Project 1: String matching – Simple Plagiarism detection using string matching algorithms

The definition for Plagiarism is as followed based on the English dictionary.
Plagiarism:

Noun
an act or instance of using or closely imitating the language and thoughts of another author without authorization and the representation of that author's work as one's own, as by not crediting the original author.

Detecting Plagiarism:

Several algorithms have been developed and are used in order to determine if someone is plagiarizing another's works. In this project we will be going over three of the algorithms and they are **Knuth-Morris-Pratt algorithm (KMP)**, **Longest Common SubString algorithm(LCSS)**, and **Rabin-Karp for Multi-Pattern Matching (RK)** algorithm. These algorithms will use different methodologies in determining plagiarism. For both **KMP** and **RK** will both uses a methodology that revolves around string pattern matching between a **pattern**, ie a sentence, to be matched against a **text or in this case a text file**. While the LCSS while compare paragraphs from a text file against paragraphs from other text files.

Knuth-Morris-Pratt Matching Algorithm (KMP)

Description:

For KMP we assume we are given a long string or text and a short pattern. Using KMP algorithm we compare the pattern against the text or P to T if u will. And example can be seen below.

Text: a a a b b c a b c a b a
Pattern: a a b b c

So one would think that Naïve Method might work for this, since as you can see once it compares the first element in pattern to text, it will the compare the 2nd element in the pattern to the text, and once it finds out it's not a match it shifts. However, the issues that this Naïve method has is that it will only work for small amounts of data due to the fact that this type of comparing process runs at time of O(nm), were **m is pattern length** and **n is text length**. KMP however runs takes O(m + n) or just O(n) if we disregard the LPS part of the code as seen in the algorithm below.

Algorithm:

```
KMP(pattern, text)
    M = pattern length;  N = text length();
    lps[] = LPS(pattern, m, lps);   //  an array that stores the prefix-suffix values for pattern
    j = 0;  // index for pattern characters
    i = 0;  // index for text chataxters

    while (i < N)
      if (pattern[j] == text[i])
        j++; i++;
      if (j == M) //Match found
        Output(j-m);
        j = lps[j-1]; // Look for next match
     else if (i < N && pattern[j) != text[i]) // mismatch after j matches
        if (j != 0)  // Do not match lps[0..lps[j-1]] characters, they will match anyway
          j = lps[j-1];
        else
          i = i+1;
```

```
LPS (pattern, M, lps[])
   int len = 0; // length of the previous longest prefix suffix
   int i = 1;  lps[0] = 0;  // lps[0] is always 0

   while (i < M)  // the loop calculates lps[i] for i = 1 to M-1
     if (pattern[i] == pattern.[len])
        len++;  lps[i] = len; i++;
      else
       if (len != 0)
          len = lps[len-1];
      else
           lps[i] = len;
            i++;
```

Example: An example of using the above algorithm, we will us the initial values seen in the description, plus the LPS.

Text: a a a b b c a b c a b a
Pattern: **a a** b b c
LPS: 0 1 0 1 0

So an example on how KMP works with the given examples would be, for the first iteration we have found matches for **a a**, but a mismatch is found so the values are shifted over, ie incrementing i and j, and nothing is changed for LPS.
i = 3 and j =3

Text: **a a** a b b c a b c a b a
Pattern: **a a** b b c

The pattern continues to shift over till a match is found and once a match is found the LPS is updated.
I = 5, j =5, so since j = m we udated LPS to LPS[j -1] = LPS[4] = 4
Text: **a a a b b c** a b c a b a
Pattern: **a a b b c**
LPS: 0 1 0 1 4

We continue to shift the pattern over once more and as we can see a mismatch does occur but we see that text[i] and pattern [j] do NOT match and j > 0, so we only updated the LPS so j = lps[j-1] = lps[0] = 0.
I = 6, j = 1
Text: **a a a b b c a** b c a b a
Pattern: **a a** b b **c**
LPS: 0 1 0 1 4

This process continues till the pattern reaches the end of the text and returns the matching positions

Time Complexity:

Since this algorithm is reliant on the **Length of the Text (N)** and **the Length of the Pattern (M)**, the amount of time it will go through the loops of KMP and Compute_Longest_Prefix_suffix will come to O(m + n), because the while loop for KMP is goes to O(n) and the Longest Prefix Suffix generation table takes O(m), and since these loops are ran in sequence the algorithm's run time O(m+ n), however if we disregard the Longest Prefix Sufix table then the run time will on be O(n).

Code:

The input is: Corpora of source files and text files.
The output is: file name from which the document was plagiarized and percent that is plagiarized.

Longest Common SubString algorithm(LCSS)

Description:

Unlike KMP, Naïve, or RK algorithms LCSS doesn't use a text and a pattern, it instead takes 2 long texts and find the longest common substring between them. This is done by building out a table called Longest Common Suffix or LCS. This table will keep track of when a match is found between each text and increment.

Algorithm:

```
LCSS (text1[], text2[], int m = length of text1, int n = length of text2)
    LCS [][] = new int[m + 1][n + 1];    // Create a table to store lengths of longest common suffixes of substrings
    result = 0;  // To store length of the longest common substring

    for (int i = 0; i <= m; i++)  // Build the LCS[m+1][n+1] in bottom up fashion
       for (int j = 0; j <= n; j++)
          if (i == 0 || j == 0)
             LCS[i][j] = 0;
          else if (text1[i - 1] == text2[j - 1])
              LCS[i][j] = LCS[i - 1][j - 1] + 1;
             result =max valve of between result and LCS[i][j];
          else
             LCS [i][j] = 0;
   return result;
```

Example:

Using the above algorithm, we compute and example give values of text1 = abcdxyz, text2 = xyzabcd, which will generate a LCS table as seen below;

		a	b	c	d	x	y	z
	0	0	0	0	0	0	0	0
x	0	0	0	0	0	1	0	0
y	0	0	0	0	0	0	2	0
z	0	0	0	0	0	0	0	3
a	0	1	0	0	0	0	0	0
b	0	0	2	0	0	0	0	0
c	0	0	0	3	0	0	0	0
d	0	0	0	0	4	0	0	0

This table will find the max value, 4 in this case and return it. Thus showing that the Longest Common SubString Length is 4, or abcd.

Time Complexity:

The run time for this is O(m*n), the reason being is that each and every value from text1, ie m and text2, ie n, is looked over and check and since this is in a nested loop it comes m*n or O(m*n)

Code:

The input is: Corpora of source files and text files.

The output is: file name from which the document was plagiarized and percent that is plagiarized.

Rabin-Karp for Multi-Pattern Matching (RK)

Description: Rabin-Karp (RK), is similar to both naïve method and KMP method of string matching, in that it takes both a large text and a short pattern to match against. However what separates them form each is that RK uses hashing in order to find the matching values.

This is achieve by calculating the hash for each patterns and then, we calculate the M-character, M being length of pattern, subsequence of the text's hash values and compare. If a match is found, ie the hash values are equal, then the algorithm will compare the pattern and the M-character sequence. In this way there is only one comparison per text sequence and character matching is to be done only when hash values match. If the has values are un equal the algorithm will recalculate the M-character subsequence of the text's hash values and re-compare, as seen in the algorithm below

Algorithm:

```
Rabin-Karp (pattern, text, prime_number, d) //d is number of characters in alphabet or ascii code. So its 256
  {
    M = pattern length; N = text length;
     i, j; //Indexes for text and pattern (I for text, j for pattern).
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    for (i = 0; i < M-1; i++)
       h = (h*d)% prime_number;  // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M; i++) // First iteration caluculation
       p = (d*p + pattern[i]) % prime_number; // Calculate the hash value of pattern
       t = (d*t + text[i]) % prime_number;  // Calculate the hash value of window of text

    for (i = 0; i <= N - M; i++)// Slide the pattern over text one by one
    {    if ( p == t)  //Check the has values of text to pattern if match the check each character 1 by 1
          for (j = 0; j < M; j++) // Loop to Check for characters one by one
            {if (text[i+j] != pattern[j])   break; }

          if (j == M) Output(Pattern found at index " + i);
       if ( i < N-M )
          t = (d*(t – text[i]*h) + text[i+M])% prime_number;  //Calculate hash value for next window of text:
           if (t < 0) // If values is negative flip it to positive
             t = (t + prime_number);
    }
```

Example:

So using the Algorithm above we will compute the example given below based on the given variables.
Text = ABDCB, Pattern = DC, prime_number =11, d = 256, M = 2, N = 5
h = (1*256)%11 = 3
So for first iteration we have, NOTE 68 is ASCIII FOR D and 65 is ASCII FOR A,
p = (256*0 + 68)% 11 = 2, t = (256*0 + 65)%11 = 10
The for the 2nd and final iteration since we only iterate to M or 2 we get.
p = (256*2 + 67)% 11 = 7, t = (256*10 + 66)%11 = 8,
Giving us p = 7 and t = 8
Now we compare which it isn't., so we shift the text, from AB to BD and recalculate
t = (256* (8 – 65*3) + 68) % 11 to get -9 and since this a negative value we add the prime_number to get 2
we check again and see that no it is so we shift the text, from BD to DC and recalculate
t = (256* (8 – 66*3) + 67) % 11 to get -4 and since this a negative value we add the prime_number to get 7
Now we check we find that the value is same, we now check the each character in text to pattern one by one and output based on were pattern is found. It will then repeat the process showing that only one match has been found.

Time Complexity:

The time complexity: the best case and worst case running time of Rabin-Karp algorithm is O(n+m) and O(nm), based on N = Length of Text and M = Length of Pattern. The reason for the worst case is if the text and pattern are more or less the same, IE Text = DDDDDDDDDDDD, Pattern = DDDD, then it will have the same has hash and continue to cause it to constantly check each character between text and pattern once a match has been found.

Code:

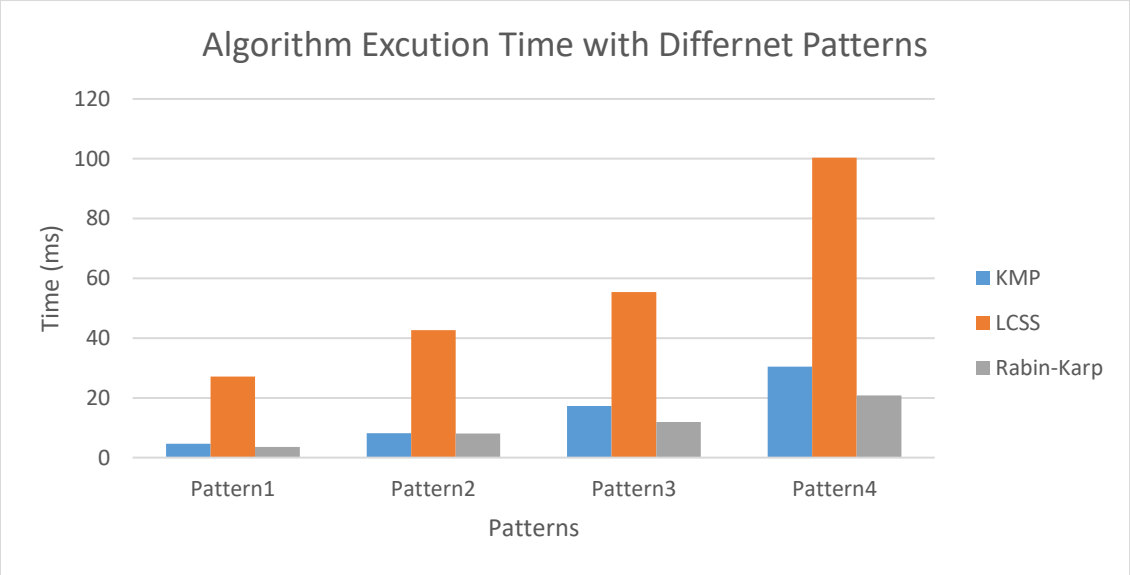
The input data: the pattern file and the original text file
The output data: file name from which the document was plagiarized and percent that is plagiarized.

Graph

The graphs seen below are execution times for the 3 algorithms based around a text file that is to be checked for plagiarism, against three other files or pattern files. This specific text that is to be check for plagiarism has 2291 characters. The 4 "pattern" files are label pattern1, pattern2, pattern3 and pattern4. Naming behind each is based on the increasing order of the number of characters contained in each file, pattern1 has the least amount of characters while pattern4 has the most. The amount in each pattern file is, based on just the number of characters, is 657 for pattern5, 1358 for pattern10, 2291 for pattern3, and 4787 for pattern4.

With regards to the matching pattern1 around 20% for all algorithms, pattern2 matches around 50% and both patterns3 and 4 match 100%.

As it can be seen based on the data below, Rabin-Karp is the faster with KMP coming in second, and finally LCSS taking the longest. This is understandable seeing as how LCSS algorithm compares vast amounts of data at a given time while the



	Pattern1	Pattern2	Pattern3	Pattern4
KMP	4.6	8.2	17.3	30.4
LCSS	27.1	42.7	55.4	100.4
Rabin-Karp	3.6	8.04	11.9	20.8