Final Project

Minerva University

CS110: Problem Solving with Data Structures and Algorithms

Professor Richard

Uladzislau Andreichuk

Contents

Counting Blooming Filters

Counting Bloom Filters (CBFs) is a probabilistic data structure that is commonly used as an indexing technique for efficient approximate set membership queries. They are an extension of the traditional Bloom Filters that can count the frequency of elements in the set and support deletion operation.

The underlying data structure of a CBF is an array of m bits initialized to zero. It also has k independent hash functions that map elements to the m bits. Each hash function produces an index in the range [0, m-1] uniformly at random.

When an element is added to the CBF, the k hash functions are applied to the element and produce a bit index, and then the corresponding bits in the array are incremented by one. This means that each time an element is added to the CBF, it increments the corresponding bits in the array. The opposite is true for deletion - the k hash functions are applied to the element, and the corresponding bits in the array are decreased by one.

When querying if an element is present in the CBF, the k hash functions are applied to the element, and the corresponding bits in the array are checked. If at least one bit is 0, the element is considered to be not in the CBF with 100% certainty. It means that there is no possibility of a false negative because if a certain element was inserted in the CBF, the corresponding CBF bits would definitely not be 0. If all bits are non-zero, the element is considered present in the CBF, although there is a possibility of false positive, that is, the element is considered to be present in the CBF although it has never been actually inserted. This can happen because the hash function

can map different elements into the same CBF bits. For example, if we insert an element x into a CBF, bits 2 and 5 are incremented by 1, which makes them non-zero. We also insert y into bits 4 and 6 after hashing. No other elements are in the CBF. We then query if element z is in the CBF. It so happens that the hash function map z onto bits 4 and 5. We check if bits 4 and 5 are zeros and find that both are non-zero. The algorithm concludes that z is present in the CBF, although, in reality, it is not true since z has never been inserted, and its corresponding bits are non-zero just because elements x and y were mapped onto those bits too.

Given that different elements can be mapped onto the same bits and cause false positive results, the bigger the number of elements to be inserted, *n*, the bigger the chance of getting a false positive result. In other words, False Positive Rate (*FPR*) gets bigger.

According to Bloom and Burton, 1970, *FPR* can be defined using the following formula, assuming a hash function selects each bit in the CBF with equally likely and all hash functions are independent of each other:

$$FPR = (1 - e^{-\frac{kn}{m}})^k$$

From this relationship we can see that the larger the CBF array, *m*, the lower the FPR, and assure that larger *n* leads to large *FPR*. However, a larger array also increases the space complexity and time complexity of the operations. As the value of *k* increases, it first reduces the probability of false positives but then increases it, as well as the space complexity and time complexity of the operations (Kim et.al., 2019). Given these dependencies, the values of *m* and *k* are typically optimized to offset the effects of the expected number of elements to be inserted, *n*, and achieve the desired level of accuracy, *FPR*.

One of the main advantages of using CBFs as an indexing technique is that they require a relatively small amount of memory and time. The space complexity of a CBF is O(m), where m is the number of bits in the array, as stated before. The time complexity of all operations in CBFs is O(k) since insertion, deletion, and search of an element only depend on the application of k hash functions (transform an element into an index, increment/decrement/check corresponding bit), each of which takes a constant amount of time regardless of the input size. Therefore, the time complexity of these operations is constant and does not depend on the size of the set. We can further simplify the scaling of these operations' runtime and express it as O(1) since k is also a constant. These properties make CBFs particularly useful in applications where time and space resources are limited.

Practical Applications

Spam filtering: CBFs can be used in email servers to efficiently identify and filter spam messages. By keeping a CBF of known spam email addresses or keywords, the server can quickly determine whether an incoming message is likely to be spam or not by searching for the incoming message's address or keywords in the CBF in O(1) time, and either reject or accept it accordingly. This helps in improving the accuracy of spam filtering and reducing the processing time of incoming email messages.

Data compression: CBFs can be used to identify and remove duplicate data items in data compression systems. By maintaining a CBF of previously seen data items, the system can

quickly identify duplicate data in O(1) time and compress it efficiently by applying a hash function to a given data object and checking if it is already contained in the CBF. This method helps in reducing storage requirements and improving the performance of data compression systems. It is also important that CBFs are not subject to false negatives, meaning that if a data object is not stored yet, we will definitely not get rid of it.

In these applications, CBFs help in improving system performance and scalability by providing an efficient and approximate way of querying the presence or absence of data items in a set. However, given the possibility of false positive results, an incoming email might be classified as spam and a data object marked as duplicate even though they were never put in a CBF.

Python Implementation

First, we need to choose a hash function for the CBF and justify the choice. A good hash function for a Counting Bloom Filter (CBF) should satisfy the following properties:

- Uniformity: It should uniformly distribute the hashed items across the CBF array, meaning that the function should map elements randomly (with equal probability) into each position in the array.
- Independence: The hash functions should be independent of each other, meaning that the output of one hash function should not affect the output of another.
- Non-colliding: The hash functions should minimize the number of collisions between different items, which means that different items should be hashed to different positions in the CBF array. This property will help minimize the risk of false positives since if

fewer elements will be mapped onto the same position, there will be less chance of incorrectly interpreting a non-zero bit in the CBF. Uniformity property is closely related to this principle as it ensures that elements are not preferentially mapped onto a few high-probability bits and hence, overlap.

- Sensitivity to input: A small change in the input should result in a significant change in the hash function output. In other words, the function should not be random but rather deterministic - each unique element should be mapped onto a unique set of bits in the CBF array. No matter how many times we repeat the mapping process for a given element, the answer (resulting set of bits) should not change.
- Efficiency: The hash function should be computationally efficient to compute so that it can be used to hash a large number of items in a reasonable amount of time

One such hash function that satisfies the above conditions is a polynomial rolling hash function, defined by Mount, 2017, as

$$H(str) = (\sum_{i=0}^{n} c_i p^i) \bmod m,$$

where *str* is an input string, *n* is the total number of characters in the string, c_i is am integer representation of the i-th character of a string, p^i is a pre-set integer to the power of the current character's index in a string, and *m* is the length of the CBF bit array.

Once the suitable hash function was chosen, I completed the Python Implementation of the CBFs (Appendix). The application was tested with a few test cases, which showed that the implementation is working correctly: it produces unique outputs for each different string and

produces the same output for a given string regardless of how many times we repeat hashing. Next, I tested the hash function itself. I repeatedly generated random strings of variable length (from 1 to 15 characters), extracted only the unique string instances, and applied the hash function to them, keeping track of any collisions (different strings being mapped onto the same index in the CBF array). The results show that after hashing approximately 5000 randomly generated unique strings, no collisions were found, indicating that different strings are being hashed to a unique set of indexes (Figure 1). The distribution of the resultant indexes is uniform as also can be seen from Figure 1, satisfying the property of uniformly distributing strings across the CBF with no preference for any specific index.

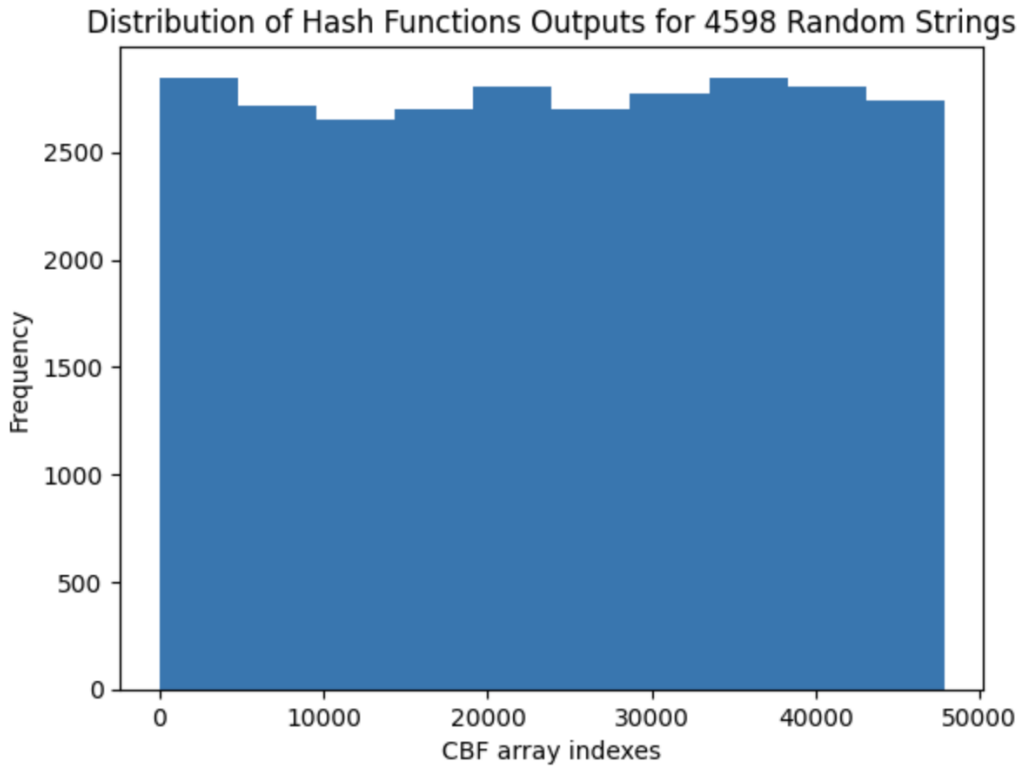

Figure 1. The figure shows the uniform distribution of hash values of more than 4500 strings, indicating that the mapping is performed randomly to minimize the number of collisions. In this example, no collisions were found.

Thus, we can see that our choice of a hash function satisfies the important hash function properties and will minimize the FPR.

In the Python implementation, we also applied dynamic formulas for determining the optimal number of hash functions and the length of the CBF array (memory size) based on the provided values of FPR and the number of items to be inserted. Luo et.al, 2019, describes the formula for the optimal number of the hash functions as

$$k = \frac{m}{n} ln2,$$

where *m* is the memory size and *n* is the number of items.

Given the optimal number of hash functions formula, we can derive a formula for the optimal memory size. We start by substituting it into the FPR expression derived previously:

$$FPR = (1 - e^{-(\frac{m}{n} ln2)\frac{n}{m}})^{\frac{m}{n} ln2}$$

Simplifying it, we get

$$ln\, FPR = -\frac{m}{n} (ln2)^2$$

Rearranging, we get

$$\frac{m}{n} = \frac{-log_2 FPR}{ln2} \approx - 1.44 log_2 FPR$$

From this equation we can see that the optimal memory size is given by

$$m \approx - n \cdot 1.44 \cdot log_2 FPR$$

Computational Analyses

Memory size scaling with FPR

Analysing this trend theoretically, we can use the memory size formula and infer that there is an inverse exponential relationship betweent the FPR and memory size : the larger the FPR gets, the smaller will the memory size become given a set number of items (Figure 2). Intuitively, it also makes sense since increased FPR means more collisions (more items are mapped onto the same bits of the array). To achieve that, decreasing the number of bits available will increase the chance of elements being mapped onto the same bits.

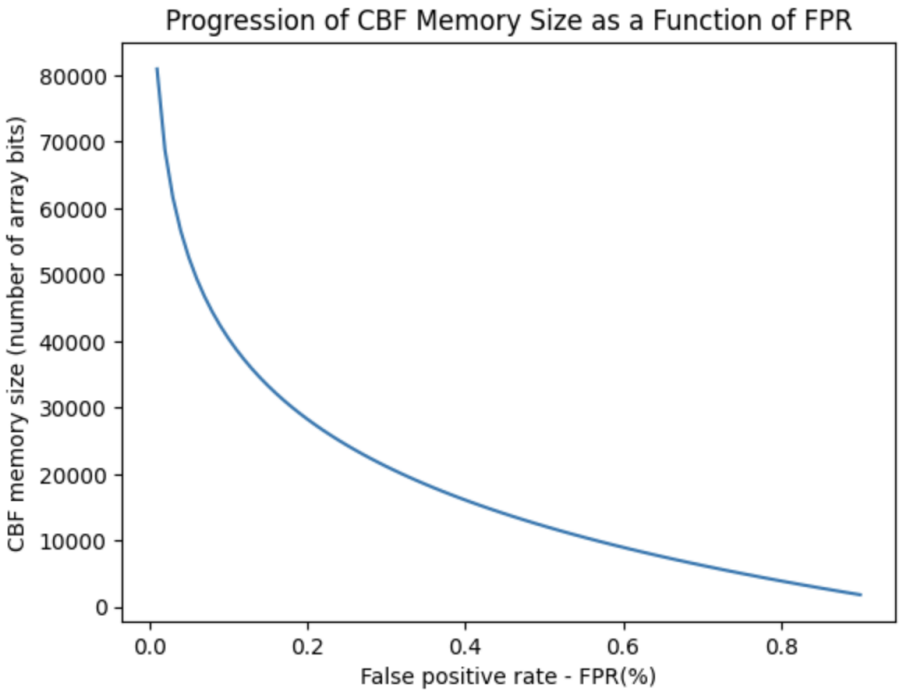

Figure 2. The figure shows how the CBF memory size decreases exponentially as the FPR grows larger. The results are presented for the *version 1* text and are the same for *version 2* and *3* texts.

Memory size scaling with the number of items stored

Using the same memory size formula, we can see that there is a direct linear relationship between the memory size and the number of items to be inserted. This is because if we keep FPR constant, the term $-\ 1.44\ \cdot\ log_2 FPR$ is going to be a positive constant (since 0<FPR<1, $log_2 FPR$ is negative). We can see this trend supported by empirical results in Figure 3.

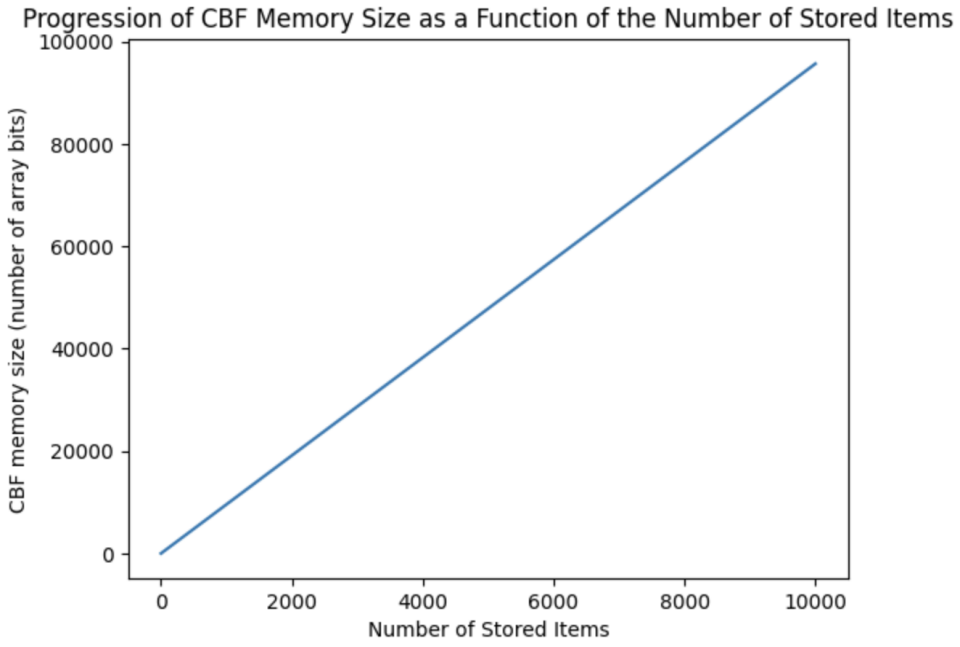

Figure 3. The figure shows how the CBF memory size increases linearly as the number of stored items grows larger. The results are presented for *version 1* text and are the same for *version 2* and *3* texts.

FPR scaling with the number of hash functions

From the FPR formula, we can see that the number of hash functions is inversely proportional to the FPR: increasing the number of hash functions decreases the FPR, as seen in Figure 4. Intuitively, it makes sense because increasing the number of hash functions increases the chance of creating a unique set of array bits for each element, thus reducing the probability of different

elements being mapped onto the exact same thing. As we can see from the figure, the FPR rapidly reduces and then remains relatively stable at around 0.5%, which is as low as it could be for a given memory size.

However, this trend only holds if we have a sufficiently large memory size (which we do in our case). Large memory size is what allows for creating unique sets of bits for each element when the number of hash functions increases. If we had a small memory size, then increasing the number of hash functions, on the other hand, would start mapping all elements onto all (or almost all) array bits, making extremely many collisions. Thus, increasing the number of hash functions can have opposite effects on the FPR depending on the memory size.

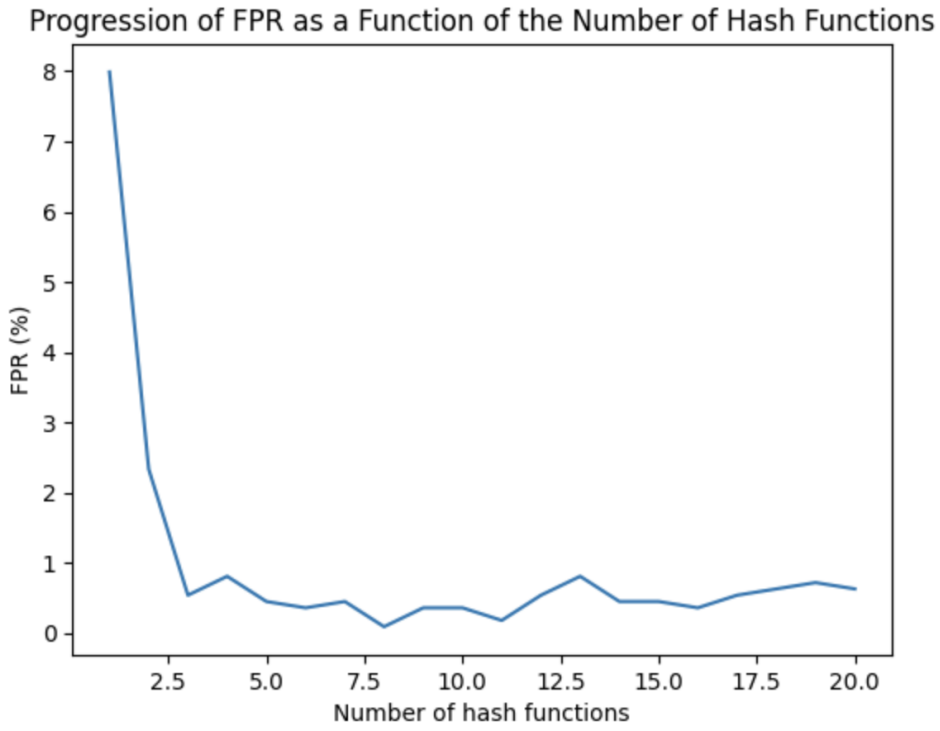

Figure 4. The figure shows how the FPR decreases rapidly due to the slight increase in the number of hash functions and then stabilizes and oscillates around 0.5% as the number of hash functions grows even larger. The results are presented for the *version 1* text and are the same for *version 2* and *3* texts.

Access time scaling with the number of items stored

For this experiment, I have chosen a few words that we could encounter in the provided Shakespear's text to check how long it would take for the algorithm to search (access) those words in the CBF. I timed the search for each word and then averaged those times across these words. I repeated this procedure for increasing number of items stored. The results can be seen from Figure 5, which shows that there is no general trend: the access time fluctuates around $6 \cdot 10^{-5} s$ with occasional rapid jumps up to $18 \cdot 10^{-5} s$, which can be attributed to the larger number of array bits to be checked until the word is found.

The generally stable access time is a result of the constant time search operation in CBFs, which we addressed earlier. Since to access an element we only need to apply hash functions to it and check if the resultant array bits are zeros or not, the search operation time is always O(1) regardless of the number of items stored.

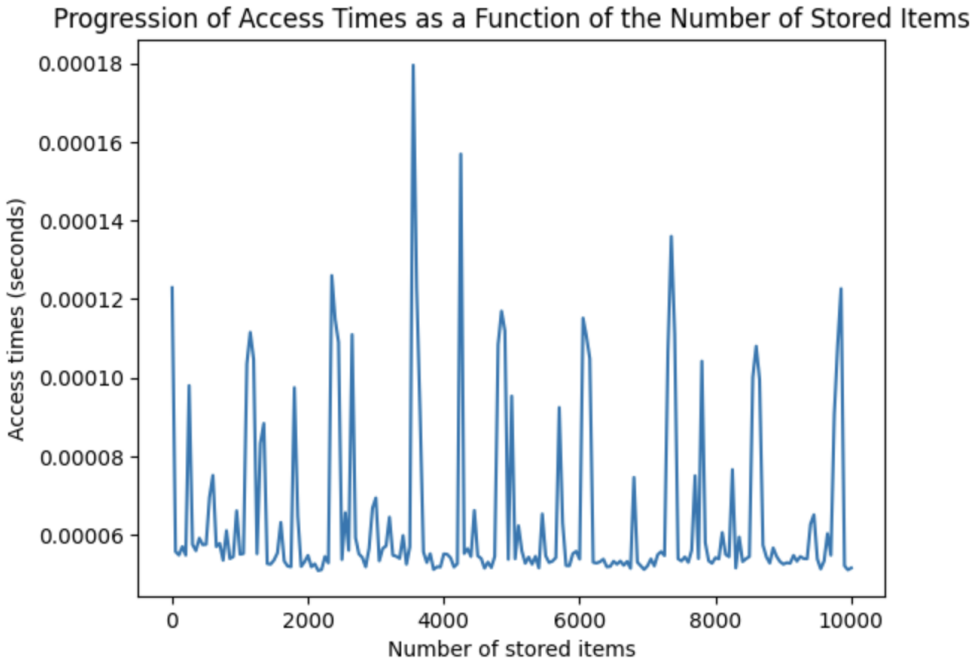

Figure 5. The figure shows how the access time develops as the number of stored items grows larger. The results are presented for *version 1* text and are the same for *version 2* and *3* texts.

Plagiarism Detector

Plagiarism Definition

Plagiarism is the act of using or presenting someone else's work or ideas as one's own without proper citation or attribution. In the context of plagiarism detection using the given PlagiarismDetector class, plagiarism between two texts is defined as the extent to which the n-grams in text1 are present in text2.

An n-gram is a contiguous sequence of n items such as words or letters. For example, if we have the sentence "The quick brown fox jumps over the lazy dog" and we consider n=3, then the resulting trigrams would be "The quick brown", "quick brown fox", "brown fox jumps", "fox jumps over", "jumps over the", "over the lazy", and "the lazy dog". The PlagiarismDetector class preprocesses the texts by splitting them into n-grams and then uses Counting Bloom Filters to detect the common n-grams between the two texts.

Thus, if a particular n-gram in text1 is also present in text2, it is counted towards the total number of common n-grams, and the plagiarism percentage is calculated by dividing the number of common n-grams by the total number of n-grams in text1. The result is multiplied by 100 to obtain a percentage value.

Advantages and Disadvantages of CBFs

Counting Bloom Filters (CBFs) are an efficient data structure that can be used to detect plagiarism between two texts. One of the main strengths of CBFs is their ability to detect plagiarism with a high degree of accuracy while minimizing the number of false positives. This is achieved by using multiple hash functions to map each n-gram in the text to multiple cells in the CBF, and then counting the number of times each cell is set. By comparing the CBFs of the two texts, the plagiarism detector can quickly identify the n-grams that are common to both texts. What also is very important is that CBFs have a zero probability of false negatives, meaning if there is plagiarism - it will be found. This property has an important academic implication as it is better to find plagiarism and later figure out it was not actually plagiarism than miss true plagiarism.

Compared to word-by-word comparisons, using n-grams and CBFs can be more effective at detecting plagiarism. This is because word-by-word comparison can be vulnerable to simple word substitutions or paraphrasing, while n-grams are less sensitive to these changes. Additionally, CBFs can be more efficient than other data structures, such as hash tables, because they use less memory and allow for constant-time insertions and lookups.

However, there are some limitations and failure modes of CBFs. One of the main limitations is that the accuracy of the plagiarism detector depends on the size of the CBF and the number of hash functions used. If the CBF is too small or the number of hash functions is too low, then the probability of false positives and false negatives increases. Additionally, CBFs can be vulnerable

to hash collisions, where two different n-grams are mapped to the same cells in the CBF, leading to false positives.

Hashing is another approach that can be used to detect plagiarism, but it may not be as effective as CBFs. Hashing is based on mapping the n-grams to fixed-length strings and then using hash functions to compare the strings. This approach can be more vulnerable to hash collisions than CBFs, and it may also be less efficient because it requires more memory to store the hash values. Overall, while hashing can be a useful tool for detecting plagiarism, it may not be as effective or efficient as CBFs in many cases.

Alternative Strategies

Longest Common Subsequence approach:

This implementation of plagiarism detection utilizes the longest common subsequence (LCS) algorithm to compare the similarity between two texts. The LCS algorithm calculates the length of the longest subsequence that is common to both texts. A subsequence is a sequence of characters that can be obtained by deleting some of the characters from the original sequence, but without changing the order of the remaining characters.

The lcs() method takes in two sequences (in this case, two strings) and returns the length of the longest common subsequence between them. It does this by initializing an m+1 by n+1 matrix, where m and n are the lengths of the two sequences. It then iterates over the two sequences, comparing the characters at each position. If the characters are the same, it sets the value in the matrix at the corresponding position to be one plus the value in the matrix at the previous

position. Otherwise, it sets the value in the matrix to be the maximum of the values to the left and above it. The final value in the matrix represents the length of the LCS.

The detect() method calls the lcs() method on the two input texts, and calculates the similarity between them as 2 times the length of the LCS divided by the total length of the two texts. The result is rounded to two decimal places and returned as a percentage.

Compared to the CBF implementation, the LCS implementation is more computationally expensive, as it has a time complexity of O(mn) where m and n are the lengths of the input texts. The CBF implementation, on the other hand, has a time complexity of O(k), where k is the length of the n-grams. However, the LCS implementation may be more accurate in some cases, as it takes into account the order of the words in the texts. The CBF implementation, in contrast, only checks whether the n-grams appear in both texts, regardless of their order.

Brute Force approach:

This implementation of the plagiarism detector is a brute force approach that uses a sliding window of length n to extract phrases (of n words) from both texts and compares them to find common phrases. The score is calculated by dividing the number of common phrases by the total number of phrases and multiplying the result by 100.

Compared to the CBF implementation, the brute force approach has a lower time complexity as it only requires iterating through the longer text once, whereas CBF requires iterating through both texts to preprocess and insert n-grams into the filter and then iterating through the n-grams

of the shorter text to search for matches in the filter. However, the time complexity of the brute force approach can become prohibitive for very long texts or very large values of n.

Also, the brute force approach works with a fixed length n-gram, while the CBF implementation can work with any n-gram length, and it can handle variations in the text, such as added or removed spaces between words, which can affect the matching of phrases in the brute force approach.

Finally, the brute force approach may not be able to handle some plagiarism variations, such as reordering words or replacing words with synonyms, which can be handled to some extent by the CBF implementation.

Comparing the runtimes of all three algorithms empirically, we can see from Figure 6 that the Brute Force approach is clearly the fastest. CBF is slower than Brute Force but still faster than Longest Common Subsequence. Based on the runtime analysis of the three plagiarism detection algorithms, the following observations can be made:

- CBF implementation has a relatively constant runtime for all input sizes since the number of hash functions and the number of buckets are fixed.
- The runtime of LCS implementation increases quadratically as the input text size increases because the LCS algorithm has O(n^2) time complexity (given that both texts have the same length n, as in our case), where n is the length of the input text.

- The runtime of Brute Force implementation increases linearly as the input text size increases since it compares all possible n-grams in the longer text with the n-grams in the shorter text.

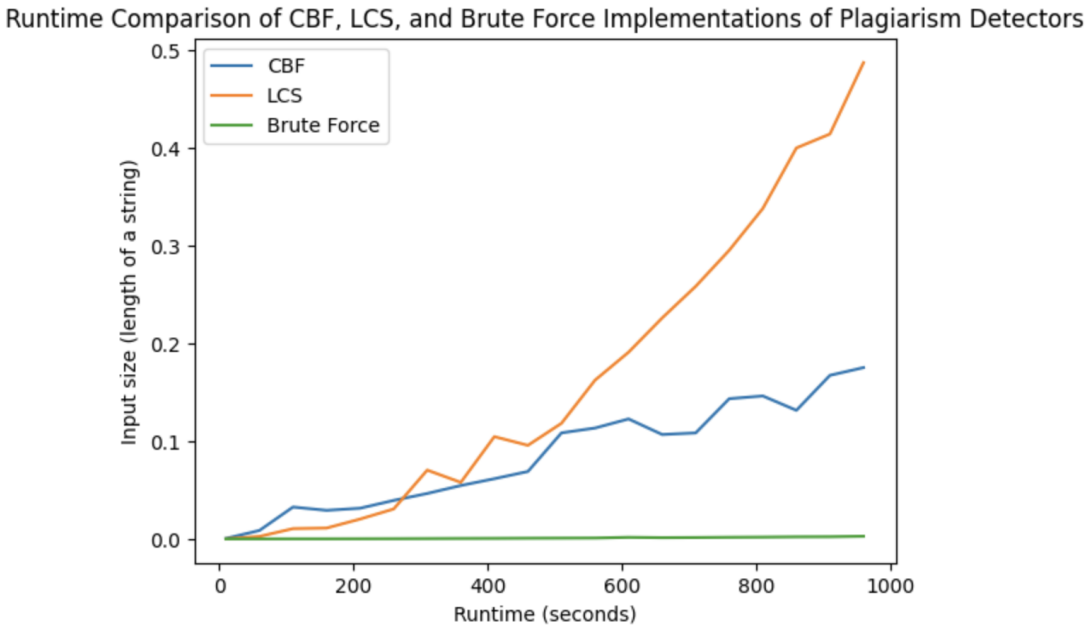

Figure 6. The comparison of the runtimes of CBF, LCS, and BF plagiarism detection approaches.

Considering the above observations, CBF and Brute Force implementations are the most efficient algorithm for large input sizes since their runtime remains constant. However, CBF has the limitation of possible false positives. LCS implementation is suitable for small input sizes since its runtime is proportional to the square of the input size.

Overall, the choice of plagiarism detection algorithm depends on the size of the input texts and the level of accuracy required. If a high level of accuracy is required for large input texts, CBF may not be suitable, and a combination of CBF and LCS could be used instead.

AI Note

No AI tools were used for this assignment.

References

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13*(7), 422–426. https://doi.org/10.1145/362686.362692

Luo, L., Guo, D., Ma, R. T. B., Rottenstreich, O., & Luo, X. (2019). Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *IEEE Communications Surveys & Tutorials*, *21*(2), 1912–1949. https://doi.org/10.1109/comst.2018.2889329

Mount, D. (2017). *CMSC 420 CMSC 420: Lecture 10 Hashing -Basic Concepts and Hash Functions*. https://www.cs.umd.edu/class/fall2019/cmsc420-0201/Lects/lect10-hash-basics.pdf

HC/LO Applications

#cs110-PythonProgramming: I have successfully implemented a Counting Bloom Filter (CBF) with all required operations, namely insertion, search, and deletion. I have also designed a suitable hash function for the CBF and ensured its accuracy by evaluating its performance on simple test cases. I also implemented mathematical formulas for optimal number of hash functions and memory size in Python. I applied CBF code to design an algorithm for a plagiarism detector. I also implemented other plagiarism detectors in Python using LCS and BF approaches and compared their runtime's empirically in Python.

#cs110-ComputationalCritique: I have provided a comprehensive explanation of why CBF is a superior choice over other algorithmic strategies and data structures for the given context. I have highlighted some key features of CBFs, such as their lack of false-negatives, and explained their benefits. Additionally, I have addressed the limitations of CBFs, including the possibility of false positives, and argued why they are still the most suitable option for plagiarism detection. I compared this approach to LCS and BF approaches in terms of strengths and limitations, as well as the scaling of their runtimes as a function of text size, and provided conclusions on which ones are better to use in which cases.

#cs110-CodeReadability: I have written proper and consistent docstrings for all functions, providing a concise description of their purpose. I have also added comments to explain each major step of the algorithm and adhered to Python naming conventions while naming my functions and variables. Furthermore, for the hash function, where expected test case results are known, I have added assert statements to assist reviewers in assessing my code's accuracy.

#cs110-AlgoStratDataStruct: I have justified my selection of CBF as the most appropriate data structure for plagiarism detection by discussing its constant time complexities for insertion, query, and deletion operations. Additionally, I have explained each step of the hash function and how it contributes to good hashing properties, such as low collision probability. My explanations are free from technical jargon and code translations to ensure clarity.

#cs110-ComplexityAnalysis: I have accurately calculated the time and space complexities of my algorithm and provided sound reasoning to justify my results. I have based my justification on code-level analyses and the results of my experiments. Moreover, I have interpreted these complexities and explained how the dependent quantity changes concerning an increase in the independent quantity. When comparing algorithms, I have used my interpretations to justify my conclusions on which algorithm is better.

#cs110-Professionalism: I have provided sufficient and evidence-based justifications for my complexity analyses. Additionally, I have formatted my code correctly and organized my assignment to make it simple to read and comprehend. I included a title page with my name in it, as well as numnbered the pages and provided a table of contents. I checked my text for typoes.

#dataviz: To effectively demonstrate the validity of my CBF implementation, I carefully selected appropriate data visualization techniques. I opted for a histogram to display the distribution of my hash function's outputs, and adhered to conventional standards by employing a line graph to showcase the relationships outlined in Computational Analyses section. My data visualization

adhered to best practices, featuring clearly labeled axes, titles, captions, legends, and well-differentiated colors to enable easy interpretation and analysis. Furthermore, I thoroughly analyzed and interpreted the data visualizations in the accompanying text. (87 words)

#algorithms: I implemented clear, step-wise, and well structured algorithms for CBF and all types of plagiarism detectors and showed that they worked correctly. See #cs110-PythonProgramming for the rest.

#critique: I critically analysed the strengths and weaknesses of CBFs and theise application as a plagiarism detector. I also critically analysed and compared other approaches, such as LCS and BF. Based on that analyses, I commented which methods are better in certain situations.

Appendix

```
import random
import string
import matplotlib.pyplot as plt
import numpy as np
import math
from requests import get
import time

class CountingBloomFilter:
    """
    Implements the Counting Bloom Filter which supports insertion, deletion, and search
    """
    def __init__(self, num_items=100000, fpr=0.01, num_hashfn=None):
        """
        Initializes the Counting Bloom Filter.

        Parameters
        ----------
        num_items : int
            Number of items to store in the filter. Default is 100000.
        fpr : float
            False positive rate of the filter. Default is 0.01.
        num_hashfn : int or boolean
            Number of hash functions to use. If not specified, the optimal
            number of hash functions is calculated based on the memory size
            and number of items.

        Attributes
        ----------
        num_items : int
            Number of items to store in the filter.
        fpr : float
            False positive rate of the filter.
        memory_size : int
            Number of cells in the Counting Bloom Filter.
        bit_array : numpy.ndarray
            Array of integers representing the Counting Bloom Filter.
```

```
        num_hashfn : int
            Number of hash functions used by the filter.
        hash_seeds : numpy.ndarray
            Array of random hash seeds used by the filter.
        """
        self.num_items = num_items
        self.fpr = fpr

        # Calculate the memory size based on the number of items and the false positive rate
        self.memory_size = int(self.num_items * (-1.44 * np.log2(self.fpr)))

        # Initialize the bit array with zeros
        self.bit_array = np.zeros(self.memory_size, dtype=np.int32)

        # Calculate the optimal number of hash functions based on memory size and number of
items
        if num_hashfn is None:
            self.num_hashfn = int((self.memory_size / self.num_items) * math.log(2))
        else:
            self.num_hashfn = num_hashfn

        # Generate random hash seeds for the number of hash functions
        self.hash_seeds = np.random.randint(0, np.iinfo(np.int32).max, size=self.num_hashfn,
dtype=np.int32)

    def hash_cbf(self, item):
        """
        Returns hash values of an item.

        Parameters
        ----------
        item : str
            Item to hash.

        Returns
        -------
        list of int
            Hash values of the item.
        """
        hash_values=[]
```

```
        # Use different integers p for different hash functions
        for seed in self.hash_seeds:
            hash_value = 0
            p_pow=1
            p=seed
            for char in item:
                # Calculate the hash value of each character using polynomial rolling hash function
                hash_value = (hash_value + (ord(char) - ord('a') + 1) * p_pow) % self.memory_size
                p_pow = (p_pow * p) % self.memory_size
            hash_values.append(hash_value)

        return hash_values

    def search(self, item):
        """
        Queries the membership of an element.

        Parameters
        ----------
        item : str
            Item to search for.

        Returns
        -------
        bool
            True if the item is probably in the filter, False otherwise.
        """
        hash_values = self.hash_cbf(item)
        # If at least one corresponding position in CBF array is 0 - the element is definitely not in
CBF
        # Otherwise, it probably is
        for hash_value in hash_values:
            if self.bit_array[hash_value] == 0:
                return False
        return True

    def insert(self, item):
        """
        Inserts a string to the filter.
```

```
        Parameters
        ----------
        item : str
            Item to search for.
        """
        hash_values = self.hash_cbf(item)
        # Increment the corresponding position in the CBF array by one
        for hash_value in hash_values:
            self.bit_array[hash_value] += 1

    def delete(self, item):
        """
        Removes a string from the filter.

        Parameters
        ----------
        item : str
            Item to search for.
        """
        if not self.search(item):
            print("Item not in filter.")
        hash_values = self.hash_cbf(item)
        # Decrement the corresponding position in the CBF array by one if the element is
considered to be present in CBF
        for hash_value in hash_values:
            if self.bit_array[hash_value] > 0:
                self.bit_array[hash_value] -= 1
            else:
                print("Item not in filter.")

# Test Cases

# Create a tiny CBF
cbf = CountingBloomFilter()

# check if it is deterninistic - hashing the same string shoudl always produce the same output
assert(cbf.hash_cbf("qwerty1") == cbf.hash_cbf("qwerty1"))
assert(cbf.hash_cbf("ytrewq1") == cbf.hash_cbf("ytrewq1"))

# check if it produces unique values for different strings
```

```
assert(cbf.hash_cbf("qwerty1") != cbf.hash_cbf("ytrewq1"))

# check numbers and symbols
assert(cbf.hash_cbf("123") != cbf.hash_cbf("-123"))
assert(cbf.hash_cbf("@@") == cbf.hash_cbf("@@"))

print("All test cases passed!")

# Hash Function Test

def min_collisions(num_items):
    """
    Creates a Counting Bloom Filter and adds random strings to it. Calculates the percentage of collisions
    found and plots the distribution of the hash functions' outputs.

    Parameters
    ----------
    num_items: int
        The number of items to add to the filter.
    """
    hash_set = []
    num_collisions = 0
    cbf = CountingBloomFilter(num_items=num_items)
    index_list = []
    string_list = []

    # generate random strings and add them to the filter
    for i in range(num_items):
        s = "".join(random.choices(string.ascii_lowercase, k=random.randint(1,15)))
        string_list.append(s)

    # check for collisions and record their hash values
    for s in set(string_list):
        h = cbf.hash_cbf(s)
        if h in hash_set:
            num_collisions += 1
        for i in h:
            index_list.append(i)
        hash_set.append(h)
```

```
    # print percentage of collisions found and plot distribution of hash function outputs
    if num_collisions > 0:
        print("Percentage of collisions found:", (num_collisions / num_items) * 100)
    else:
        print("No collisions found")

    plt.hist(index_list)
    plt.title(f'Distribution of Hash Functions Outputs for {len(set(string_list))} Random Strings')
    plt.xlabel('CBF array indexes')
    plt.ylabel('Frequency')

min_collisions(5000)

# Computational Analyses

url_version_1 = 'https://bit.ly/39MurYb'
url_version_2 = 'https://bit.ly/3we1QCp'
url_version_3 = 'https://bit.ly/3vUecRn'


def get_txt_into_list_of_words(url):
    '''
    Cleans the text data.

    Parameters
    ----------
    url : string
        The URL for the txt file.

    Returns
    -------
    data_just_words_lower_case: list
        List of "cleaned-up" words sorted by the order they appear in the original file.

    '''

    # Define a list of characters that we want to remove from the text
    bad_chars = [';', ',', '.', '?', '!', '_', '[', ']', '(', ')', '*']
```

```
    # Get the text data from the URL
    data = get(url).text

    # Remove the unwanted characters from the text
    data = ''.join(c for c in data if c not in bad_chars)

    # Replace newlines, tabs, and carriage returns with spaces
    data_without_newlines = ''.join(
        c if (c not in ['\n', '\r', '\t']) else ' ' for c in data)

    # Split the text into a list of words
    data_just_words = [
        word for word in data_without_newlines.split(' ') if word != '']

    # Convert all words to lowercase
    data_just_words_lower_case = [word.lower() for word in data_just_words]

    # Return the cleaned-up list of words
    return data_just_words_lower_case



version_1 = get_txt_into_list_of_words(url_version_1)
version_2 = get_txt_into_list_of_words(url_version_2)
version_3 = get_txt_into_list_of_words(url_version_3)

**Memory Size vs. FPR**

def memory_size_vs_fpr(data):
    """
    Plot the progression of CBF memory size as a function of false positive rate.

    Parameters
    ----------
    data : list
        The data to be used for testing the Counting Bloom Filter.
    """
    memory_sizes = []
    fpr = np.arange(0.01, 0.91, 0.01)
```

```
    # For each FPR, create a CBF and record its memory size
    for i in fpr:
        cbf = CountingBloomFilter(num_items=len(data), fpr=i)
        memory_sizes.append(cbf.memory_size)

    # Plot the memory size vs. FPR
    plt.plot(fpr, memory_sizes)
    plt.title('Progression of CBF Memory Size as a Function of FPR')
    plt.xlabel('False positive rate - FPR(%)')
    plt.ylabel('CBF memory size (number of array bits)')
    plt.show()


# Use all versions for testing
print('Version 1 Data')
memory_size_vs_fpr(version_1)
print('Version 2 Data')
memory_size_vs_fpr(version_2)
print('Version 3 Data')
memory_size_vs_fpr(version_3)
```

Memory Size vs. Number of Items

```
def memory_size_vs_num_items(data):
    """
    Plot the progression of CBF memory size as a function of the number of items.

    Parameters
    ----------
    data : list
        The data to be used for testing the Counting Bloom Filter.
    """

    memory_sizes = []
    num_items = np.arange(1,10002,50)

    # For each number of items, create a CBF and record its memory size
    for i in num_items:
        cbf=CountingBloomFilter(num_items=i)
        memory_sizes.append(cbf.memory_size)
```

```
    # Plot the memory size vs. number of items
    plt.plot(num_items,memory_sizes)
    plt.title('Progression of CBF Memory Size as a Function of the Number of Stored Items')
    plt.xlabel('Number of Stored Items')
    plt.ylabel('CBF memory size (number of array bits)')
    plt.show()



# Use all versions for testing
print('Version 1 Data')
memory_size_vs_num_items(version_1)
print('Version 2 Data')
memory_size_vs_num_items(version_2)
print('Version 3 Data')
memory_size_vs_num_items(version_3)
```

FPR vs. Number of Hash Function

```
def fpr_vs_num_hashfn(data):
    """
    Plot the progression of FPR as a function of the number of hash functions.

    Parameters
    ----------
    data : list
        The data to be used for testing the Counting Bloom Filter.
    """

    fpr_list = []
    num_hashfn = np.arange(1,21,1)
    # Extract unique words
    data=list(set(data))

    for i in num_hashfn:
        fpr=0
        cbf=CountingBloomFilter(num_items=len(data),num_hashfn=i)

        # Insert half of input data into the CBF
```

```
        for word in data[:len(data)//2]:
            cbf.insert(word)

        # Check of words from the other half of the data are in the CBF
        for word in data[len(data)//2:]:
            if cbf.search(word)==True:
                fpr+=1

        # Find FPR as a percentage of falsely classified words
        fpr_list.append((fpr/len(data[:len(data)//2]))*100)

    # Plot the FPR vs. number of hash functions
    plt.plot(num_hashfn,fpr_list)
    plt.title('Progression of FPR as a Function of the Number of Hash Functions')
    plt.xlabel('Number of hash functions')
    plt.ylabel('FPR (%)')
    plt.show()



# Use Version 1 for testing
print('Version 1 Data')
fpr_vs_num_hashfn(version_1)
print('Version 2 Data')
fpr_vs_num_hashfn(version_2)
print('Version 3 Data')
fpr_vs_num_hashfn(version_3)
```

Access Time vs. Number of Items

```
def acc_time_vs_num_items(data):
    """
    Plot the progression of access time as a function of the number of items

    Parameters
    ----------
    data : list
        The data to be used for testing the Counting Bloom Filter.
    """
```

```
    acc_times = []
    num_items = np.arange(1,10002,50)
    # Choose a set of random words that I believe are definitely in the text
    search_words=['the','Shakespear','love','and','tomorrow']

    for i in num_items:
        cbf=CountingBloomFilter(num_items=i)
        times=[]

        # Insert all words in the CBF
        for word in data:
            cbf.insert(word)

        # Search for selected words in the CBF and record time
        for word in search_words:
            start=time.time()
            cbf.search(word)
            end=time.time()
            times.append(end-start)

        acc_times.append(np.mean(times))

    # Plot the access time vs. number of items
    plt.plot(num_items,acc_times)
    plt.title('Progression of Access Times as a Function of the Number of Stored Items')
    plt.xlabel('Number of stored items')
    plt.ylabel('Access times (seconds)')
    plt.show()



# Use Version 1 for testing
print('Version 1 Data')
acc_time_vs_num_items(version_1)
print('Version 2 Data')
acc_time_vs_num_items(version_2)
print('Version 3 Data')
acc_time_vs_num_items(version_3)

# **Plagiarism Detector Applicaton**
```

CBFs and Hashing

```
class PlagiarismDetector:
    """
    A class to detect plagiarism between two texts using Counting Bloom Filters.
    """
    def __init__(self, num_items=10000, fpr=0.01, num_hashfn=None, n=10):
        """
        Initializes the PlagiarismDetector.

        Parameters:
        -----------
        num_items : int
            Number of cells in the Counting Bloom Filter.
        fpr : float
            Desired false positive rate of the Counting Bloom Filter.
        num_hashfn : int, optional
            Number of hash functions to use in the Counting Bloom Filter.
        n : int
            Size of n-grams to use in the plagiarism detection.

        Attributes:
        -----------
        cbf : CountingBloomFilter
            The Counting Bloom Filter used for plagiarism detection.
        n : int
            The size of n-grams used in plagiarism detection.
        """
        self.cbf = CountingBloomFilter(num_items=num_items, fpr=fpr,
num_hashfn=num_hashfn)
        self.n = n

    def preprocess(self, text):
        """
        Preprocesses the text by converting it to lowercase and splitting it into n-grams.

        Parameters:
        -----------
        text : str
```

```
        Text to preprocess.

    Returns:
    --------
    ngrams : list of str
        Preprocessed n-grams.
    """
    # Devide the text into n-grams
    ngrams = [''.join(text[i:i+self.n]) for i in range(len(text)-self.n+1)]
    return ngrams

def add_to_cbf(self,ngrams):
    """
    Adds n-grams to the Counting Bloom Filter.

    Parameters:
    -----------
    ngrams : list of str
        N-grams to add to the Counting Bloom Filter.
    """
    for i in ngrams:
        self.cbf.insert(i)

def detect_plagiarism(self, text1, text2):
    """
    Detects the extent of plagiarism between two texts as a percentage of common phrases.

    Parameters:
    -----------
    text1 : list of str
        First text to compare.
    text2 : list of str
        Second text to compare.

    Returns:
    --------
    result : list of float and list of str
        The percentage of n-grams in text1 that are also in text2 and the list of common phrases between the two texts.
    """
```

```
        intersection=0
        common_phrases=[]
        # Divide into n-grams and add to CBF
        ngrams1 = self.preprocess(text1)
        ngrams2 = self.preprocess(text2)
        self.add_to_cbf(ngrams1)
        # Count common n-grams between texts based on CBF presense and return corresponding
percentage
        for i in ngrams2:
            if self.cbf.search(i):
                intersection+=1
                common_phrases.append(i)
        return [intersection / len(ngrams1) * 100 if len(ngrams1) > 0 else 0.0, common_phrases]
```

Version 1 vs. Version 2

```
def plag_detector_result(text1, text2):
    """
    Prints the result of plagiarism detection between two texts.

    Parameters:
    -----------
    text1 : list of str
        First text to compare.
    text2 : list of str
        Second text to compare.
    """
    # Create an instance of a plagiarism detector and use it, printing the results
    detector = PlagiarismDetector()
    plagiarism_percentage = detector.detect_plagiarism(text1, text2)
    print(f"Plagiarism percentage: {plagiarism_percentage[0]:.2f}%")
    print(f"Common Phrases: {plagiarism_percentage[1]}")

plag_detector_result(version_1, version_2)
```

Version 2 vs. Version 3

```
plag_detector_result(version_2, version_3)
```

Version 1 vs. Version 3

```
plag_detector_result(version_1, version_3)
```

Version 1 vs. Version 1

```
plag_detector_result(version_1, version_1)
```

Longest Common Subsequence

```
class PlagiarismDetectorLCS:
    """
    A class to detect plagiarism between two texts using the Longest Common Subsequence
(LCS) algorithm.
    """
    def __init__(self, text1, text2):
        """
        Initializes the PlagiarismDetectorLCS.

        Parameters
        ----------
        text1 : str
            First text to compare.
        text2 : str
            Second text to compare.
        """
        self.text1 = text1
        self.text2 = text2

    def lcs(self, seq1, seq2):
        """
        Computes the length of the longest common subsequence between two sequences.

        Parameters
        ----------
        seq1 : str
            First sequence.
        seq2 : str
            Second sequence.

        Returns
```

```
        -------
        int
            Length of the longest common subsequence between seq1 and seq2.
        """

        # Determine the lengths of both sequences
        m, n = len(seq1), len(seq2)

        # Initialize a 2D array to store the length of longest common subsequence
        lcs_lengths = [[0] * (n + 1) for _ in range(m + 1)]

        # Iterate over each index of both sequences and update lcs_lengths accordingly
        for i in range(m):
            for j in range(n):

                # If the characters at the current index are the same, update lcs_lengths[i+1][j+1]
                if seq1[i] == seq2[j]:
                    lcs_lengths[i+1][j+1] = lcs_lengths[i][j] + 1

                # If the characters at the current index are different, update lcs_lengths[i+1][j+1] using
max function
                else:
                    lcs_lengths[i+1][j+1] = max(lcs_lengths[i+1][j], lcs_lengths[i][j+1])

        # Return the length of longest common subsequence
        return lcs_lengths[m][n]

    def detect(self):
        """
        Detects the extent of plagiarism between two texts using the LCS algorithm.

        Returns
        -------
        similarity : float
            Similarity (as percentage) between texts based on the length of the longest common
subsequence between seq1 and seq2.
        """
        # Compute the length of the longest common subsequence
        lcs_length = self.lcs(self.text1,self.text2)
```

```
        # Calculate the similarity
        similarity = (2.0 * lcs_length) / (len(self.text1)+ len(self.text2))

        # Round the similarity to 2 decimal places and return as a float
        return round(similarity * 100, 2)
```

Version 1 vs. Version 2

```
def plag_detector_lcs_result(text1,text2):
    """
    Detects the extent of plagiarism between two texts using the LCS algorithm and prints the
result.

    Parameters
    ----------
    text1 : list of str
        The first text to compare.
    text2 : list of str
        The second text to compare.
    """
    # Create an instance of a plagiarism detector and use it to find the similarity between texts,
printing the results
    detector = PlagiarismDetectorLCS(text1,text2)
    plagiarism_percentage = detector.detect()
    print(f"Plagiarism percentage: {plagiarism_percentage:.2f}%")

plag_detector_lcs_result(version_1, version_2)
```

Version 2 vs. Version 3

```
plag_detector_lcs_result(version_2, version_3)
```

Version 1 vs. Version 3

```
plag_detector_lcs_result(version_1, version_3)
```

Version 1 vs. Version 1

```
plag_detector_lcs_result(version_1, version_1)
```

Brute Force

```
def BruteForcePlagiarismDetector(text1, text2, n=4):
    """
    Detect copy-paste plagiarism between two texts.

    Parameters
    ----------
    text1: list
        The first text to compare.
    text2: list
        The second text to compare.
    n: int
        The number of words in a phrase.

    Returns
    -------
    score : float
        The similarity score as a percentage
    common_phrases : lst
        A list of common phrases between 2 texts
    """
    # Determine which text is shorter
    shorter_text = text1 if len(text1) <= len(text2) else text2
    longer_text = text1 if len(text1) > len(text2) else text2

    # Join shorter text into a single string
    shorter_text_joined = " ".join(shorter_text)

    # Initialize variables to keep track of common phrases and total phrases
    common_phrases = []
    total_phrases = 0

    # Iterate over phrases of length n in the longer text
    for i in range(len(longer_text) - n + 1):
        # Create a phrase from the current window of words
        current_phrase = " ".join(longer_text[i:i+n])
        # If the phrase is in the shorter text, add it to the list of common phrases
        if current_phrase in shorter_text_joined:
            common_phrases.append(current_phrase)
```

```
        total_phrases += 1

    # Calculate the similarity score as the percentage of common phrases out of total phrases
    score = (len(common_phrases) / total_phrases) * 100

    return score, common_phrases
```

Version 1 vs. Version 2

```
def plag_detector_bf_result(text1,text2):
    """
    Detect copy-paste plagiarism between two texts using Brute Force algorithm.
    It prints out the similarity score as a percentage and common phrases.

    Parameters
    ----------
    text1 : list
        The first text to compare.
    text2 : list
        The second text to compare.
    """
    plagiarism_percentage, common_phrases = BruteForcePlagiarismDetector(text1,text2)
    print(f"Plagiarism percentage: {plagiarism_percentage:.2f}%")
    print("Common Phrases: ", common_phrases)

plag_detector_bf_result(version_1, version_2)
```

Version 2 vs. Version 3

```
plag_detector_bf_result(version_2, version_3)
```

Version 1 vs. Version 3

```
plag_detector_bf_result(version_1, version_3)
```

Version 1 vs. Version 1

```
plag_detector_bf_result(version_1, version_1)
```

Test of All Algorithms on a Simple Text

```
text1 = ["The", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]
text2 = ["The", "quick", "brown", "dog", "jumped", "over", "the", "lazy", "fox"]

def quick_test(text1,text2):
    """
    Quickly tests the performance of three different plagiarism detection approaches on a simple input.

    Parameters
    ----------
    text1 : list of str
        The first text to compare.
    text2 : list of str
        The second text to compare.
    """
    # CBF
    detector = PlagiarismDetector()
    similarity_cbf= detector.detect_plagiarism(text1, text2)

    #LCS
    detector2 = PlagiarismDetectorLCS(text1, text2)
    similarity_lcs = detector2.detect()

    #Brute Force
    similarity_bf, common_phrases  = BruteForcePlagiarismDetector(text1, text2)

    print(f"Plagiarism percentage according to CBF approach: {similarity_cbf[0]:.2f}%")
    print(f"Plagiarism percentage according to LCS approach: {similarity_lcs:.2f}%")
    print(f"Plagiarism percentage according to Brute Force approach: {similarity_bf:.2f}%")

quick_test(text1,text2)
```

Plagiarism Detectors Comparison

```
# Define a function to generate random text of a given length
def generate_random_text(length):
```

```
    """
    Generates a random text of given length.

    Parameters
    ----------
    length : int
        The length of the random text to be generated.

    Returns
    -------
    text : list
        A list of random letters generated from 'a' to 'j'.
    """
    text = []
    for i in range(length):
        text.append(random.choice(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']))
    return text

def compare_plagiarism_methods(input_sizes, num_tests):
    """
    Compare the runtime of different plagiarism detection methods.

    Parameters
    ----------
    input_sizes : list
        A list of input sizes (i.e., lengths of the texts) to compare.
    num_tests : int
        The number of times to run each method for each input size.

    Returns
    -------
    cbf_runtime_list : list
        A list of average runtimes (in seconds) of the CBF implementation for each input size.
    lcs_runtime_list : list
        A list of average runtimes (in seconds) of the LCS implementation for each input size.
    bf_runtime_list : list
        A list of average runtimes (in seconds) of the brute force implementation for each input
size.
    """
    cbf_runtime_list=[]
```

```
lcs_runtime_list=[]
bf_runtime_list=[]

# Run the tests for each input size
for input_size in input_sizes:
    # Generate two random texts of the given length
    text1 = generate_random_text(input_size)
    text2 = generate_random_text(input_size)

    # Measure the runtime of the CBF implementation
    cbf_start_time = time.time()
    for i in range(num_tests):
        detector = PlagiarismDetector()
        detector.detect_plagiarism(text1, text2)
    cbf_end_time = time.time()

    # Measure the runtime of the LCS implementation
    lcs_start_time = time.time()
    for i in range(num_tests):
        detector = PlagiarismDetectorLCS(text1, text2)
        plagiarism_percentage = detector.detect()
    lcs_end_time = time.time()

    # Measure the runtime of the Brute Force implementation
    bf_start_time = time.time()
    for i in range(num_tests):
        BruteForcePlagiarismDetector(text1, text2)
    bf_end_time = time.time()

    # Calculate the average runtimes
    cbf_runtime = (cbf_end_time - cbf_start_time) / num_tests
    lcs_runtime = (lcs_end_time - lcs_start_time) / num_tests
    bf_runtime = (bf_end_time - bf_start_time) / num_tests

    # Store average runtimes
    cbf_runtime_list.append(cbf_runtime)
    lcs_runtime_list.append(lcs_runtime)
    bf_runtime_list.append(bf_runtime)

return cbf_runtime_list,lcs_runtime_list,bf_runtime_list
```

```
# Define inout sizes and run the function
input_sizes=[i for i in range(10, 1000, 50)]
cbf_runtime_list,lcs_runtime_list,bf_runtime_list =
compare_plagiarism_methods(input_sizes,50)

# Plot the results
plt.plot(input_sizes ,cbf_runtime_list,label='CBF')
plt.plot(input_sizes ,lcs_runtime_list,label='LCS')
plt.plot(input_sizes ,bf_runtime_list,label='Brute Force')
plt.legend()
plt.title('Runtime Comparison of CBF, LCS, and Brute Force Implementations of Plagiarism
Detectors')
plt.ylabel('Input size (length of a string)')
plt.xlabel('Runtime (seconds)')
plt.show()
```