Final Computational Application

Plagiarism Detector using CBF

CS 110: Data Structures and Algorithms
Prof. R. Shekhar

December 14, 2022

Bloom Filter and Counting Bloom Filters

Bloom Filter

Bloom Filter (BF) is a probabilistic data structure that is used to test set membership, meaning we can ask the question: Given an element x, does it belong to the set S? Since it is a probabilistic data structure false positive results are possible but false negatives are not. So for the set membership query, the result can either be "possibly in set" or "definitely not in set".

Definition: Given a set S of elements (or keys), a Bloom Filter is made up of a bit array B that is composed of m bits (like a list indexed from B[0] to B[m-1]) and a set of k hash function $h_1, h_2, h_3, \ldots . h_k : K \rightarrow \mathbb{Z}_m$ (See Figure 1 for an example). Each of the k hash functions hashes/maps the elements of S to one of the m array positions in the bit array B.

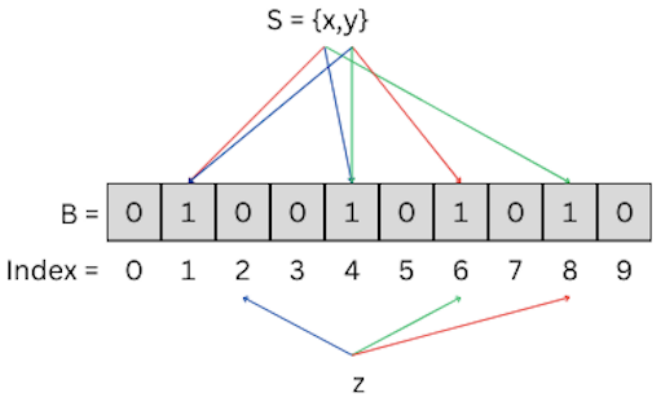


Figure 1: The figure represents a Bloom Filter B, x and y are elements of set S that are inserted in the filter. The different colored arrows represent different hash functions and they show the position in the bit array x and y are mapped to. The element z here is not part of the set S because the blue hash function hashes it to a bit-array position with 0. For this figure, m = 9, k = 3.

To insert an element into the bloom filter, set all the bits corresponding to the k-array positions of an element based on the k-hash functions to 1.

To search for an element in a set S, find the k-array positions from the k hash functions and then check if any of the k-array positions are set to 0 in the bloom filter. If even a single bit in the k-array positions is set to 0 this would mean that the element is definitely not in S because all bits corresponding to the k-array positions should have been set to 1 when the element was inserted. If all the bits are 1, this would mean that the element is possibly in S, possibly because there is a chance that the bits were set to 1 by other elements that were inserted into the bloom filter.

A drawback of Bloom filters is that elements cannot be deleted from the Bloom filter without risking the introduction of false negatives as deleting an element can set the value of bits corresponding to other elements to 0. Deletion of an element would require the reconstruction of the Bloom filter.

Variants of Bloom Filter, like Counting Bloom Filters allow deletions without the risk of false negatives given the element to be deleted was previously inserted in the set.

Counting Bloom Filters (CBFs)

A Counting Bloom Filter (CBF) is an advanced variant of the standard Bloom Filter that retains the probabilistic characteristics while enhancing its capabilities. Unlike the binary setup of the standard Bloom Filter, which employs a bit array where each position can only be 0 or 1, a Counting Bloom Filter uses an array of multi-bit counters. Each counter in this array can hold a nonnegative integer value, significantly increasing the state information that the structure can maintain for each element.

The multi-bit counters allow for quick threshold checks for an element in a set, meaning we can ask the question: Given an element x and a threshold θ, do θ occurrences of element x belong to the set S? Similar to the probabilistic nature of the Bloom filters, false positive results are possible but false negatives are not. For the threshold question, the answer can either be "definitely less than θ occurrences of element x in the set" or "probably θ occurrences of element x in the set". This helps track how many times an element is inserted. This can also be used as a test for set membership, we can simply put the value of θ = 1.

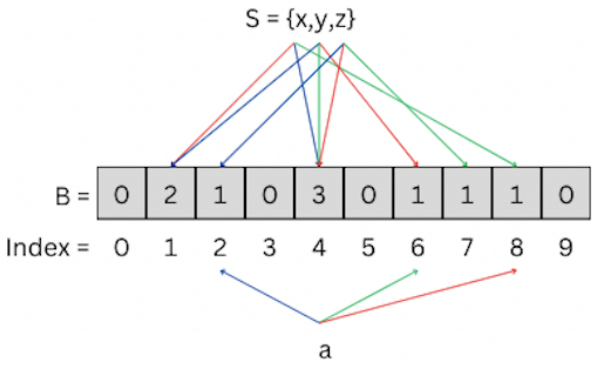


Figure 2: The figure represents a Counting Bloom Filter B, x,y and z are elements of set S that are inserted in the filter. The different colored arrows represent different hash functions and they show the position in the bit array the elements of S are mapped to. The element a here will return true if searched in S with θ =1 but it will be a false positive. For this figure, m = 9, k = 3.

Operations supported by CBFs and their order of growth

Insertion: Similar to Bloom Filter, when an element is inserted into a CBF, it undergoes hashing through k hash functions. Each hash function maps the element to a specific index in the CBF's internal array. The counters at these indices are then incremented by one. This increment reflects the addition of the element and is crucial for the CBF's ability to track multiple occurrences of the same element.
The time complexity of this operation is O(k), which is linear with respect to the number of hash functions. This linear relationship arises because the time it takes to insert an element is directly proportional to the number of hash function computations required, and each of these computations is assumed to execute in constant time.

Deletion: This operation allows for the removal of elements from the filter. The process begins similarly to insertion, where the element to be removed is passed through the same set of hash functions. These functions produce indices that correspond to positions in the CBF's array. The counters at these positions are then decremented. It's crucial during deletion to ensure that counters do not fall below zero, as this would lead to incorrect representations of the elements in the filter.
Like insertion, the deletion operation also has a time complexity of O(k). This is because the process involves k hash computations and k subsequent decrements of the counters, each of which is a constant-time operation. Deletion is a significant feature as it provides the flexibility to dynamically modify the set of elements represented in the CBF, which is particularly useful in evolving datasets.

Search: The search operation in CBFsis designed to ascertain whether an element is potentially part of the set represented by the filter. To perform a search, the element is processed through the CBF's hash functions, generating a series of indices. The CBF then checks the counters at these indices. If all counters are greater than zero, it suggests that the element might be present in the filter. However, if any counter is zero, it is definitive that the element has not been added to the CBF.
The time complexity for the search operation is also O(k), mirroring that of insertion and deletion. This is attributed to the necessity of performing k hash computations and examining k counters in the array. While efficient, it's important to note that search operations in CBFs can yield false positives, where the filter indicates the presence of an element that is not actually in the set.

Applications of CBFs

1. Web security services use CBFs to maintain a database of known malicious URLs and check each accessed URL against this database.

   The list of known malicious URLs is extensive and continuously growing. Storing and searching through this list using traditional methods would require significant memory and time. A CBF allows for a quick lookup to check if a URL is potentially malicious. If a URL check results in a positive, a more thorough (and computationally expensive) check can be performed to confirm the result. This two-step process efficiently filters out safe URLs, reducing the need for expensive computations.

2. Social media platforms or user account management systems use CBFs to quickly check if a username is already taken.

   With millions of users, checking for the availability of a username in a vast database can be time-consuming if done exactly. A CBF can be used to perform a fast, approximate check to see if a username might be taken. If the CBF indicates that the username is available (negative result), it is indeed available. If the CBF indicates the username might be taken (positive result), a precise database query can follow to confirm. This reduces the load on the database by filtering out clearly available usernames with a quick CBF check.

3. In a building security system, CBFs can be used to validate whether a keycard is active or has been reported lost/stolen.

   A security system might need to handle thousands of keycards. When a keycard is swiped, the system must rapidly determine if it should grant access. A CBF can keep track of all valid keycards and quickly indicate if a card is potentially invalid. This is particularly useful when keycards are revoked, as the CBF can be updated to reflect this change. If a card is flagged by the CBF, further checks can be performed to ensure that it has indeed been revoked, thus maintaining security while ensuring quick access for legitimate cardholders.

Python Implementation of CBFs

```python
#Q3 IMPLEMENTATION OF CBFs
import random
random.seed(10)
import math
class CountingBloomFilter:
    """
    A Counting Bloom Filter implementation that supports insertion, deletion,
    and membership query of elements.

    Attributes
    ----------
    fpr : float
        The desired false positive rate (between 0 and 1).
    num_item : int
        The expected number of items to be stored in the bloom filter.
    memory_size : int
        The size of the bloom filter's internal array, calculated
        based on the desired false positive rate
        and the expected number of items.
    num_hashfn : int
        The number of hash functions to be used, calculated based on
        the memory size and the number of items.
    array : list
        The internal array of the bloom filter where each index represents a counter.
    """

    def __init__(self, fpr, num_item):
        """
        Initialize the Counting Bloom Filter with a desired false
        positive rate and expected number of items.

        Parameters
        ----------
        fpr : float
            The desired false positive rate (between 0 and 1).
        num_item : int
            The expected number of items to be stored in the bloom filter.
        """

        self.fpr = fpr
        self.num_item = num_item

        # Calculate the size of the array needed to achieve the desired false positive rate.
        self.memory_size = math.ceil(-(num_item * math.log(fpr)) / (math.log(2) ** 2))

        # Calculate the optimal number of hash functions based on the size of the array
        #and the expected number of items.
        self.num_hashfn = math.ceil((self.memory_size / num_item) * math.log(2))

        # Initialize the internal array of the bloom filter with counters set to 0.
        self.array = [0] * self.memory_size

    def search(self, item):
        """
        Search for an item in the Counting Bloom Filter.

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

        Returns
        -------
        bool
            True if the item might be in the filter, False if
            the item is definitely not in the filter.
        """
        hash_values = self.hash_cbf(item)
        return all(self.array[hash_value] > 0 for hash_value in hash_values)

    def insert(self, item):
        """
        Insert an item into the Counting Bloom Filter.

        Parameters
        ----------
        item : str
            The item to insert.
        """
        hash_values = self.hash_cbf(item)
        for hash_value in hash_values:
            self.array[hash_value] += 1
```

```
def delete(self, item):
    """
    Remove an item from the Counting Bloom Filter, if it exists.

    Parameters
    ----------
    item : str
        The item to remove.
    """
    if self.search(item):
        hash_values = self.hash_cbf(item)
        for hash_value in hash_values:
            if self.array[hash_value] > 0:  # Ensure not to go below zero
                self.array[hash_value] -= 1
    else:
        print(f"{item} does not exist in the CBF")

def convert_to_int(self, string):
    """
    Generate a integer for a given string using ord()

    Parameters
    ----------
    string : str
        The string to convert to integer.

    Returns
    -------
    int
        The integer value of the string.
    """
    int_value = 211  # Prime number seed for reduced collision probability

    for i in range(len(string)):
        char = ord(string[i])
        int_value = int_value*char

    return int_value

def hash_cbf(self, item):
    """
    Compute a series of hash values for an item using a base hash function.

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

    Returns
    -------
    list
        A list of hash values for the given item.
    """
    hash_values = []
    for i in range(self.num_hashfn):

        # Double hashing: combine the base hash with a shifted version multiplied by the index
        hash_value = (self.convert_to_int(item) + i * \
                      (self.convert_to_int(item) << 5)) % self.memory_size
        hash_values.append(hash_value)

    return hash_values
```

Choice of Hash Function:

The CBF implementation uses the division method, bit manipulation, use of prime numbers, and double hashing to create a class of hash functions that can be used to create k hash functions based on **num_hashfn.**

A good hash function should have the following properties:

1. It should approximately satisfy the assumption of simple uniform hashing i.e., each element should be equally likely to be hashed to any of the m array positions in the CBF. Our CBF implementation ensures that in two ways:

   Since for the plagiarism detector application of CBF later, the input will be a list of strings, we define the **convert_to_int()** function to convert a string into an integer value that can later be hashed to an array position for our CBF. The method utilizes the ord() function of Python to convert every character of the string into a Unicode and then stores the weighted sum of those characters into int_value, then becomes the final integer value of a given string. This was done to ensure that no two strings transform into the same integer value because that will lead to a collision of array positions after hashing as the same integer values hash to the same array positions for a hash function. The use of a prime number for the weighted sum calculation further decreases this scenario.

   After we have an integer value for a string, we use **hash_cbf()** to convert that string into k hash values based on num_hashfn. Double hashing ensures that no two hash functions hash the integer value to the same array position. We obtain a new hash function by multiplying the integer value by i and shifting it to 5 bits. This helps us create h hash functions as the value of i goes from 0 to i-1. Finally, by dividing the final integer value by m and taking the remainder, we ensure every string is mapped to of on m array positions.

   As seen in Figure 3, the aforementioned choice of hash functions gives us an approximately simple uniform hashing.

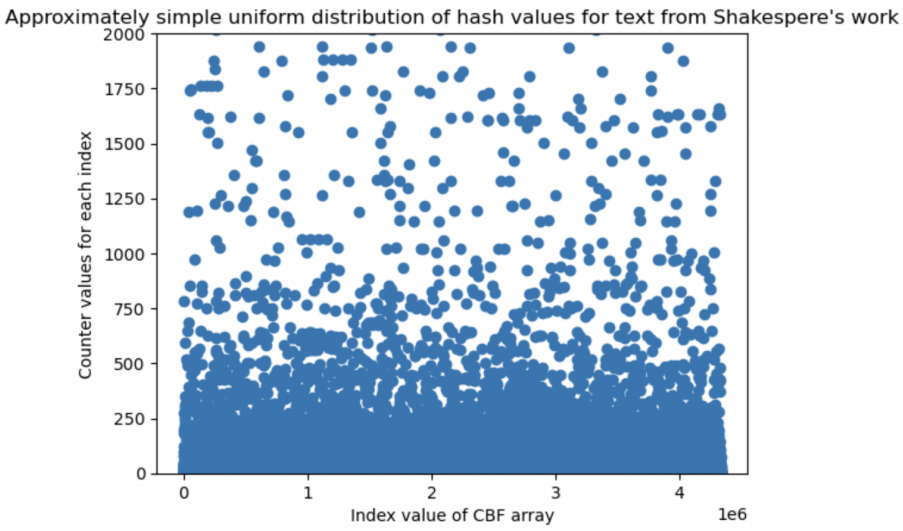


Figure 3: Figure to validate the approximately simple uniform hashing of the hash functions defined in CBF implementation

2. The hash function should be deterministic, meaning for a given string, the hash values should be the same no matter how many times we insert that element into the CBF. The hash function does not use any random element to decide the hash value of a string. Hence it is deterministic.
3. The hash functions should be efficient. The hash functions I have created are made of constant time operations. Hence, they are quick and efficient.

Testing the CBF implementation

I have created three test cases to show that the CBF implementation works as intended. The test cases test the insertion, deletion, and search functionality. They pass the assertion statements. Hence, our CBF implementation is correct.

```python
#Q3(C.) TESTING THE CBF IMPLEMENTATION
random.seed(10)
cbf = CountingBloomFilter(fpr = 0.01, num_item = 3)

# Test 1: Checks Insertion and Deletion Functionality
L1 = ["Mango","Orange","Banana"]

for i in L1:
    cbf.insert(i)

for i in L1:
    cbf.delete(i)
    cbf

for i in L1:
    assert cbf.search(i) == False
print("All assertions were passed for Test case 1")


# Test 2: Checks Deletion Functionality
L2 = ["Big","Medium","Small"]

for i in L2:
    cbf.insert(i)

for i in L2:
    cbf.delete(i)

for i in L2:
    cbf.delete(i)


# Test 3:  Checks Insertion and Search Functionality
L1 = ["CS","NS","BS"]

for i in L1:
    cbf.insert(i)

for i in L1:
    assert cbf.search(i) == True

assert cbf.search("AH") == False

print("All assertions were passed for Test case 3")
```

```
All assertions were passed for Test case 1
Big does not exist in the CBF
Medium does not exist in the CBF
Small does not exist in the CBF
All assertions were passed for Test case 3
```

Relationship between memory size (m), number of items(n), False positive rate (FPR)

We can use some basic probability theory to derive the following equation (Wikipedia,2023).

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

Relationship between the number of hash functions (k), number of items(n), and memory size (m)

The following equation can also be derived using probability (Wikipedia,2023)

$$k = \frac{m}{n} \ln 2$$

Effectiveness of CBF implementation

a. How does the memory size scale with FPR?

Theoretically,

$$m = -\frac{n \cdot \ln \varepsilon}{(\ln 2)^2}$$

From this formula, it is clear that if n is constant then,

$$m \propto C \cdot \ln(\varepsilon)$$
$$C = \frac{n}{(\ln 2)^2}$$

Meaning m is inversely proportional to ε because the false-positive rate is between 0 and 1. So as the FPR decreases we can expect the memory size to increase.

Using the all_text data which gives us a constant n and the CBF implementation for varying false positive rates, we can see memory size is dependent on FPR according to our theoretical expectations.

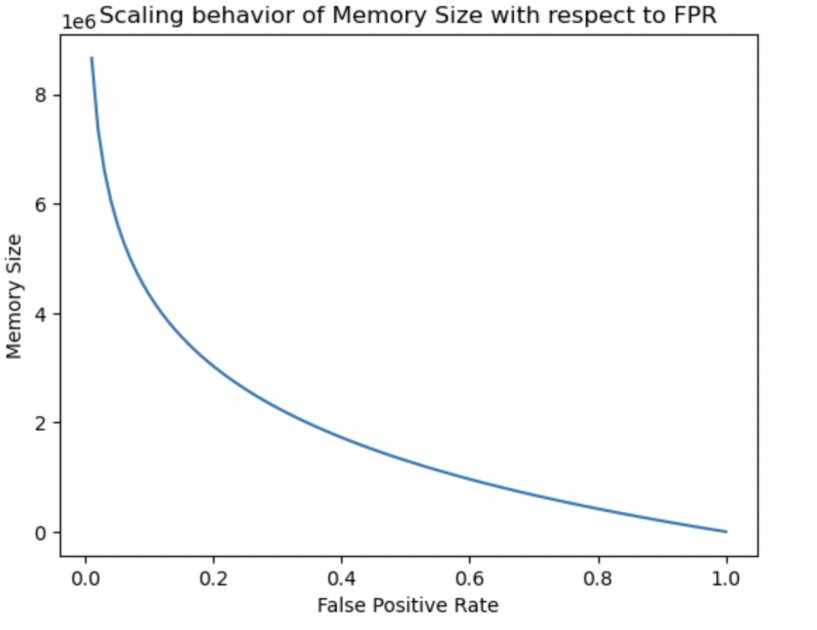


Figure 4: Line plot showing the scaling behavior of memory size based on FPR

b. How does the memory size scale with the number of items stored for a fixed FPR?

Again using the formula as in a. If we keep the FPR fixed, m will be directly proportional to the number of items (n). So, we can expect a linear dependency between the two variables.

$$m \propto C \cdot n$$
$$C = \frac{\ln \varepsilon}{(\ln 2)^2}$$

By using the CBF implementation with different values of FPR, we can see how our memory size is dependent on n. As expected theoretically, m scales linearly with n for varying values of FPR (See Fig).

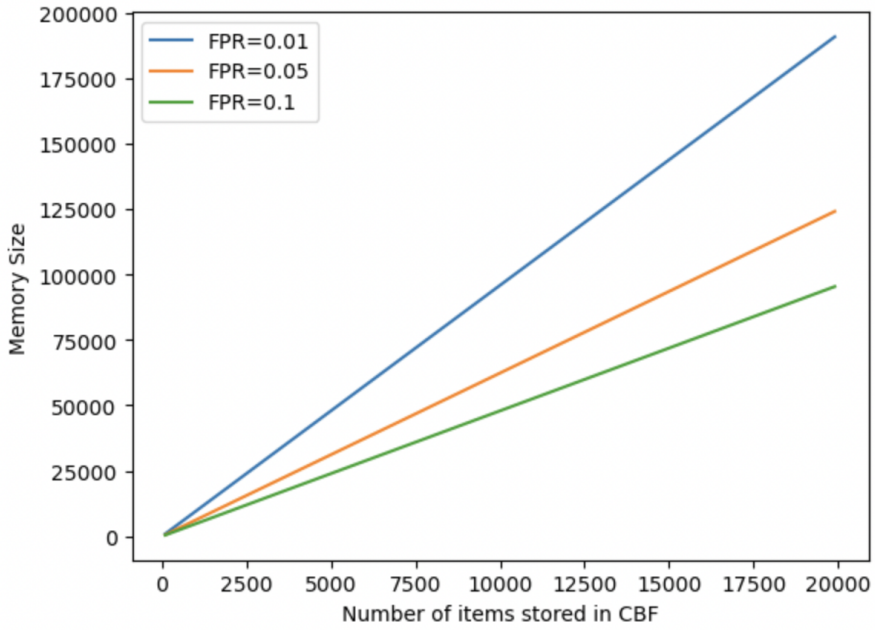


Figure 5: Line plot showing the scaling behavior of memory size based on items stored in CBF

c. How does the actual FPR scale with the number of hash functions?

Theoretically,

$$k = \frac{m}{n} \cdot \ln 2 \ = -\frac{\ln \varepsilon}{\ln 2}$$
$$after\ some\ rearrangement,$$
$$\ln \varepsilon \propto C \cdot k$$
$$C = -\ln 2$$

Based on the equation above, if we keep m and n constant, the FPR should decrease as the number of hash functions increases.

We can modify our CBF code so we can vary the number of hash functions while keeping m and n constant and see how FPR scales with k.

As expected, FPR decreases as the number of hash functions increases.

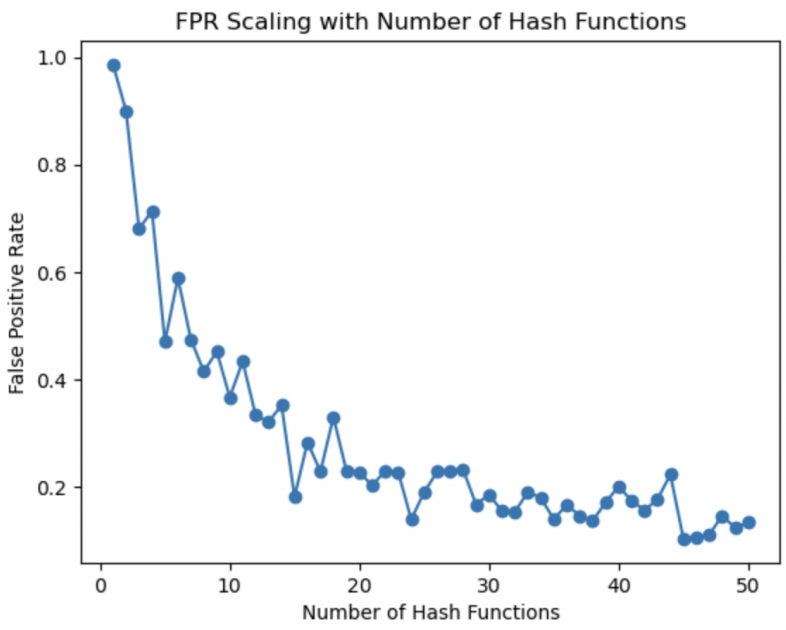


Figure 6: Line plot showing the scaling behavior of FPR based on the number of hash functions

d. How does the access time to hashed values scale with the number of items stored in a CBF kept at constant FPR?

Theoretically, the access time to hashed values should remain constant as we only need to evaluate k hash functions to search for an item irrespective of the number of items stored. So, I would expect to see constant scaling.

Based on experiments (See Figure), we can see that the scaling is constant for access time to hashed values as the number of items increases. There are some fluctuations for small values of n but that can be because of internal memory usage and time calculations.

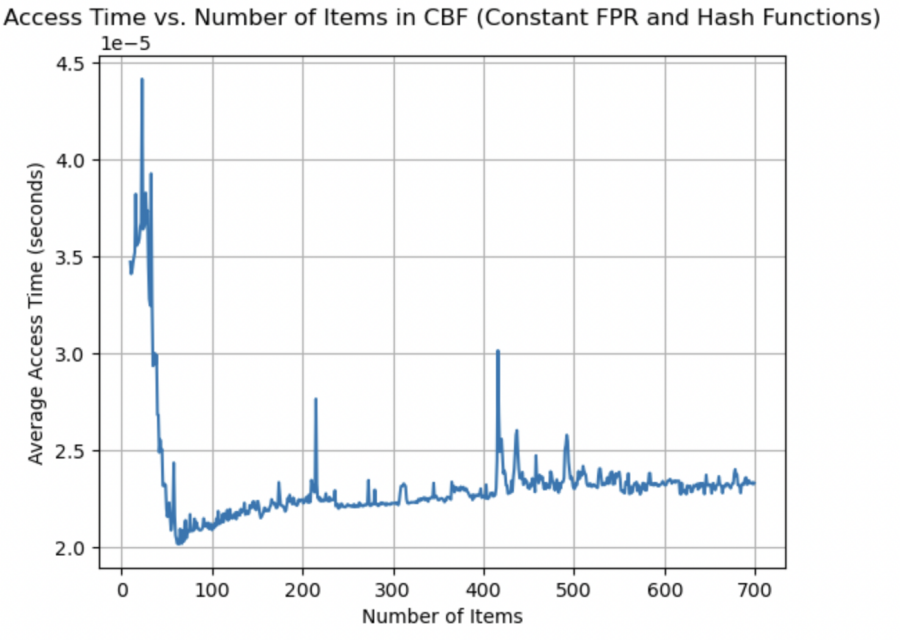


Figure 7: Line plot showing the scaling behavior of average access time to hash values based on the number of items.

Using CBFs for Plagiarism Detection

Defining Plagiarism: Generally, plagiarism is defined as the practice of using others' work or ideas as one's own without proper acknowledgment. In this implementation, I have defined a text to be plagiarized if five consecutive words are exactly the same in the two text files.

Defining plagiarism between two texts like this makes sense because there are many common words ("because", "therefore", "as", etc.) to phases ("According to the question", "In conclusion", and "Once upon a time"m, etc.) that would be flagged as plagiarism if we use less than five words as the threshold. Note five consecutive words are used as a rule of thumb here and it is not a standard rule anywhere.

Algorithmic Strategy:

The higher-level algorithmic strategy used for building the plagiarism detector is that we break the text file a (the file we want to check against plagiarism) we want to check for plagiarism into a string of 5 words. Then we insert all the 5 word strings into our CBF. We break the text file of

text file b into strings of 5 words as well and then we search for all those strings in our CBF and keep a count of the number of matches with text file a. We then divide the plagiarism count with the length of text file b to get the extent of plagiarism between text file a and b.

Note that I have used the **get_txt_into_list_of_words(url)** function provided in the assignment to first break the original text file into a list of words.

Python Implementation

```
def detect_plagiarism(text_a, text_b, phrase_size, fpr):
    """
    Detects the extent of plagiarism between two texts using a
    Counting Bloom Filter.

    This function inserts phrases from the first text into a
    Counting Bloom Filter and then checks
    each phrase from the second text against the filter to
    determine if it might be plagiarized.

    Parameters
    ----------
    text_a : str
        The source text where phrases will be inserted into the Counting Bloom Filter.
    text_b : str
        The text to check for plagiarism against the source text.
    phrase_size : int
        The number of consecutive words in a phrase for plagiarism checking.
    fpr : float
        The desired false positive rate for the Counting Bloom Filter.

    Returns
    -------
    float
        The ratio of phrases in the second text that were found
        in the Counting Bloom Filter, representing
        the extent of plagiarism.
    """

    # Tokenize both texts into phrases of specific phrase size
    phrases_a = [' '.join(text_a[i:i+phrase_size]) for i in range(len(text_a) - phrase_size + 1)]
    phrases_b = [' '.join(text_b[i:i+phrase_size]) for i in range(len(text_b) - phrase_size + 1)]

    # Initialize the Counting Bloom Filter with text A's phrases
    cbf = CountingBloomFilter(fpr=fpr, num_item=len(phrases_a))
    for phrase in phrases_a:
        cbf.insert(phrase)

    # Check for plagiarism by searching for text B's phrases in the filter
    plagiarism_count = 0
    for phrase in phrases_b:
        if cbf.search(phrase):
            plagiarism_count += 1

    # Calculate the extent of plagiarism
    extent_of_plagiarism = plagiarism_count / len(phrases_b)
    return extent_of_plagiarism*100
```

```
# Define the texts and parameters
text_version_1 = version_1  # Replace with the actual text content
text_version_2 = version_2  # Replace with the actual text content
text_version_3 = version_3   # Replace with the actual text content
phrase_size = 5  # The size of the phrase to check for plagiarism
fpr = 0.01  # The desired false positive rate

plagiarism_1_1 = detect_plagiarism(text_version_1, text_version_1, phrase_size, fpr)
plagiarism_1_2 = detect_plagiarism(text_version_1, text_version_2, phrase_size, fpr)
plagiarism_1_3 = detect_plagiarism(text_version_1, text_version_3, phrase_size, fpr)
plagiarism_2_1 = detect_plagiarism(text_version_2, text_version_1, phrase_size, fpr)
plagiarism_2_2 = detect_plagiarism(text_version_2, text_version_2, phrase_size, fpr)
plagiarism_2_3 = detect_plagiarism(text_version_2, text_version_3, phrase_size, fpr)
plagiarism_3_1 = detect_plagiarism(text_version_3, text_version_1, phrase_size, fpr)
plagiarism_3_2 = detect_plagiarism(text_version_3, text_version_2, phrase_size, fpr)
plagiarism_3_3 = detect_plagiarism(text_version_3, text_version_3, phrase_size, fpr)

print()
print(f"Plagiarism extent between version 1 and 1: {float(plagiarism_1_1)}%")
print(f"Plagiarism extent between version 1 and 2: {float(plagiarism_1_2)}%")
print(f"Plagiarism extent between version 1 and 3: {float(plagiarism_1_3)}%")

print()
print(f"Plagiarism extent between version 2 and 1: {float(plagiarism_2_1)}%")
print(f"Plagiarism extent between version 2 and 2: {float(plagiarism_2_2)}%")
print(f"Plagiarism extent between version 2 and 3: {float(plagiarism_2_3)}%")

print()
print(f"Plagiarism extent between version 3 and 1: {float(plagiarism_3_1)}%")
print(f"Plagiarism extent between version 3 and 2: {float(plagiarism_3_2)}%")
print(f"Plagiarism extent between version 3 and 3: {float(plagiarism_3_3)}%")
```

```
Plagiarism extent between version 1 and 1: 100.0%
Plagiarism extent between version 1 and 2: 36.99893579283434%
Plagiarism extent between version 1 and 3: 37.69590643274854%

Plagiarism extent between version 2 and 1: 37.55468842379094%
Plagiarism extent between version 2 and 2: 100.0%
Plagiarism extent between version 2 and 3: 37.859649122807014%

Plagiarism extent between version 3 and 1: 10.00354735721887%
Plagiarism extent between version 3 and 2: 10.263686886602814%
Plagiarism extent between version 3 and 3: 100.0%
```

Strengths of the Plagiarism detector:

1. Space Efficiency: CBFs are highly space-efficient compared to storing all phrases or words in a hash table or database. This is crucial when dealing with large texts or multiple documents.
2. Time Efficiency: Checking for the presence of a phrase in a CBF is much faster than searching through a list or database. This is particularly advantageous when dealing with large-scale plagiarism checks, as it drastically reduces the time required for analysis.
3. Handling Large Datasets: CBFs can handle large datasets efficiently, making them suitable for applications like plagiarism detection across extensive corpora or the internet.
4. Scalability: CBFs scale well with the size of the input data. As the amount of text increases, CBFs can maintain performance without a proportional increase in resource consumption.

Failure mode/ Limitations of the detector:

1. Our detector misses continuous sentences, meaning if we have a 25-word long plagiarised sentence our detector cannot tell if the extent of plagiarism is because of multiple 5 sentences long plagiarised sentences or a single 25-word long sentence.
2. Our detector cannot detect paraphrased sentences because it is based on detecting plagiarism for five-word long sentences (or specific phrase length in general). This is a problem because paraphrasing is also considered plagiarism without acknowledgment.

Hashing phrases and checking these hashes is computationally more efficient than comparing words one by one. Hashing allows for a constant-time lookup regardless of the size of the text, which is not possible with word-by-word comparison.

Other Algorithmic Strategies for Plagiarism detection

Comparing phrases using lists

A naive approach would be to create a list of phrases (of a specific number of words) for both files and then search from every phrase of list A with every phrase from list two and keep a count of matched phrases. Unlike the CBF approach, this strategy would not lead to false positives and is straightforward to apply.

Unfortunately, this approach has a time complexity of $O(n^2)$ if both the files have n words because we will be iterating through every phrase from list A to the entire list B phrases.

Python Implementation

```python
def list_method(text_a, text_b, phrase_size):
    """
    Defines the level of plagiarism between two texts with lists methods

    Input
    ----------
    text_a, text_b: list
        two texts to check level of plagiarism
    phrase_size: int
        number of words in the phrase

    Returns
    ----------
    None
    """

    phrases = []
    plagiarism = 0

    # Tokenize both texts into phrases of specific phrase size
    text_a = [' '.join(text_a[i:i+phrase_size]) for i in range(len(text_a) - phrase_size + 1)]
    text_b = [' '.join(text_b[i:i+phrase_size]) for i in range(len(text_b) - phrase_size + 1)]


    # Adding all elements to the list
    for i in range(len(text_a) - phrase_size + 1):
        phrases.append(text_a[i:i+phrase_size])

    # Searching if an element is in the list of text_2
    for i in range(len(text_b) - phrase_size + 1):
        if text_b[i:i+phrase_size] in phrases:
            plagiarism += 1

    plagiarism_level = (plagiarism / (len(text_b) - phrase_size + 1))
    return plagiarism_level * 100
```

```python
# Example texts for testing
version_1_identical = "This is a test text for plagiarism detection here.".split()
version_2_identical = "This is a test text for plagiarism detection here.".split()
# Identical to version_1_identical

version_1_different = "This is a test text for plagiarism detection here.".split()
version_2_different = "Completely different content with no common phrases here.".split()
# Completely different from version_1_different

# Define the phrase size for testing
phrase_size = 4  # Adjust this as needed

# Calculate plagiarism levels
plagiarism_level_identical = list_method(version_1_identical, version_2_identical, phrase_size)
plagiarism_level_different = list_method(version_1_different, version_2_different, phrase_size)

# Assertion for identical texts
expected_plagiarism_identical = 100
assert plagiarism_level_identical == expected_plagiarism_identical, \
f"Test failed for identical texts: Expected \
{expected_plagiarism_identical}%, got {plagiarism_level_identical}%"

# Assertion for different texts
expected_plagiarism_different = 0
assert plagiarism_level_different == expected_plagiarism_different, \
f"Test failed for different texts: Expected \
{expected_plagiarism_different}%, got {plagiarism_level_different}%"

print("All tests passed!")
```

```
All tests passed!
```

Jaccard Similarity Method

The Jaccard similarity (Author: Fatih Karabiber Ph.D. in Computer Engineering et al.) approach offers a more efficient alternative to the naive method of phrase comparison between two texts. Instead of comparing each phrase in Text A against every phrase in Text B, which results in $O(n^2)$ time complexity for texts with n phrases, the Jaccard approach first tokenizes each text into sets of phrases. This tokenization is linear, with a complexity of $O(n)$ for each text, where n represents the number of words in the text. Then, it simply calculates the intersection (common phrases) and the union (total unique phrases) of these sets. The complexity of these set operations in Python is generally $O(n + m)$, where n and m are the sizes of the sets. This makes the overall time complexity of the Jaccard similarity approach $O(n + m)$, which is significantly more efficient than $O(n^2)$ for large texts. Moreover, by calculating the ratio of the size of the intersection to the size of the union, Jaccard similarity provides a percentage score representing how similar the two texts are, without the risk of false positives inherent in the CBF approach. This method is not only faster but also provides a straightforward and intuitive metric for text similarity.

Python Implementation

```python
def jaccard_similarity_with_phrases(text_a, text_b, phrase_size):
    """
    Calculate the Jaccard Similarity between two texts, based on phrases of a specified size.

    Parameters:
    text_a (str): First text.
    text_b (str): Second text.
    phrase_size (int): The number of consecutive words in a phrase.

    Returns:
    float: Jaccard Similarity coefficient.
    """

    # Tokenize both texts into phrases of specific phrase size
    set1 = set([' '.join(text_a[i:i+phrase_size]) for i in range(len(text_a) - phrase_size + 1)])
    set2 = set([' '.join(text_b[i:i+phrase_size]) for i in range(len(text_b) - phrase_size + 1)])


    # Find the intersection and union of the two phrase sets
    intersection = set1.intersection(set2)
    union = set1.union(set2)

    # Calculate the Jaccard Similarity
    if not union:
        return 1.0  # Return 1 if both sets are empty

    jaccard_similarity_coefficient = float(len(intersection)) / len(union)

    return jaccard_similarity_coefficient*100
```

```python
# Example texts
version_1_i = "This is a test text for Jaccard similarity calculation."
version_2_i = "This is a test text for Jaccard similarity calculation."
# Identical to version_1_identical

version_1_d = "This is a test text for Jaccard similarity calculation."
version_2_d = "Completely different content with no common phrases."
# Completely different from version_1_different

# Define the phrase size
phrase_size = 5  # You can adjust this based on your needs

# Calculate similarities
similarity_identical = jaccard_similarity_with_phrases(version_1_i.split(), \
                                                       version_2_i.split(), phrase_size)
similarity_different = jaccard_similarity_with_phrases(version_1_d.split(), \
                                                       version_2_d.split(), phrase_size)

# Assertion for identical texts
expected_similarity_identical = 100
assert similarity_identical == expected_similarity_identical, \
f"Test failed for identical texts: Expected \
{expected_similarity_identical}%, got {similarity_identical}%"

# Assertion for different texts
expected_similarity_different = 0
assert similarity_different == expected_similarity_different, \
f"Test failed for different texts: Expected \
{expected_similarity_different}%, got {similarity_different}%"

print("All tests passed!")
```

```
All tests passed!
```

Experimental Analysis

Finally, to check the time complexity of the three methods described above, experiments were conducted to see how time scales with the number of words (n) in the two documents.

Based on the results from the experiments, we can see that the Jaccard method has a time complexity of O(n) as expected theoretically. This means it shows linear scaling behavior or if we double the number of words, we would expect the time to increase by roughly twice as well.

The list method shows the time complexity of $O(n^2)$. This means the scaling behavior is quadratic as we increase the number of words or if we increase the number of words by twice we would expect the time to increase by roughly four times. This is also evident from the visual inspection of the graph as when the number of words doubles from 2000 to 4000 we can see that time increases from roughly 0.1 seconds to 0.4 seconds which is a 4 times increase in time.

The CBF method shows the time complexity of O(n). Similar to the Jaccard method, this means it shows linear scaling and has the same interpretations.

Even though the Jaccard and CBF implementations show the same linear scaling, we can see from the final figure of all the methods together that the Jaccard method is faster than the CBF method as it seems to have a smaller constant value. So, we can prefer it over the CBF method but there is the nuance that unlike the CBF method we cannot control the upper bound of false positive rate in the Jaccard method.

In general, it is best to use the CBF method as compared to the Jaccard and List method because it shows linear scaling and we can control the false positive rate.

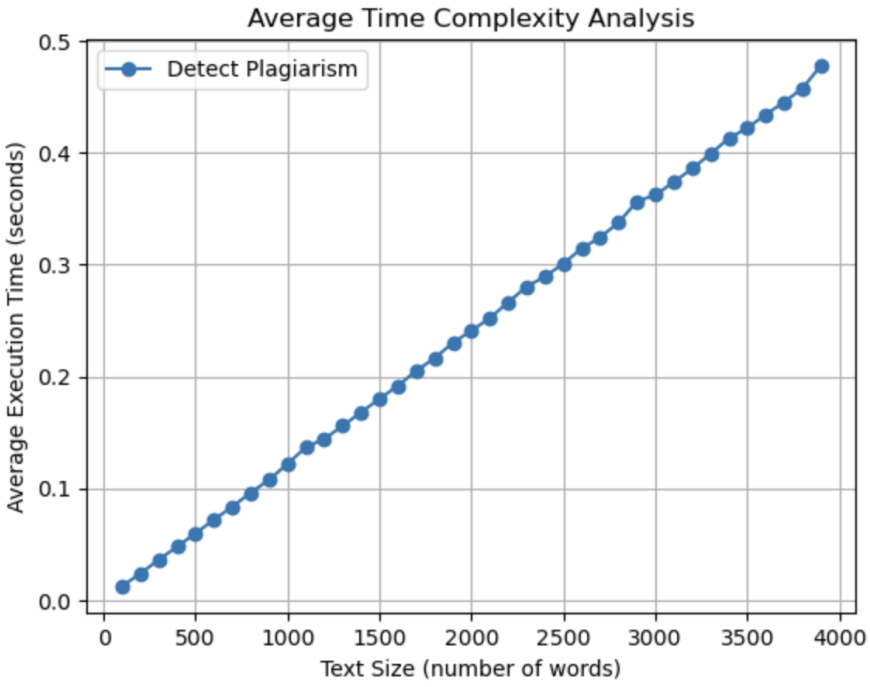


Figure 9: Plot showing linear scaling behavior for CBF-based plagiarism detection algorithm

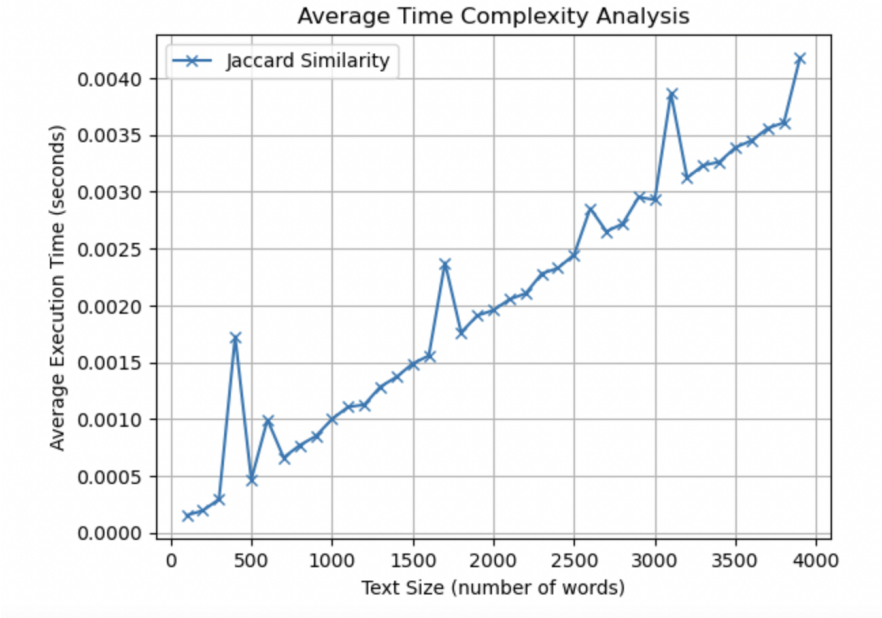


Figure 10: Plot showing linear scaling behavior for Jaccard plagiarism detection algorithm

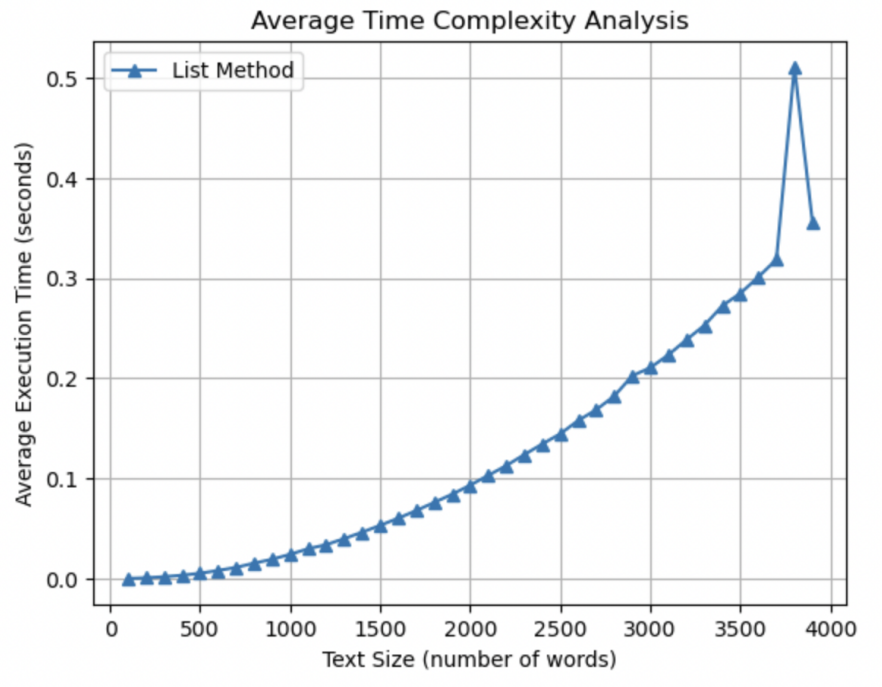


Figure 11: Plot showing quadratic scaling behavior for list-based plagiarism detection algorithm

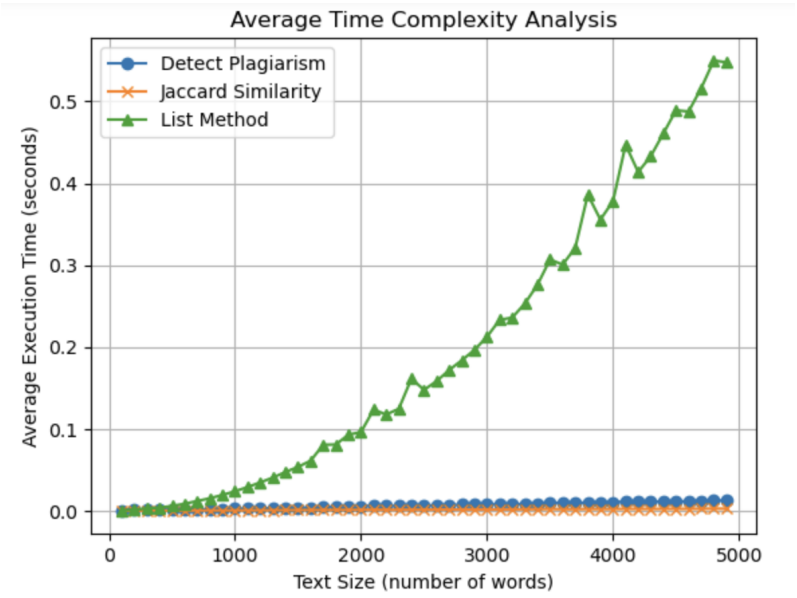


Figure 12: Plot showing scaling behavior for Jaccard, list, and, CBF-based plagiarism detector algorithm

Bibliography

Author: Fatih Karabiber Ph.D. in Computer Engineering, Fatih Karabiber Ph.D. in Computer Engineering, & Developer, V. J. S. (n.d.). Jaccard similarity. Learn Data Science - Tutorials, Books, Courses, and More. *https://www.learndatasci.com/glossary/jaccard-similarity/#:~:text=The%20Jaccard%20similarity%20measures%20the,of%20observations%20in%20either%20set*.

GeeksforGeeks. (2023, September 13). Bloom filters - introduction and implementation. GeeksforGeeks. *https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/*

Wikimedia Foundation. (2023, December 5). *Bloom filter*. Wikipedia. https://en.wikipedia.org/wiki/Bloom_filter

LO applications:

#PythonProgramming: I have only got a 4 on this LO so far so I follow my usual strategy for a good application. Firstly, all the code works as intended and takes input and returns output as intended. I have provided assertion statements to demonstrate that my code works as intended [Word Count: 49].

#CodeReadability: I have tried my best to not provide very long or too many inline comments based on previous feedback. I have also followed PEP8 conventions for functions and classes that I have created in this project and provided consistent docstrings everywhere [Word Count: 41].

#Professionalism: I have followed all the instructions of this assignment and provided a well-presented report. Previously, I have received feedback to provide titles for my flowcharts so I made sure that all my figures have titles and are well-presented. I have also provided a more comprehensive AI statement as compared to the last assignment [Word Count: 53].

#AlgoStratDataStructure: I have explained counting bloom filters as data structures in a simple manner and have created figures to explain them more intuitively. I have also briefly explained the algorithmic strategy of my plagiarism detector code and other methods at a high level [Word Count: 42].

#ComplexityAnalysis: In the assignment, I have explained the scaling growth of all the operations of the CBFs and also explained the scaling growth of different plagiarism detection methods that I have implemented and interpreted the scaling growth correctly. [Word Count: 37].

#ComputationalCritique: I have compared the theoretical and practical time complexity of three different plagiarism detection methods and gave a critical evaluation of their strengths and limitations. Previously I was asked to do a numerical test to validate my quadratic complexity and this time I have explained so in my experimental analysis [Word Count: 50].

HC applications:

#dataviz: I have effectively used Python to create a visualization for different time complexities of different algorithms used. I have made sure that all my visualizations have a title, caption, x, and y axes, and legends (whenever necessary) [Word Count: 37]

#organization: I have organized this report in an easy format by dividing the concepts into different sections and subsections. I have only the necessary code in the middle of the text to

maintain clarity and moved the rest to the appendix. I have also only provided graphs after the discussion of their content or how they are created so they are logically read [Word Count: 62].

#audience: I have tailored this report to be similar to the style of a chapter from the Cormen et al. textbook by using academically rigorous language but at the same time making the writing accessible for a beginner audience. I have provided links to resources in the bibliography so others can learn more about the topic if they want to (like for probability derivations) [Word Count: 63]

AI STATEMENT: *I used Grammarly to check for grammatical errors and used ChatGPT to paraphrase some parts of the assignment.*

Appendix (All CODE)

```python
#Q3 IMPLEMENTATION OF CBFs
import random
random.seed(10)
import math
class CountingBloomFilter:
    """
    A Counting Bloom Filter implementation that supports insertion, deletion,
    and membership query of elements.

    Attributes
    ----------
    fpr : float
        The desired false positive rate (between 0 and 1).
    num_item : int
        The expected number of items to be stored in the bloom filter.
    memory_size : int
        The size of the bloom filter's internal array, calculated
        based on the desired false positive rate
        and the expected number of items.
    num_hashfn : int
        The number of hash functions to be used, calculated based on
        the memory size and the number of items.
    array : list
        The internal array of the bloom filter where each index represents a counter.
    """

    def __init__(self, fpr, num_item):
        """
        Initialize the Counting Bloom Filter with a desired false
        positive rate and expected number of items.

        Parameters
        ----------
        fpr : float
            The desired false positive rate (between 0 and 1).
        num_item : int
            The expected number of items to be stored in the bloom filter.
        """

        self.fpr = fpr
        self.num_item = num_item

        # Calculate the size of the array needed to achieve the desired false positive rate.
        self.memory_size = math.ceil(-(num_item * math.log(fpr)) / (math.log(2) ** 2))

        # Calculate the optimal number of hash functions based on the size of the array
        #and the expected number of items.
        self.num_hashfn = math.ceil((self.memory_size / num_item) * math.log(2))

        # Initialize the internal array of the bloom filter with counters set to 0.
        self.array = [0] * self.memory_size

    def search(self, item):
        """
        Search for an item in the Counting Bloom Filter.

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

        Returns
        -------
        bool
            True if the item might be in the filter, False if
            the item is definitely not in the filter.
        """
        hash_values = self.hash_cbf(item)
        return all(self.array[hash_value] > 0 for hash_value in hash_values)

    def insert(self, item):
        """
        Insert an item into the Counting Bloom Filter.

        Parameters
        ----------
        item : str
            The item to insert.
        """
        hash_values = self.hash_cbf(item)
        for hash_value in hash_values:
            self.array[hash_value] += 1
```

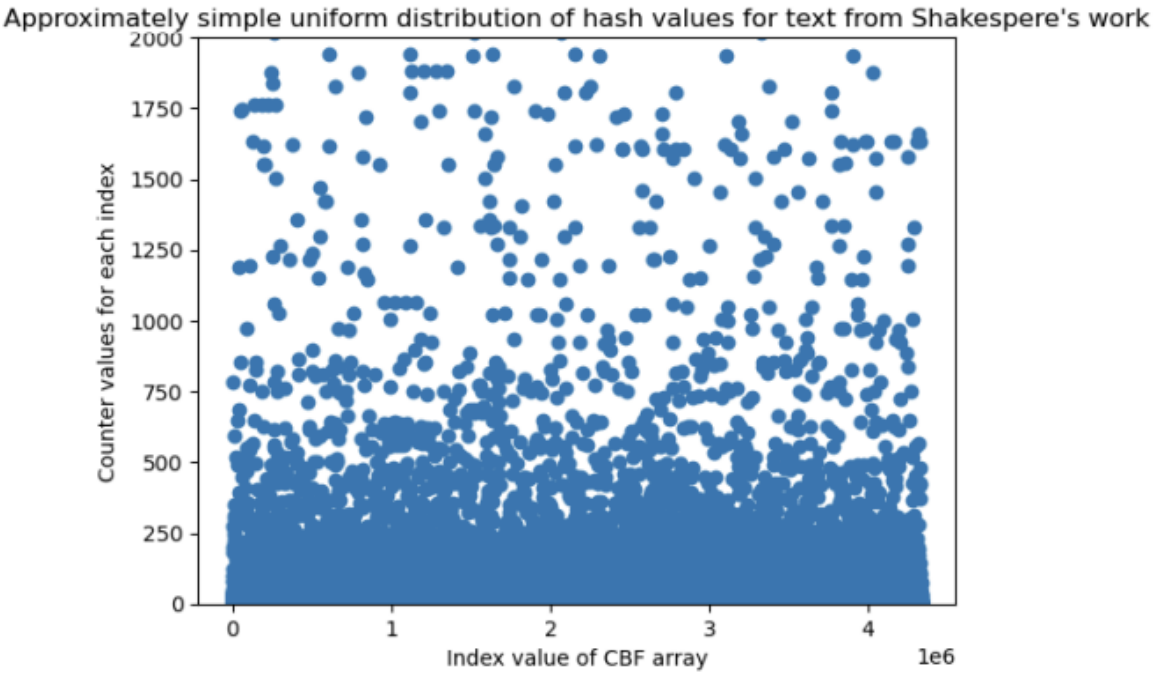
```python
def delete(self, item):
    """
    Remove an item from the Counting Bloom Filter, if it exists.

    Parameters
    ----------
    item : str
        The item to remove.
    """
    if self.search(item):
        hash_values = self.hash_cbf(item)
        for hash_value in hash_values:
            if self.array[hash_value] > 0:  # Ensure not to go below zero
                self.array[hash_value] -= 1
    else:
        print(f"{item} does not exist in the CBF")

def convert_to_int(self, string):
    """
    Generate a integer for a given string using ord()

    Parameters
    ----------
    string : str
        The string to convert to integer.

    Returns
    -------
    int
        The integer value of the string.
    """
    int_value = 211  # Prime number seed for reduced collision probability

    for i in range(len(string)):
        char = ord(string[i])
        int_value = int_value*char

    return int_value

def hash_cbf(self, item):
    """
    Compute a series of hash values for an item using a base hash function.

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

    Returns
    -------
    list
        A list of hash values for the given item.
    """
    hash_values = []
    for i in range(self.num_hashfn):

        # Double hashing: combine the base hash with a shifted version multiplied by the index
        hash_value = (self.convert_to_int(item) + i * \
                      (self.convert_to_int(item) << 5)) % self.memory_size
        hash_values.append(hash_value)

    return hash_values
```

Function to convert url text file into list of words

```
#Note: This is the code from the assignement instructions
from requests import get
def get_txt_into_list_of_words(url):
    '''Cleans the text data
    Input
    ----------
    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.
    '''
    bad_chars = [';', ',', '.', '?', '!', '_', '[', ']', '(', ')', '*']
    data = get(url).text
    data = ''.join(c for c in data if c not in bad_chars)
    data_without_newlines = ''.join(c if (c not in ['\n', '\r', '\t']) else " " for c in data)
    data_just_words = [word for word in data_without_newlines.split(" ") if word != ""]
    data_just_words_lower_case = [word.lower() for word in data_just_words]
    return data_just_words_lower_case
```

Testing Uniform distribution of the hash functions

```
: #Q3(A.) Uniform distribution of the hash function
import matplotlib.pyplot as plt
import math
import random
random.seed(10)
def plot_cbf_distribution(url):
    """
    Visualize the distribution of hash values in a Counting Bloom Filter (CBF)
    after inserting elements from a text file.

    This function takes a URL pointing to a text file, processes the text
    into a list of words, and then inserts these words into a CBF. It plots
    the distribution of the internal counters of the CBF, giving a visual representation
    of how uniformly the hash function distributes the elements across the array.

    Parameters
    ----------
    url : str
        URL of the text file from which words are read and inserted into
        the CBF. The text file should contain a large body of text for a meaningful
        distribution visualization.

    Notes
    -----
    - The function assumes the presence of a `get_txt_into_list_of_words`
    function that takes a URL and returns a list of words from the text file.
    - The CBF is initialized with a pre-defined false positive rate and
    size based on the length of the word list.
    - The `plt.ylim` is set to (0, 2000) for visualization purposes and
    might need adjustment based on the actual range of counter values.

    """

    data = get_txt_into_list_of_words(url)
    cbf = CountingBloomFilter(0.1,len(data))

    previous_cbf = None
    for i in range(len(data)):
        previous_cbf = cbf.insert(data[i])

    plt.plot(list(range(cbf.memory_size)),cbf.array,'o')
    plt.xlabel("Index value of CBF array")
    plt.ylabel("Counter values for each index")
    plt.ylim(0,2000)
    plt.title("Approximately simple uniform distribution of hash values for text from Shakespere's work")
    plt.show()

url = ("https://gist.githubusercontent.com/raquelhr/78f66877813825dc344efefd"\
       +"c684a5d6/raw/361a40e4cd22cb6025e1fb2baca3bf7e166b2ec6/")
plot_cbf_distribution(url)
```


Approximately simple uniform distribution of hash values for text from Shakespere's work
Counter values for each index
2000
1750
1500
1250
1000
750
500
250
0
0
1
2
3
4
Index value of CBF array
1e6

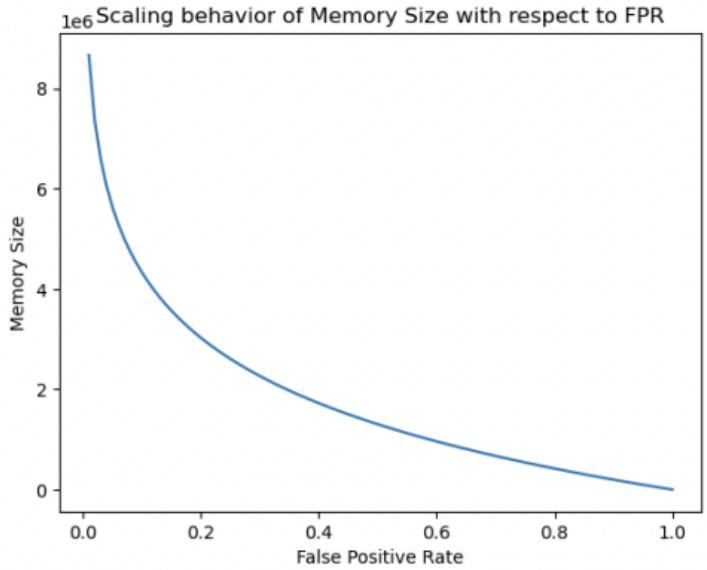
TESTING THE CBF IMPLEMENTATION

```
#Q3(C.) TESTING THE CBF IMPLEMENTATION
random.seed(10)
cbf = CountingBloomFilter(fpr = 0.01, num_item = 3)

# Test 1: Checks Insertion and Deletion Functionality
L1 = ["Mango","Orange","Banana"]

for i in L1:
    cbf.insert(i)

for i in L1:
    cbf.delete(i)
    cbf

for i in L1:
    assert cbf.search(i) == False
print("All assertions were passed for Test case 1")


# Test 2: Checks Deletion Functionality
L2 = ["Big","Medium","Small"]

for i in L2:
    cbf.insert(i)

for i in L2:
    cbf.delete(i)

for i in L2:
    cbf.delete(i)


# Test 3:  Checks Insertion and Search Functionality
L1 = ["CS","NS","BS"]

for i in L1:
    cbf.insert(i)

for i in L1:
    assert cbf.search(i) == True

assert cbf.search("AH") == False

print("All assertions were passed for Test case 3")
```

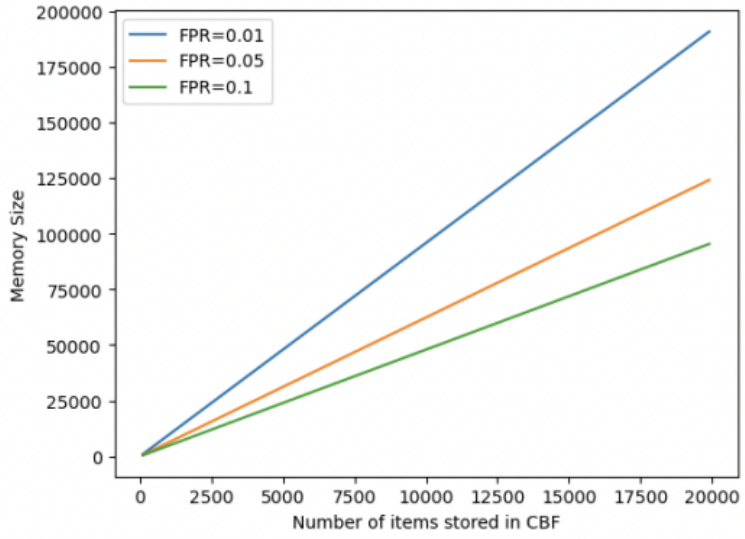
```
All assertions were passed for Test case 1
Big does not exist in the CBF
Medium does not exist in the CBF
Small does not exist in the CBF
All assertions were passed for Test case 3
```

VERIFYING THE EFFECTIVENESS OF THE CBF

```python
#Q4(a)
random.seed(10)
#shakespere text
all_text = get_txt_into_list_of_words(url)

def plot_memory_vs_fpr(data):
    """
    Plot the scaling behavior of memory size with respect to the false positive rate (FPR)
    for a Counting Bloom Filter.

    This function creates a series of Counting Bloom Filters with varying FPRs and
    a fixed number of items. It then plots the memory size of each filter against its FPR.

    Parameters
    ----------
    data : list
        The list of items to be inserted into the Counting Bloom Filter.
        The length of this list determines the number of items in each filter.
    """
    fpr = []  # List to store false positive rates
    m_size = []  # List to store memory sizes corresponding to each FPR
    num_item = len(data)  # Number of items to insert into the filter

    # Loop to create Counting Bloom Filters with varying FPRs and record their memory sizes
    for i in range(1, 101):
        fpr_value = i / 100  # Calculate the false positive rate
        cbf = CountingBloomFilter(fpr=fpr_value, num_item=num_item)  # Create a CBF with the given FPR
        fpr.append(float(fpr_value))  # Append the FPR to the list
        m_size.append(cbf.memory_size)  # Append the memory size to the list

    # Plotting the results
    plt.plot(fpr, m_size)
    plt.title('Scaling behavior of Memory Size with respect to FPR')
    plt.xlabel('False Positive Rate')
    plt.ylabel('Memory Size')
    plt.show()

plot_memory_vs_fpr(all_text)
```



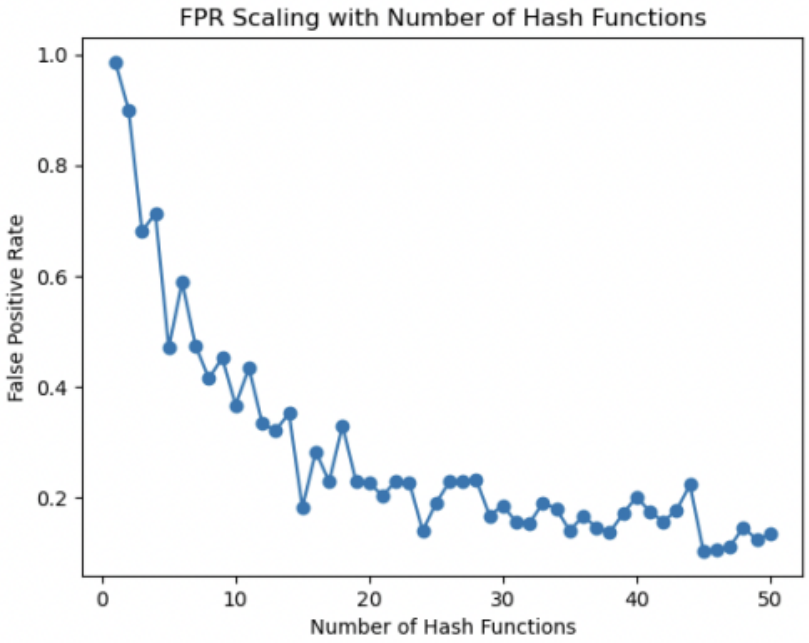
```python
#Q4(b)
random.seed(10)
def plot_memory_vs_num_items(fpr_list):
    """
    Plot the scaling behavior of the memory size of a Counting Bloom Filter with
    respect to the number of items stored,
    for different false positive rates (FPRs).

    This function generates Counting Bloom Filters for a range of numbers
    of items at each specified FPR. It then plots the memory size required for
    each Bloom Filter against the number of items for each FPR.

    Parameters
    ----------
    fpr_list : list
        A list of false positive rates (FPRs) to test. Each FPR will be represented
        as a separate line in the plot.
    """
    for fpr in fpr_list:
        m_size = []  # List to store memory sizes for each number of items
        num_items = []  # List to store different numbers of items

        # Loop over a range of numbers of items
        for n in range(100, 20000, 100):
            num_items.append(n)  # Append the number of items
            cbf = CountingBloomFilter(fpr, num_item=n)  # Create a CBF with
            #the current number of items and FPR
            m_size.append(cbf.memory_size)  # Append the memory size of the CBF

        # Plot memory size against number of items for the current FPR
        plt.plot(num_items, m_size, label=f'FPR={round(fpr, 3)}')

    # Add labels, legend, and show the plot
    plt.xlabel('Number of items stored in CBF')
    plt.ylabel('Memory Size')
    plt.legend()
    plt.show()

# Example usage
plot_memory_vs_num_items([0.01, 0.05, 0.1])
```



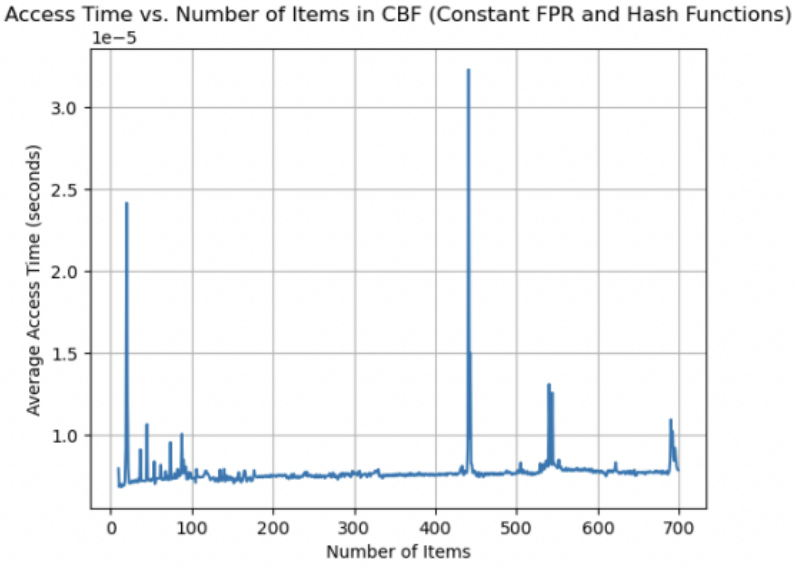
```
: #Q4(C) SUBCLASS OF CBF TO BE USED FOR Q4(C)
  import math
  random.seed(10)
  class CBF2:
      """
      A Counting Bloom Filter implementation that supports insertion, deletion,
      and membership queries for elements. This variant of Bloom Filter uses a counter
      for each position in the internal array to allow deletion of items.

      Attributes
      ----------
      num_item : int
          The number of items expected to be stored in the bloom filter.
      memory_size : int
          The size of the bloom filter's internal array, indicating the total number of counters.
      num_hashfn : int
          The number of hash functions to be used in the bloom filter.
      array : list
          The internal array of the bloom filter where each index represents a counter.
      """

      def __init__(self, num_item, num_hashfn, memory_size):
          """
          Initialize the Counting Bloom Filter with specified parameters.

          Parameters
          ----------
          num_item : int
              The expected number of items to be stored in the bloom filter.
          num_hashfn : int
              The number of hash functions to be used for the bloom filter.
          memory_size : int
              The size of the bloom filter's internal array (number of counters).
          """

          self.num_item = num_item
          self.num_hashfn = num_hashfn
          self.memory_size = memory_size
          self.array = [0] * self.memory_size

      def search(self, item):
          """
          Search for an item in the Counting Bloom Filter.

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

          Returns
          -------
          bool
              True if the item might be in the filter, False if the item is definitely not in the filter.
          """
          hash_values = self.hash_cbf(item)
          return all(self.array[hash_value] > 0 for hash_value in hash_values)

      def insert(self, item):
          """
          Insert an item into the Counting Bloom Filter.

          Parameters
          ----------
          item : str
              The item to insert.
          """
          hash_values = self.hash_cbf(item)
          for hash_value in hash_values:
              self.array[hash_value] += 1

      def delete(self, item):
          """
          Remove an item from the Counting Bloom Filter, if it exists.

          Parameters
          ----------
          item : str
              The item to remove.
          """
          if self.search(item):
              hash_values = self.hash_cbf(item)
              for hash_value in hash_values:
```

```
            if self.array[hash_value] > 0:  # Ensure not to go below zero
                self.array[hash_value] -= 1
        else:
            print(f"{item} does not exist in the CBF")

    def hashfn(self, string):
        """
        Generate a hash for a given string.

        Parameters
        ----------
        string : str
            The string to hash.

        Returns
        -------
        int
            The hash value of the string.
        """
        hash = 9973  # Prime number seed for reduced collision probability
        for i in range(len(string)):
            char = ord(string[i])
            hash = ((hash << 5) + hash) + char  # Bit manipulation for hash calculation
        return hash

    def hash_cbf(self, item):
        """
        Compute a series of hash values for an item using a base hash function.

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

        Returns
        -------
        list
            A list of hash values for the given item.
        """
        hash_values = []
        for i in range(self.num_hashfn):
            # Double hashing: combine the base hash with a shifted version multiplied by the index
            hash_value = (self.hashfn(item) + i * (self.hashfn(item) << 5)) % self.memory_size
            hash_values.append(hash_value)
        return hash_values
```

```python
#Q4(c)
random.seed(10)
def measure_fpr(cbf, test_items):
    """
    Calculate the false positive rate (FPR) for a given Counting Bloom Filter.

    This function tests a set of items against the
    Bloom Filter and calculates the proportion of these items that
    are falsely identified as being in the filter (false positives).

    Parameters
    ----------
    cbf : CountingBloomFilter
        The Counting Bloom Filter to be tested.
    test_items : list
        A list of items to test against the Bloom Filter.
        These items should ideally not be in the filter
        to accurately measure false positives.

    Returns
    -------
    float
        The false positive rate (FPR) calculated as the number
        of false positives divided by the total number of test items.
    """

    false_positives = sum(1 for item in test_items if cbf.search(item))
    fpr = false_positives / len(test_items)
    return fpr

# Experiment setup and execution
num_items = len(all_text[0:2000])  # Number of items to insert
test_items = all_text[0:2000]  # Items not in the data_set

hash_functions_list = []  # List to store numbers of hash functions
fpr_list = []  # List to store corresponding false positive rates

for num_hashfn in range(1, 51):
    """
    Perform an experiment to measure how the false positive rate
    (FPR) of a Counting Bloom Filter scales with the number of hash functions.

    This loop creates Counting Bloom Filters with a varying number of hash functions.
    Each filter is tested with a fixed set of items to measure the FPR.

    The range for the number of hash functions is set from 1 to 50.
    """

    cbf = CBF2(num_item=num_items, num_hashfn=num_hashfn, memory_size=1000*num_hashfn)

    for i in range(num_items):
        cbf.insert("item" + str(i))

    fpr = measure_fpr(cbf, test_items)
    hash_functions_list.append(num_hashfn)
    fpr_list.append(fpr)

# Plot the results
plt.plot(hash_functions_list, fpr_list, marker='o')
plt.title('FPR Scaling with Number of Hash Functions')
plt.xlabel('Number of Hash Functions')
plt.ylabel('False Positive Rate')
plt.show()
```


FPR Scaling with Number of Hash Functions
False Positive Rate
1.0
0.8
0.6
0.4
0.2
0
10
20
30
40
50
Number of Hash Functions

```
)]: #Q4(d)
import random
import matplotlib.pyplot as plt
import numpy as np
import time
random.seed(10)

def access_time(num_item, fpr, num_hashfn):
    """
    Calculate the average access time for searching elements in a Counting Bloom Filter.

    Parameters
    ----------
    num_item : int
        The number of items to be stored in the CBF.
    fpr : float
        The false positive rate for the CBF.
    num_hashfn : int
        The number of hash functions used by the CBF.

    Returns
    -------
    float
        The average access time per search operation in seconds.
    """
    # Initialize CBF with a constant number of hash functions
    cbf = CountingBloomFilter(fpr=fpr, num_item=num_item)

    # Insert items into the CBF
    for i in range(num_item):
        cbf.insert(f"item{i}")

    # Time the search operations
    start_time = time.time()
    for i in range(num_item):
        cbf.search(f"item{i}")
    end_time = time.time()

    # Calculate the average access time per search operation
    total_time = end_time - start_time
    average_time = total_time / num_item
    return average_time

# Experiment setup
fpr = 0.1  # Example false positive rate
num_hashfn = 5  # Constant number of hash functions
num_items_list = np.arange(10, 700, 1)  # Varying number of items
access_times = []

# Measure the access time for different numbers of items
for num_items in num_items_list:
    time_taken = access_time(num_items, fpr, num_hashfn)
    access_times.append(time_taken)

# Plot the results
plt.plot(num_items_list, access_times)
plt.title('Access Time vs. Number of Items in CBF (Constant FPR and Hash Functions)')
plt.xlabel('Number of Items')
plt.ylabel('Average Access Time (seconds)')
plt.grid(True)
plt.show()
```



DATA FOR PLAGIARISM DEDECTOR

```
0]: #Q5
    #data to test for plagiarism detector
    url_version_1 = 'https://bit.ly/39MurYb'
    url_version_2 = 'https://bit.ly/3we1QCp'
    url_version_3 = 'https://bit.ly/3vUecRn'

    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)
```

```python
def detect_plagiarism(text_a, text_b, phrase_size, fpr):
    """
    Detects the extent of plagiarism between two texts using a
    Counting Bloom Filter.

    This function inserts phrases from the first text into a
    Counting Bloom Filter and then checks
    each phrase from the second text against the filter to
    determine if it might be plagiarized.

    Parameters
    ----------
    text_a : str
        The source text where phrases will be inserted into the Counting Bloom Filter.
    text_b : str
        The text to check for plagiarism against the source text.
    phrase_size : int
        The number of consecutive words in a phrase for plagiarism checking.
    fpr : float
        The desired false positive rate for the Counting Bloom Filter.

    Returns
    -------
    float
        The ratio of phrases in the second text that were found
        in the Counting Bloom Filter, representing
        the extent of plagiarism.
    """

    # Tokenize both texts into phrases of specific phrase size
    phrases_a = [' '.join(text_a[i:i+phrase_size]) for i in range(len(text_a) - phrase_size + 1)]
    phrases_b = [' '.join(text_b[i:i+phrase_size]) for i in range(len(text_b) - phrase_size + 1)]

    # Initialize the Counting Bloom Filter with text A's phrases
    cbf = CountingBloomFilter(fpr=fpr, num_item=len(phrases_a))
    for phrase in phrases_a:
        cbf.insert(phrase)

    # Check for plagiarism by searching for text B's phrases in the filter
    plagiarism_count = 0
    for phrase in phrases_b:
        if cbf.search(phrase):
            plagiarism_count += 1

    # Calculate the extent of plagiarism
    extent_of_plagiarism = plagiarism_count / len(phrases_b)
    return extent_of_plagiarism*100
```

```
: # Define the texts and parameters
  text_version_1 = version_1  # Replace with the actual text content
  text_version_2 = version_2  # Replace with the actual text content
  text_version_3 = version_3   # Replace with the actual text content
  phrase_size = 5  # The size of the phrase to check for plagiarism
  fpr = 0.01  # The desired false positive rate

  plagiarism_1_1 = detect_plagiarism(text_version_1, text_version_1, phrase_size, fpr)
  plagiarism_1_2 = detect_plagiarism(text_version_1, text_version_2, phrase_size, fpr)
  plagiarism_1_3 = detect_plagiarism(text_version_1, text_version_3, phrase_size, fpr)
  plagiarism_2_1 = detect_plagiarism(text_version_2, text_version_1, phrase_size, fpr)
  plagiarism_2_2 = detect_plagiarism(text_version_2, text_version_2, phrase_size, fpr)
  plagiarism_2_3 = detect_plagiarism(text_version_2, text_version_3, phrase_size, fpr)
  plagiarism_3_1 = detect_plagiarism(text_version_3, text_version_1, phrase_size, fpr)
  plagiarism_3_2 = detect_plagiarism(text_version_3, text_version_2, phrase_size, fpr)
  plagiarism_3_3 = detect_plagiarism(text_version_3, text_version_3, phrase_size, fpr)

  print()
  print(f"Plagiarism extent between version 1 and 1: {float(plagiarism_1_1)}%")
  print(f"Plagiarism extent between version 1 and 2: {float(plagiarism_1_2)}%")
  print(f"Plagiarism extent between version 1 and 3: {float(plagiarism_1_3)}%")

  print()
  print(f"Plagiarism extent between version 2 and 1: {float(plagiarism_2_1)}%")
  print(f"Plagiarism extent between version 2 and 2: {float(plagiarism_2_2)}%")
  print(f"Plagiarism extent between version 2 and 3: {float(plagiarism_2_3)}%")

  print()
  print(f"Plagiarism extent between version 3 and 1: {float(plagiarism_3_1)}%")
  print(f"Plagiarism extent between version 3 and 2: {float(plagiarism_3_2)}%")
  print(f"Plagiarism extent between version 3 and 3: {float(plagiarism_3_3)}%")
```

```
Plagiarism extent between version 1 and 1: 100.0%
Plagiarism extent between version 1 and 2: 36.99893579283434%
Plagiarism extent between version 1 and 3: 37.69590643274854%

Plagiarism extent between version 2 and 1: 37.55468842379094%
Plagiarism extent between version 2 and 2: 100.0%
Plagiarism extent between version 2 and 3: 37.859649122807014%

Plagiarism extent between version 3 and 1: 10.00354735721887%
Plagiarism extent between version 3 and 2: 10.263686886602814%
Plagiarism extent between version 3 and 3: 100.0%
```

JACCARD METHOD IMPLEMENTATION

```python
def jaccard_similarity_with_phrases(text_a, text_b, phrase_size):
    """
    Calculate the Jaccard Similarity between two texts, based on phrases of a specified size.

    Parameters:
    text_a (str): First text.
    text_b (str): Second text.
    phrase_size (int): The number of consecutive words in a phrase.

    Returns:
    float: Jaccard Similarity coefficient.
    """

    # Tokenize both texts into phrases of specific phrase size
    set1 = set([' '.join(text_a[i:i+phrase_size]) for i in range(len(text_a) - phrase_size + 1)])
    set2 = set([' '.join(text_b[i:i+phrase_size]) for i in range(len(text_b) - phrase_size + 1)])


    # Find the intersection and union of the two phrase sets
    intersection = set1.intersection(set2)
    union = set1.union(set2)

    # Calculate the Jaccard Similarity
    if not union:
        return 1.0  # Return 1 if both sets are empty

    jaccard_similarity_coefficient = float(len(intersection)) / len(union)

    return jaccard_similarity_coefficient*100
```

```python
# Example texts
version_1_i = "This is a test text for Jaccard similarity calculation."
version_2_i = "This is a test text for Jaccard similarity calculation."
# Identical to version_1_identical

version_1_d = "This is a test text for Jaccard similarity calculation."
version_2_d = "Completely different content with no common phrases."
# Completely different from version_1_different

# Define the phrase size
phrase_size = 5  # You can adjust this based on your needs

# Calculate similarities
similarity_identical = jaccard_similarity_with_phrases(version_1_i.split(), \
                                                       version_2_i.split(), phrase_size)
similarity_different = jaccard_similarity_with_phrases(version_1_d.split(), \
                                                       version_2_d.split(), phrase_size)

# Assertion for identical texts
expected_similarity_identical = 100
assert similarity_identical == expected_similarity_identical, \
f"Test failed for identical texts: Expected \
{expected_similarity_identical}%, got {similarity_identical}%"

# Assertion for different texts
expected_similarity_different = 0
assert similarity_different == expected_similarity_different, \
f"Test failed for different texts: Expected \
{expected_similarity_different}%, got {similarity_different}%"

print("All tests passed!")
```

```
All tests passed!
```

LIST METHOD IMPLEMENTATION

```
def list_method(text_a, text_b, phrase_size):
    """
    Defines the level of plagiarism between two texts with lists methods

    Input
    ----------
    text_a, text_b: list
        two texts to check level of plagiarism
    phrase_size: int
        number of words in the phrase

    Returns
    ----------
    None
    """

    phrases = []
    plagiarism = 0

    # Tokenize both texts into phrases of specific phrase size
    text_a = [' '.join(text_a[i:i+phrase_size]) for i in range(len(text_a) - phrase_size + 1)]
    text_b = [' '.join(text_b[i:i+phrase_size]) for i in range(len(text_b) - phrase_size + 1)]


    # Adding all elements to the list
    for i in range(len(text_a) - phrase_size + 1):
        phrases.append(text_a[i:i+phrase_size])

    # Searching if an element is in the list of text_2
    for i in range(len(text_b) - phrase_size + 1):
        if text_b[i:i+phrase_size] in phrases:
            plagiarism += 1

    plagiarism_level = (plagiarism / (len(text_b) - phrase_size + 1))
    return plagiarism_level * 100
```

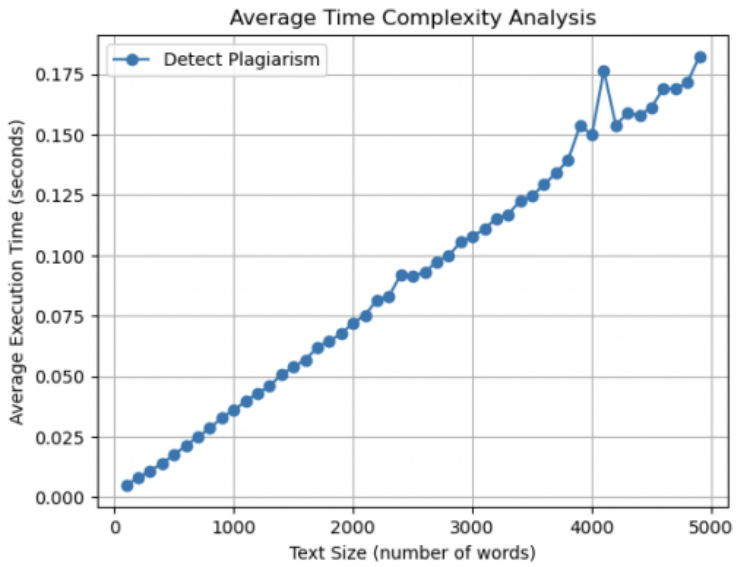
```
# Example texts for testing
version_1_identical = "This is a test text for plagiarism detection here.".split()
version_2_identical = "This is a test text for plagiarism detection here.".split()
# Identical to version_1_identical

version_1_different = "This is a test text for plagiarism detection here.".split()
version_2_different = "Completely different content with no common phrases here.".split()
# Completely different from version_1_different

# Define the phrase size for testing
phrase_size = 4  # Adjust this as needed

# Calculate plagiarism levels
plagiarism_level_identical = list_method(version_1_identical, version_2_identical, phrase_size)
plagiarism_level_different = list_method(version_1_different, version_2_different, phrase_size)

# Assertion for identical texts
expected_plagiarism_identical = 100
assert plagiarism_level_identical == expected_plagiarism_identical, \
f"Test failed for identical texts: Expected \
{expected_plagiarism_identical}%, got {plagiarism_level_identical}%"

# Assertion for different texts
expected_plagiarism_different = 0
assert plagiarism_level_different == expected_plagiarism_different, \
f"Test failed for different texts: Expected \
{expected_plagiarism_different}%, got {plagiarism_level_different}%"

print("All tests passed!")
```

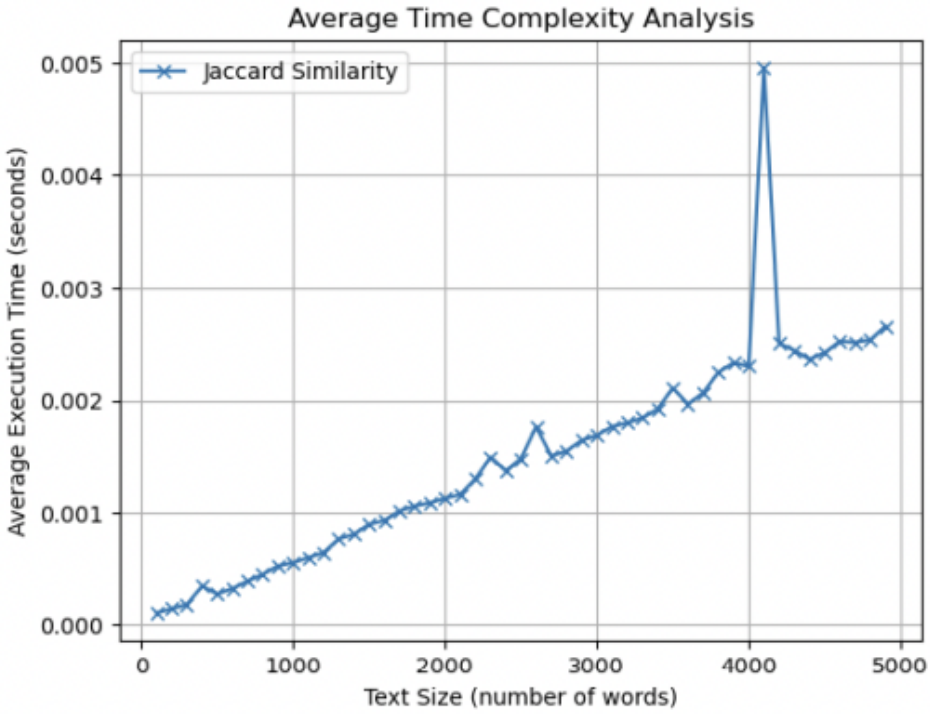
```
All tests passed!
```

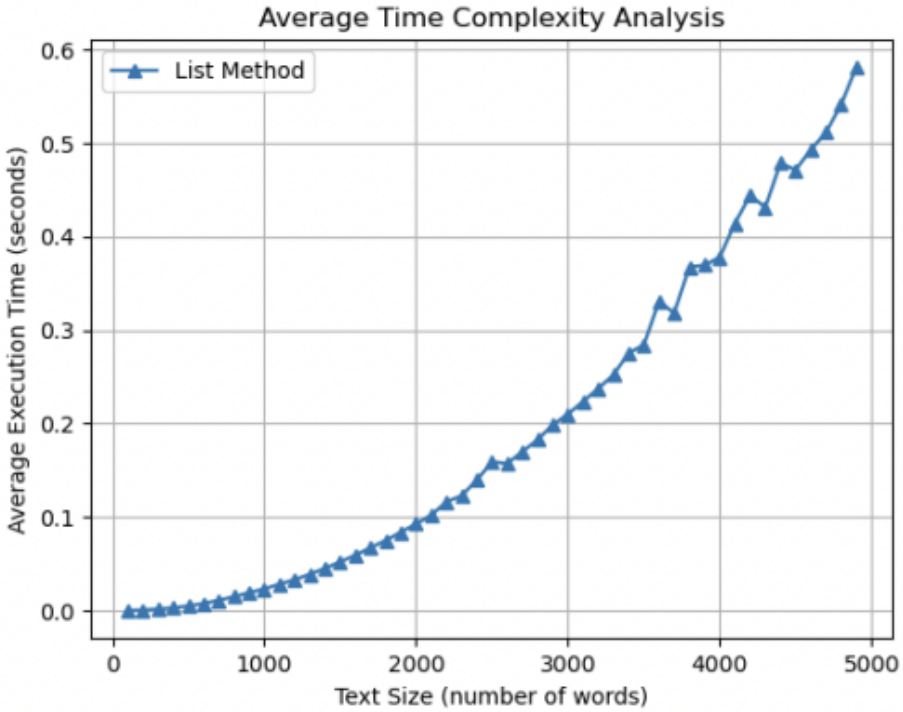
```
#Q5
#data to test for plagiarism detector
url_version_1 = 'https://bit.ly/39MurYb'
url_version_2 = 'https://bit.ly/3we1QCp'
url_version_3 = 'https://bit.ly/3vUecRn'

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)
```

```python
# We already have the functions defined in the previous cells,
#so now we will proceed with the analysis.
# First, we need to define a function to measure the execution
#time of another function.

import time
import random
random.seed(10)
import string
import matplotlib.pyplot as plt


# Define the function to generate random text of given size
def generate_text(size):
    """
    Generate a random text of a given size.

    This function creates a list of random lowercase letters of the specified length.

    Parameters
    ----------
    size : int
        The number of characters in the generated text.

    Returns
    -------
    list
        A list of random lowercase letters of length 'size'.
    """
    return [random.choice(string.ascii_lowercase) for _ in range(size)]

# Define a function to measure the execution time of a function
def measure_execution_time(func, *args):
    """
    Measure the execution time of a specified function.

    This function times how long it takes for a given function to
    execute and returns the execution time.

    Parameters
    ----------
    func : function
        The function to measure the execution time of.
    *args
        Arguments to be passed to the function being measured.

    Returns
    -------
    float
        The time taken to execute the function in seconds.
    result
        The return value of the function being measured.
    """
    start_time = time.time()
    result = func(*args)
    execution_time = time.time() - start_time
    return execution_time, result

# Now let's perform the time complexity analysis for each method
# Define text sizes for the analysis
text_sizes = [i for i in range(100, 5000, 100)]
phrase_size = 6
fpr = 0.01

# Lists to hold execution times for each algorithm
times_detect_plagiarism = []
times_jaccard_similarity = []
times_list_method = []

# Perform analysis
for size in text_sizes:
    # Initialize accumulators for average calculation
    total_time_plagiarism = 0
    total_time_jaccard = 0
    total_time_list_method = 0

    # Define number of runs to calculate average runtime
    num_runs = 5

    for _ in range(num_runs):
        text_1 = generate_text(size)
```

```
        # Time the detect_plagiarism function
        exec_time, _ = measure_execution_time(detect_plagiarism, \
                                              text_1, text_2, phrase_size, fpr)
        total_time_plagiarism += exec_time

        # Time the jaccard_similarity_with_phrases function
        exec_time, _ = measure_execution_time(jaccard_similarity_with_phrases, \
                                              text_1, text_2, phrase_size)
        total_time_jaccard += exec_time

        # Time the list_method function
        exec_time, _ = measure_execution_time(list_method, text_1, text_2, phrase_size)
        total_time_list_method += exec_time

    # Calculate the average time
    avg_time_plagiarism = total_time_plagiarism / num_runs
    avg_time_jaccard = total_time_jaccard / num_runs
    avg_time_list_method = total_time_list_method / num_runs

    # Append the average times to the lists
    times_detect_plagiarism.append(avg_time_plagiarism)
    times_jaccard_similarity.append(avg_time_jaccard)
    times_list_method.append(avg_time_list_method)


plt.plot(text_sizes, times_detect_plagiarism, label='Detect Plagiarism', marker='o')
plt.xlabel('Text Size (number of words)')
plt.ylabel('Average Execution Time (seconds)')
plt.title('Average Time Complexity Analysis')
plt.legend()
plt.grid(True)
plt.show()

plt.plot(text_sizes, times_jaccard_similarity, label='Jaccard Similarity', marker='x')
plt.xlabel('Text Size (number of words)')
plt.ylabel('Average Execution Time (seconds)')
plt.title('Average Time Complexity Analysis')
plt.legend()
plt.grid(True)
plt.show()

plt.plot(text_sizes, times_list_method, label='List Method', marker='^')
plt.xlabel('Text Size (number of words)')
plt.ylabel('Average Execution Time (seconds)')
plt.title('Average Time Complexity Analysis')
plt.legend()
plt.grid(True)
plt.show()
```




Average Time Complexity Analysis
Jaccard Similarity
Average Execution Time (seconds)
0.000
0.001
0.002
0.003
0.004
0.005
0
1000
2000
3000
4000
5000
Text Size (number of words)


Average Time Complexity Analysis
List Method
Average Execution Time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
1000
2000
3000
4000
5000
Text Size (number of words)

```
plt.plot(text_sizes, times_detect_plagiarism, label='Detect Plagiarism', marker='o')
plt.plot(text_sizes, times_jaccard_similarity, label='Jaccard Similarity', marker='x')
plt.plot(text_sizes, times_list_method, label='List Method', marker='^')
plt.xlabel('Text Size (number of words)')
plt.ylabel('Average Execution Time (seconds)')
plt.title('Average Time Complexity Analysis')
plt.legend()
plt.grid(True)
plt.show()
```