**COSC 320 – 001**

***Analysis of Algorithms***

2022/2023 Winter Term 2

# Project Topic Number: #2

# Milestone 1

**Group Members: Sam Garg, Samuel Street, Jimmy Ji**

**Abstract**
In this milestone, we formulate the problem in specific mathematical terms, then we wrote a pseudo-code version of an algorithm (RKA) that will achieve the goal outlines in the problem formulation. We then analyzed the algorithm for its time complexity and proved the correctness of the algorithm (ie our algorithm will always produce the desired output). We also considered some unexpected cases and divided up what needed to be done.

**Problem Formulation**
We are given a corpus of documents docs = {d1, d2, d3, …, dn} with a length of documents len_1, len_2, len_3, …, len_n where each document di has length len_i and a potentially plagiarized document A of length len as inputs, the plagiarism detection algorithm should output the documents (docs) from with document A was plagiarized X = {x1, x2, x3, …, xn}. This can be represented as:

PlagiarismDetector(docs, A)
Input: docs={d1, d2, d3, …,dn}, A
Output: X = {x1, x2, x3, …,xn}  xi is in docs, X is plagiarized from xi

Such that for each document di  present in X, there exists a pattern of 3 or more consecutive words in A that match a pattern of 3 or more consecutive words in document di with at least 24% consecutive words matched of the word length of di

**Pseudo-code**

```
PFIH: possible future improvement here
arr = input from the data source
arr2 = [0 for i in arr.length]
barr = Array //for potentially plagiarized document each word = element in array
            //we will develop code to insert random pieces of text from corpus into this test document
count = 0 //number of plagiarized words found
perc = 0 //%of words plagiarized
Word(string, int):
        Word.contents = “string”
        Word.value = Hash(String) //we will use a hash library
for i in arr.length:                                                    //implementing a take on RKA
        for word in arr[i].length:
                arr[i][j] = Word(arr[i][j])
for i in barr.length:
        arr[i] = Word[(arr[i])
pb2 = 2  //index for rolling hash functions going through document being checked
pa2 = 2 //indexes for rolling hash function going through given documents
pc = 0
b1 = arr[p2-2]; b2 = arr[p2-1]; b3 = arr[p2]
s1 = b1.value+b2.value+b3.value
while(pb2<=barr.length-1)
        for i in arr.length:
                a1 = arr[j][pa2-1]; a2 = arr[j][pa2-2]; a3 = arr[j][p2]
                s2 = a1.value+a2.value+a3.value
                for word in arr[i].length-2:
```

```
                    if(s1 ==s2):
                        if(a1.string.length = b1.string.length and a2.string.length = b2.string.length and
a3.string.length = b3.string.length)
                                    if(check all letters in each string and if they match)
                                        if pc = 0                //PFIH
                                            pc = 3;
                                        else:
                                            pc = pc+1
                                        pb2 = pb2+1
                                        s1 = s1 - b1.value
                                        b1 = b2; b2 = b3; b3 = arr[p2]
                                        s1 = s1 + b3.value
                                        arr[i] = 1
                                        break both for loops
                    if(pc >= 3)
                        count = count + pc
                        pc = 0
                    pb2 = pb2+1
                    s1 = s1 - b1.value
                    b1 = b2; b2 = b3; b3 = arr[p2]
                    s1 = s1 + b3.value
count = count + pc
perc = count/barr.length
if perc > 24%
        print(perc "% similar, high chance of plagiarism")
else:
        print(perc "% similar, low chance of plagiarism")
print("these are the document where similarities were found")
for i in arr2.length
        if arr2[i] = 1
        print arr[i]
```

**Algorithm Analysis**

Our algorithm is a slightly modified version of the Rabin–Karp algorithm. Because our standard of plagiarism is the number of words repeated, the input would have to be converted from a string which an array of characters to an array of words. This is done by going through the whole string and checking for words and saving them to an array. So, given an input of string at size n, the documents to compare to at size m. The total time needed would be a constant time multiplied by the total length, resulting in a total time of

$$T(n, m) = c(n + m)$$

Therefore, the time complexity is

$$O(n + m)$$

The result is two arrays of size $c_1 n$ and $c_2 m$, where $c_1$ and $c_2$ are constants that depend on the average word length, these arrays would be hashed. The average hash time complexity is $O(1)$ or constant c, so the average time would be

$$T(n, m) = c(c_1 n + c_2 m)$$

Which results in a time complexity of

$$O(n + m)$$

The next step is to compare the arrays, which is done by a nested loop, the outer loop goes through the entire document. The inner loop goes through the whole length of the input document. At each iteration of the loop, the hash value of the input and the documents are compared. Each comparison should take a constant time $c$. If the same value is reached, another for loop is run to determine if the next three or more words are identical. If so, the number would be recorded, and the pointer for the inner loop would be moved to the end of the repeated words. This is done so the words that have already been checked would not be rechecked, which means the number of iterations in the inner loop is always the same as n. The number of repeats is then calculated to determine plagiarism, the operation should also be constant($c_3$), so the total time is

$$T(n, m) = (c_2(m - 2)) * (c_1(n - 2)) * c + c_3$$

Because the standard is three or more repeated words, the last two words for both the input string and the documents can be skipped, hence the -2. This results in the time complexity of

$$O(mn)$$

Therefore, the average time complexity of the entire algorithm is

$$O(n + m)+O(n + m) + O(mn)$$

Taking the most significant growth which simplifies to

$$O(mn)$$

**Proof of correctness**

Defining correctness and assumptions used.

**Assumption 1:** using synonyms for words in document will not be counted towards plagiarism

**defining correctness:** The algorithm is correct if as the algorithm goes through array arr, the subarray containing all of the word checked thus far has had all of the words in sequences of length ≥ 3 in the potentially plagiarized document that match sequences of equal length in the copus of document provided have been counted

**Proof via loop invariants:**

**iteration:** One full pass in the while loop (which includes going through both for loops)

**Initialization**: It is true prior to the first iteration as no words have been checked and and the count is at zero, thus all sequences of ≥3 in both the document being checked and the documents being looked through have been counted

**Maintenance:** If it is true prior to an iteration then it is true after the iteration: proof.

If the loop is true prior to the iteration then all of the potentially plagiarized words (those in sequences of length ≥3 that appear in the original document and the between the documents being scanned) have been counted.

**Case 1:** If the last 3 words were found in another document then they have been counted and then three words being checked in the potentially plagiarized document would have been moved so that the words being checked would be of the for [“matching word2”, “matching word3”, “unknown4”].

**Case 1a):** if the string is found in one of the documents then [“matching word2”, “matching word3”, “unknown4”] -> [“matching word2”, “matching word3”, “matching word4”] and pc = pc+1 and matching words in sequences of length ≥3 have been in either the count or pc variable, and then the next set of words is checked ie [“matching word3”, “matching word4”, “unknown5”] and maintenance is maintained.

**Case 1b):** If the sequence is not found then [“matching word2”, “matching word3”, “unknown4”] -> [“matching word2”, “matching word3”, “not matching 4”] and count = pc + count and the pc = 0 meaning that the overall count is not changed and the count must then still be valid and so maintenance is maintained

**Case 2:** If the last 3 words in the document were not counted and they are of the form, [“not matching”, “matching word2”, “matching word3”]. Then as the iteration starts the words being looked at would shift to [“matching word2”, “matching word3”, “unknown4”],

**Case 2a:** if the phrase was found in a supporting document then [“matching word2”, “matching word3”, “unknown4”] ->[“matching word2”, “matching word3”, “matching word4”] and then because pc = 0, pc = 3 and all of the word are then counted maintaining the count and so maintenance holds.

**Case 2b**: if the phrase was not found then the count both remain unchanged and maintenance is maintained all other cases are variations on case 2b and thus maintenance is always maintained

**Termination:** When the loop terminates all of the words in sequences of ≥3 that are also in one of the corpus documents have been counted and placed in the count variable. Proof: at the end of the last iteration as shown in maintenance all of the potentially plagiarized words in the subarray consisting of all the checked words in arr will have been counted. Except that this time the subarray will consist of the whole array arr and so this means that all of the potentially plagiarized words in the document being checked will have been counted and at the end of the loop pc is added to count so that all of the counted words are in one variable. This means that at the end of the while loop, all words have been counted and the the program is correct

**Unexpected Cases/Difficulties**

RSA is used (or at least the version found online) to find a single keyword and with the use of many keywords as is the case here the time complexity of the algorithm will be quite high we will have to find a way to lower the time complexity in the future. Also extended the idea of RSA so that rather than just letters being given values all of the words are pre-processed and given a value allowing the RSA method to be used on whole words at a time which is quite relatively speaking. One other potential thing which could be done is removing all words that are extremely common, greatly lowering the number of keywords being looked through. This would require that we change our qualifications of plagiarism, but only slightly. Also as of yet we have only invested time into the RSA algorithm as it seems to have a lot of potential.

One of the other difficulties we encountered was handling synonyms. The algorithm needs to be able to detect plagiarism even when the plagiarized text uses synonyms for the words in the original text. This can be solved by using a thesaurus or a word embedding model to identify synonyms. This would increase the complexity of the program/algorithm since it involves a word embedding model.

**Task Separation and Responsibilities.**

Samuel Street: Pseudo Code, discussing problem, proof of correctness, unexpected difficulties discussion, task separation discussion

Sam Garg: Formulating the problem, discussing problem, unexpected difficulties, task separation discussion

Jimmy Ji: Time complexity calculation, discussing problem, analyze algorithm, task separation discussion, abstract