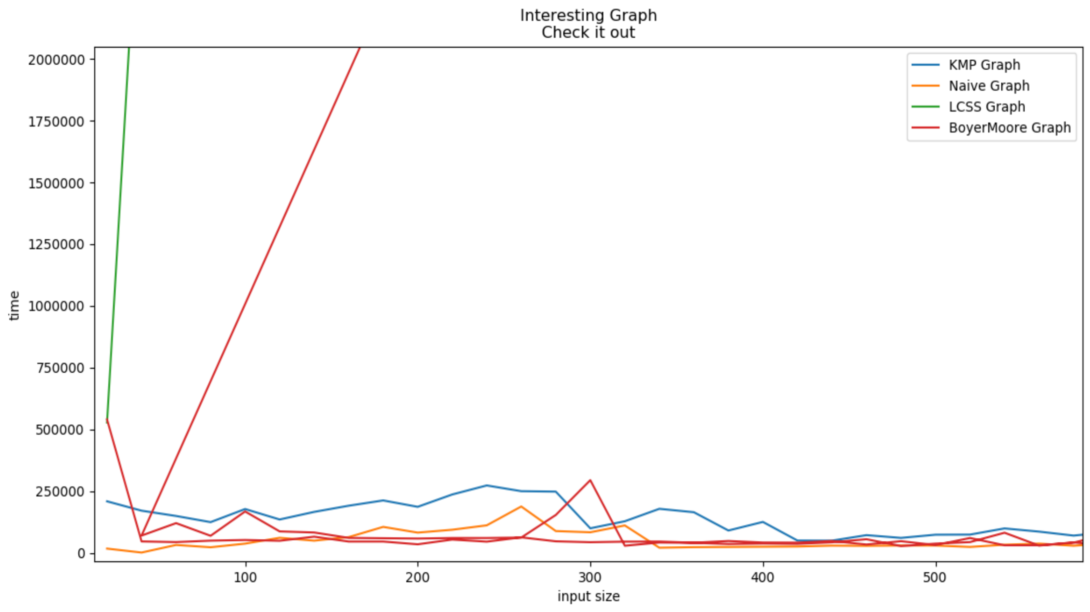
Comparison of all the Algorithms:

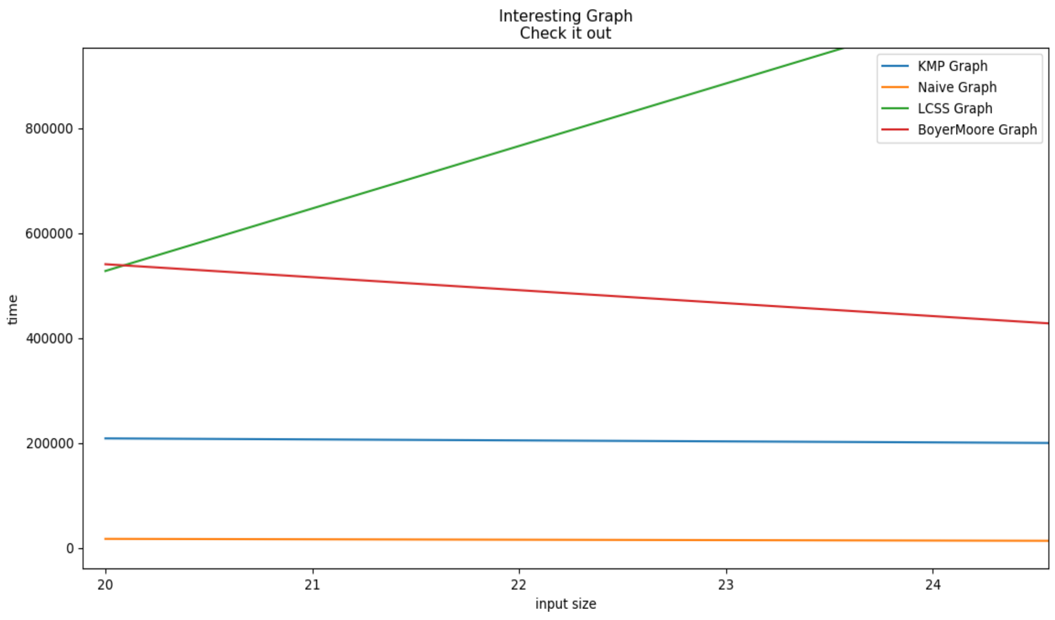
The performance of algorithm depends on two factors :

first -> Input, number of inputs and type of inputs

second -> Methodology of algorithm

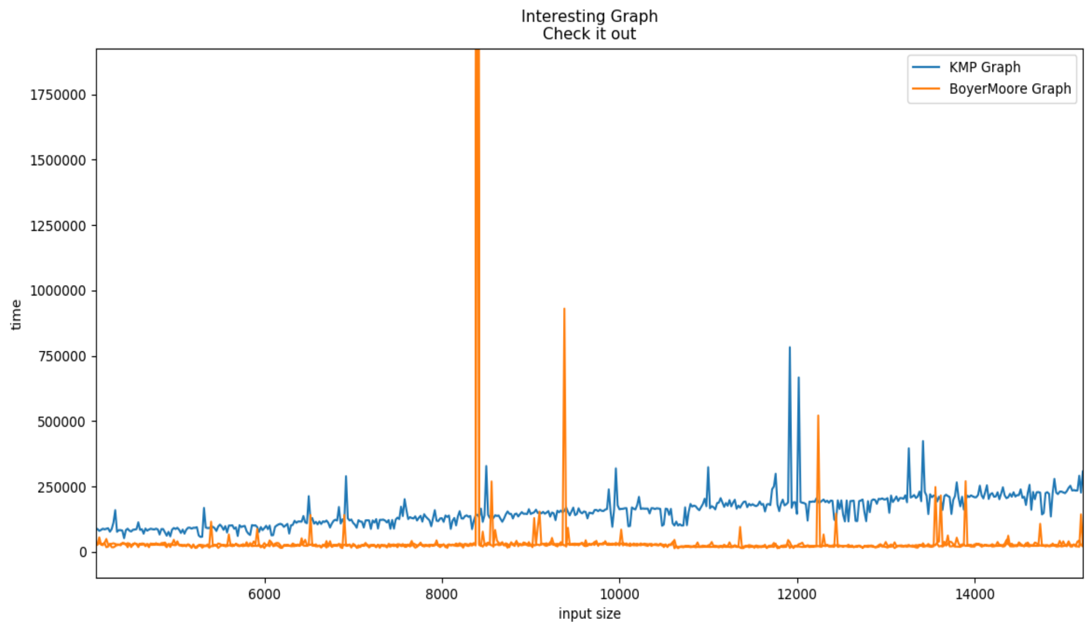
so there may be possible that some variation in performance occur as input changes.





On further zooming the graph, we see that all algorithms perform in accordance with their running time.

“**Something interesting that we noticed during the experiment”** was about the KMP and BoyerMoore algorithm giving the following output where the run time was overlapping:



The above figure(graph) shows execution time vs input size on sequential search using Boyer Moore and KMP algorithms. This explains the performance difference between the two algorithms. It is evident that Boyer Moore performs better to KMP approach.

Both have the worst case time complexity and Its been proven by the graphs shown above.

The Conclusion is Naive Search and LCSS are comparatively slower to KMP and Boyer Moore. The difference that renders KMP and Boyer Moore faster is the movement in the Sliding window. KMP has a different mechanism so as BoyerMoore.