Project 3 - Final Computational Application

Shaghayegh Moradirad

Minerva University

CS110

This report provides an overview of counting bloom filters, including their history, advantages, a description of hash functions and their optimization for memory size, false positive rate, and the number of hash functions. Additionally, an example of the implementation of counting bloom filters for plagiarism detection is included.

1. Write a summary of the CBFs as an indexing technique, with their underlying data structure and hashing functionality. This is, to list all the operations they support. For all CBF operations, including an analysis of their asymptotic order of growth.

Counting Bloom Filters:

- Underlying data structure (All the supported operations)
- How it is used as an indexing technique
- Hashing functionality

A Counting Bloom Filter is a variation of a Bloom Filter; thus we should first, learn about the Bloom filter. A Bloom filter is a probabilistic data structure that is used to test whether an element is a member of a set or not.

Probabilistic data structures provide an approximation of the answer or a way to approximate the answer while using as little memory as possible within a reasonable running time.
For example, suppose we have a set S={1,3,4} and we want to see if the element p=5 is in this set or not.

- If we use the linear search: time-complexity at the worst case is linear, O(n) since we need to go through all of the elements to check them if they are equal to p or not.

- If we try to be more efficient and use a Hash set, then we can look up efficiently with an average case[1] time-complexity of O(1) since for searching for example 5, we will look at index 5, if the element is there, then we have it in our set; if not, then we do not have it in our set. Basically, we would straightly go to the index so the lookup is of constant time. However, this method has an issue; it is not memory efficient since the elements themselves, and hash values for each element must be stored in memory, and the hash table must be sized appropriately to avoid collisions[2].

[1] Some sources simplify O(λ),where λ is some input-dependent variable, to O(1) for the average case; yet still O(n) for the worst case time-complexity; for instance, looking up a key may take O(n) time if too many elements were hashed into the same key. λ in the hash set with the changing method for collison, can depend on the hash function and how the added values get different or similar locations depending on the hash function.
[2] A collision occurs when two or more elements are assigned the same hash value and are therefore stored in the same location in the hash table. If the hash table is too small, collisions will occur more frequently, which can lead to slower lookup times and increased memory usage. On the other hand, if the hash table is too large, it will use more memory than necessary. As a result, finding the right balance between avoiding collisions and minimizing memory usage can be challenging.

- Bitsets are another possible solution to this problem; very fast and memory efficient because they use a fixed-size array of bits, they can store and manipulate a large number of bits in a small amount of memory, and they provide fast access to individual bits in the sequence. However, if there is a large gap between the values, a large portion of the array will be unused. This can lead to wasted memory and decreased performance because the unused bits in the array still consume memory and must be accessed when manipulating the bits in the sequence.[3]

- Bloom filters are designed to solve this problem; they merge the idea of Hashsets and Bitsets into a data structure.

Let's begin with an example; suppose you have a bitset of a certain length; let's say 32, and you have k number of hash functions instead of only one; let's say 2 hash functions.
So each number in the set is hashed into 2 bitset entries, making them 1s instead of 0s. Then to see if a number p=5 is in the set, we use the same 2 hash functions, if all of the resulting indexes have a value of 1 in them, we can say probably p is in the original set S. But if any of the entries are 0, that means that p is for sure not in the set S.[4]

The reason is because of the way they use fixed-size bit arrays and hash functions to store and query elements in the set.

Refresher: A Bloom filter uses a fixed-size bit array and a set of hash functions to store and query elements in the set. When an element is added to the set, the Bloom filter calculates the hash values for the element using the hash functions and sets the corresponding bits in the bit array to 1. When querying for an element, the Bloom filter again calculates the hash values for the element and checks the corresponding bits in the bit array. If all of the bits are set to 1, the Bloom filter reports that the element is in the set.

False Positive - Possible
This is because the bit array is fixed in size, it is possible that two different elements will be assigned the same hash values and will therefore set the same bits in the bit array. When this occurs, querying for either of the elements will result in a positive match, even though only one of the elements is actually in the set. This is a false positive.

---

[3] This is because a bitset uses a fixed-size array of bits to represent the sequence of bits, and the size of the array is determined when the bitset is created. Additionally, if we decide to compress the bitset, not all operations are supported after encoding, without re-encofing.
[4]So if all entries are 1, we can not still know for sure that p is in the set S; but if any entry is 0, we can know for sure that the p is not in the set S; meaning false positives are possible but false negatives are not possible. So it will never report that an element is not in the set when it is.

False Negative - Impossible
This is because if an element is in the set, the Bloom filter will have set the corresponding bits in the bit array to 1 when the element was added. Therefore, querying for the element will always result in a positive match, regardless of whether there have been any false positives for other elements.

This is where the probabilistic nature of the Bloom filter manifests itself; in the false positive rate. A Bloom filter is space-efficient and allows for quick membership tests, but at the cost of potentially providing false positives. Different data structures are beneficial in different contexts; for example, Bloom filters offer space efficiency but may result in false positives, whereas hash tables can provide membership tests with a guarantee of no false positives but may require more space.

Bloom filters operations:

Adding an element to the filter:
To add an element to the filter, the element is hashed using each of the hash functions specified when the filter was created. The resulting hash values are used to set the corresponding indices in the filter. (from 0 to 1)

Adding an element to a Bloom filter is at the worst-case an O(1) operation because it only involves setting a fixed number of bits in the filter. The time complexity is constant regardless of the number of elements already in the filter because the number of hash functions and the size of the filter is fixed. So, setting the bits in the filter only requires a fixed number of steps, which means that the time complexity is constant.

In contrast, other data structures, such as hash tables, that can provide membership tests with a guarantee of no false positives typically have a worst-case time complexity of O(n) for adding elements, where n is the number of elements in the set. This means that their performance can degrade as the number of elements in the set increases.

looking up whether an element is a member of the set represented by the filter:
To test whether an element is a member of the set, the element is hashed using the same hash functions, and the resulting hash values are used to check the corresponding indices in the filter. If any of the indices are not 1, then the element is definitely not in the set. However, if all of the bits are set, meaning all the corresponding indices have 1s, it is still possible (although unlikely) that the element is not in the set, because it is possible for multiple elements to hash to the same values and set the same bits in the filter.

Looking up an element to a Bloom filter is at the worst-case an O(1) operation because it only involves checking a fixed number of bits in the filter. The time complexity is constant regardless of the number of elements already in the filter because the number of hash functions and the size of the filter is fixed. So, the time complexity is constant.

In contrast, other data structures, such as hash tables[5] typically have a worst-case time complexity of O(n) for looking up elements, where n is the number of elements in the set. This means that their performance can degrade as the number of elements in the set increases.

CBF and operations

Counting Bloom Filters have the same underlying principles as Bloom Filters, but their implementation is different. Instead of using zero or one to represent the presence or absence of an element, Counting Bloom Filters utilize counters in each slot. This difference allows for the deletion of elements from the data structure, which is not possible in Bloom Filters.

Why removing an element from a Bloom Filter is not possible?

Deleting an element from a Bloom filter would require unsetting the bits that were set to 1 when the element was added. However, this would create a problem: if an element is added to the set multiple times, and then deleted once, it would be impossible to determine whether the remaining bits that were set to 1 correspond to the element or to some other element that was added to the same positions. This would make it impossible to accurately test for membership which is the main operation supported by a Bloom filter. This is where Counting Bloom Filters come to the rescue.

Initialize

To begin the use of a Counting Bloom Filter, we must first create an array of m slots, each of which holds a numerical counter. At the start, all the slots will be set to zero as the data structure has not yet had any elements added. This operation has O(1) worst-case time complexity based on the number of elements added to the filter (input size). As it is upper-bounded and lower-bounded by the same function (of a constant), we can more accurately describe the time complexity as $\theta(1)$.

More in-depth:

The time complexity of initializing a counting Bloom filter depends on the size of the bloom filter and the number of hash functions used. Typically, a Bloom filter is initialized by creating an array of m bits, where m is the size of the filter, and setting all of the bits to 0. This operation has a time complexity of O(m), which is linear in the size of the filter.

[5]It is important to note that the operations in the Hash set usually perform much better since their average-case time-complexity is $O(\lambda)$ where λ is an input dependent variable which in this case depends on the hash function; and if the hash function is a good one for the inputs (a good hash function is introduced later in the paper) the average case is usually far better than the worst case tiem-complexity of O(n).

The number of hash functions used by a Bloom filter is typically a small constant, usually between 3 and 5. Each hash function is typically implemented as a randomly generated hash function, which means that generating the hash functions has a time complexity of O(k), where k is the number of hash functions.
Overall, the time complexity of initializing a counting Bloom filter is O(m + k), which is linear in the size of the filter and the number of hash functions. This means that the time required to initialize the filter is not significantly affected by the number of elements in the set, so it is a relatively fast operation. The number of elements in the filter does not directly affect the time complexity of initialization, so the time required to initialize the filter is not significantly affected by the number of elements in the set.

However, the number of elements in the set can affect the overall performance of the filter. If the number of elements in the set is much larger than the size of the filter, the filter will become full more quickly, which will increase the false positive rate. This means that membership tests and count operations will become less accurate.

Therefore, when choosing the size of a counting Bloom filter, it is important to consider the expected number of elements in the set (capacity) and choose a size that is large enough to provide an acceptable false positive rate while still being space-efficient. This will help to ensure that the filter performs well and provides accurate results.[6]

It is important to note that Counting Bloom Filters have a constant memory complexity of O(1) because of a fixed-size array (m). If, however, the size of the array was allowed to flexibly change over time, then the space efficiency of the filter would be lost.

Delete
The time complexity of the remove operation in a counting Bloom filter depends on the size of the filter and the number of hash functions used.[7] Thus, this operation has O(1) worst-case time complexity based on the number of elements added to the filter (input size). As it is upper-bounded and lower-bounded by the same function (of a constant), we can more accurately describe the time complexity as $\theta(1)$.

More in-depth:
A counting Bloom filter uses an array of slots, and several hash functions to map elements to positions in the array. When an element is added to the set, the hash functions are used to

[6]It is important to note that the initialization can be based on either of the two parameters from the four parameters: p = the probability of false-positive; m=: memory size, the number of slots in the filter; c = capacity, the expected number of items that we plan to store; k = number of hash functions. To have tuned parameters, there is a formula for calculating these parameters from each other that yields the best result (covered later in the paper).
[7] which are constant in my implementation.

determine which positions in the array to set to 1. So now, when an element is supposed to be removed, the count at each of the positions in the filter that the element maps to is decremented by 1.

In a similar process, the hash functions are used to map the element to positions in the array. Then, for each of the positions that the element maps to, the count at that position is decremented. This operation has a time complexity of O(k), where k is the number of hash functions.

Overall, the time complexity of the remove operation in a Counting Bloom filter is O(k), which is linear in the number of hash functions. This means that the time required to remove an element is not significantly affected by the size of the filter or the number of elements that we are expecting to have added.

Insert

In order to add an element to a Counting Bloom filter, the element is mapped to certain positions in the array using hash functions. For each of these positions, the count of that element is increased by 1. This process has a time complexity of O(k), where k is the number of hash functions. However, since the number of hash functions is constant and the number of items that need to be added to the set does not influence it, the worst-case time complexity is O(1). Additionally, as it is upper-bounded and lower-bounded by the same function (of a constant), we can more accurately describe the time complexity as $\theta(1)$.

Search/Lookup

To look up an element in a data structure, we start by utilizing the hashing functions to determine the position of the element we are looking for. Then, we check each slot at the returned position to see if the counter value is greater than zero. If it is in all the slots associated with that element, then the element we are looking for exists in the data structure. On the other hand, if the counter value is zero for any of the slots associated with that element, then the element does not exist. It is important to note that sometimes the lookup function may give a false-positive result, meaning that the lookup function says the element is present when it is not. This operation has a time complexity of O(k), where k is the number of hash functions. However, since the number of hash functions is constant and the number of items that need to be added to the set does not influence it, the worst-case time complexity is O(1). Regardless of the length of the phrase, the time required to search for it remains constant; as it is upper-bounded and lower-bounded by the same function (of a constant), we can more accurately describe the time complexity as $\theta(1)$.

False positive:

One of the approximations for the error rate is the equation[8] below:

[8] (Manning Publications. (2020, September 22)

$$False\ Positive\ rate\ \simeq (1 - e^{-kc/m})^{k}$$

Here, it is important to note that m (memory size) has the most significant effect on the False positive rate. Since the number of hash functions (k), if increased, reduces the FPR by reducing the risk of collision, however, if m is not large enough, this effect diminishes. So for the increase of k to show a reduction in FPR, we should have a large enough m. An extreme example of it would be the bit array that is full and reports cbf.search() as True for any element; this array has to be rehashed and have a larger memory allocated for it and changing the number of the hash functions cannot have any effect.

As m increases, it increases the encoding space which reduces the probability of collision, thus also reducing the probability of false positives. And by increasing the number of hash functions (k) we increase the number of keys and thus collision is reduced and also FPR because the chance of two elements overlapping with the use of multiple different hash functions is very low.

It's important to note that as the hash functions increase, calculating them would require computational power thus adding to the time complexity of our algorithm; however, there are tricks like double hashing to reduce these costs.

Overall, the power of CBF is the control over the tradeoff between accuracy and performance, giving us the ability to make a decision about how much accuracy we are willing to sacrifice for improved performance.

2. Give a few examples of practical, real-life computational applications that can benefit from using CBFs and carefully justify why.

Some areas that probabilistic data structures apply to include:

- Determine if the element is present in the data or not
- Count the number of unique elements in the data
- Count the frequency of a particular element in the data

Advantages of using CBFs include:

Security of the elements of the set (due to not storing the elements themselves,)
Fast look-up performance,
Memory efficiency due to not storing the elements themselves,
Efficient approximate counting mechanism[9]

[9]particularly for determining if a particular data element has been referenced θ or more times, where θ is a predefined threshold.

Biomedical research applications:

Biomedical research applications often need to process large datasets of genetic sequences to identify mutations and disease-associated genes. Counting Bloom filters can be used to efficiently store and query the sequences, and to keep track of the number of occurrences of each mutation and gene, which can be useful for identifying common patterns and associations.

One of the key benefits of using counting Bloom filters for this application is their space efficiency. Counting Bloom filters use a compact bit array to represent the set, which means that they can store a large number of elements in a relatively small amount of memory. This is important for genetic sequence data, as the datasets can be very large and require a significant amount of storage.

Another advantage of counting Bloom filters is their fast lookup performance (explained earlier.) Counting Bloom filters use multiple hash functions to map elements to positions in the array, which allows them to quickly determine whether an element is a member of the set. This is useful for identifying mutations and disease-associated genes, as the datasets can be very large and require efficient searching to find the relevant sequences.

In addition, the count operation is a key feature of counting Bloom filters that sets them apart from regular Bloom filters.[10] Counting Bloom filters support the count operation, which allows them to keep track of the number of times an element has been added to the set. This is useful for identifying common patterns and associations in the genetic sequence data, as it allows researchers to easily determine the frequency of each mutation and gene.[11]

Plagiarism

Counting Bloom filters can be used in plagiarism detection applications to efficiently store and query large datasets of text documents, and to keep track of the number of occurrences of repeated phrases in the documents.

Plagiarism detection systems often need to process large collections of documents, such as academic papers or articles, to identify instances of plagiarism. This can be a challenging task, as the datasets can be very large and may contain a large number of unique phrases. Counting Bloom filters can be useful in this context because they can store a large number of elements in a

[10] Unlike regular Bloom filters, which only support the add and lookup operations, counting Bloom filters also support the count operation, which allows them to keep track of the number of times an element has been added to the set.

[11] When applied to DNA sequence identification, approximate count thresholding can be used as a fast way to locate or match strings in a DNA sequence that occur more than θ times (Kim et al., 2019).

relatively small amount of memory, and they support fast lookup performance and the count operation.

To use counting Bloom filters for detecting repeated phrases in plagiarism detection, the system first pre-processes the text documents to extract the individual phrases and remove any common words or punctuation. Then, the system adds each phrase to a counting Bloom filter, incrementing the count for each occurrence of the phrase. This allows the system to keep track of the number of occurrences of each phrase in the documents.

When a new document is submitted for analysis, the system pre-processes the document in the same way and uses the counting Bloom filter to look up each phrase. If the count for a particular phrase is greater than a predetermined threshold, the system can flag the phrase as potentially plagiarized, indicating that it appears frequently in the existing documents. The system can then use additional methods, such as comparing the sentence structure and word order, to determine whether the document is actually plagiarized.

Overall, counting Bloom filters can be useful for efficiently storing and querying large datasets of text documents, and for identifying potential instances of plagiarism by keeping track of the number of occurrences of repeated phrases in the documents.

3. Implement CBFs in Python using the code template below. Ensure you carefully organize your Python code with appropriate docstrings and meaningful comments and provide a thorough justification for your choice of hash functions.

- propose your own hash functions and explain your choices thoroughly. Specifically, there are several properties that a good hash function should satisfy, and you need to demonstrate that your CBF implementation aligns with those properties.[12]

Hash function:

A good hash function:

1. Minimizes the collision (maximizing the property below for as many inputs as possible)

$$\text{If } key1 \neq key\,2 \rightarrow hash(key1) \neq hash(key2)$$

Collision being: If $key1 \neq key\,2 \rightarrow hash(key1) = hash(key2)$

It should be collision-resistant. This means that it should be difficult for two different inputs to produce the same output. This is important for the CBF to be able to accurately represent the elements it contains, as collisions will cause the CBF to report false positives.

[12]The code and hash functions are included in the Appendix of code at the end of the report.

2. Unifrom

$$\text{If } key1 = key\,2 \rightarrow hash(key1) = hash(key2)$$

It should be deterministic, meaning that given the same input, it should always return the same output. This is important for the CBF to work properly, as it relies on being able to consistently and reliably map elements to slots in the filter.

3. Efficiently computable. This is important for performance reasons, as the CBF will be called many times and we don't want it to slow down our program.

4. Evenly distributed. This means that the output of the hash function should be spread out evenly over the range of possible outputs. This is important for the CBF to use its space efficiently, as an evenly distributed hash function will map elements to slots in the filter in a more uniform manner, allowing the CBF to use its space more efficiently.

While the first two properties are about the hash code, the fourth is covered by compression maps. Conversion maps are used to convert integers from $(-\infty, +\infty)$ into a range of [0, N-1]; meaning turning the hash values and converting them to corresponding hash values that are restrained within a certain fixed length. These can include methods including:

1. Division

   This method works by taking a numerical key, such as an integer, and dividing it by a predetermined number, known as the divisor. The remainder of the division is then used as the hash value. For instance, if the compression map is as follows, for N=101

$$CMP(y) = y \bmod N$$

   Then for the hashcode values of 101, 202, 303, the corresponding compression map results are going to be 0,0,0; which suggest that this method is prone to causing collision due to clustering. Thus we need a better compression map method for our hash function.

2. Multiply Add Divide (MAD)

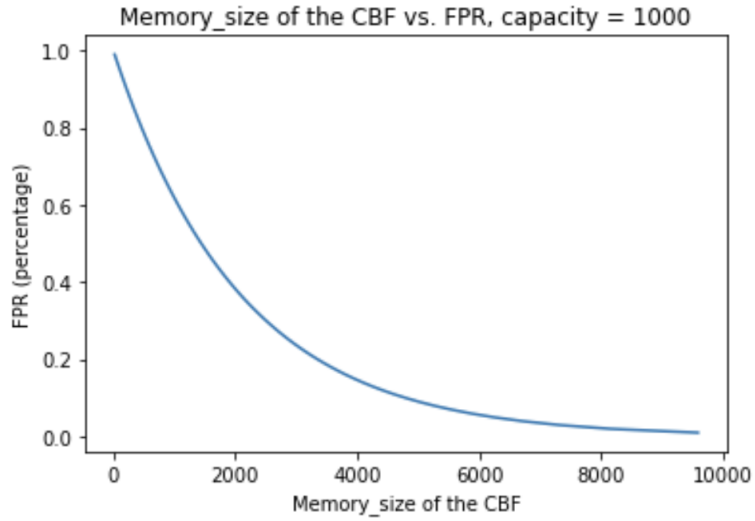
   This method combines the operations of multiplication, addition, and division as below; where N is a prime integer, p is a prime integer smaller than N, a and b are nonzero constants.

$$CMP(y) = ((ay + b)\ \%\ p)\ \%\ N$$

   The reason N and p are prime is that prime numbers do not have a common factor thus making the modulo operation effective in reducing collisions as there will be much fewer numbers with the same final hashed values compared to using a nonprime number. For instance, for the hashcode values 101, 202, and 303, the corresponding compression map results are going to be 1,4, and 2.

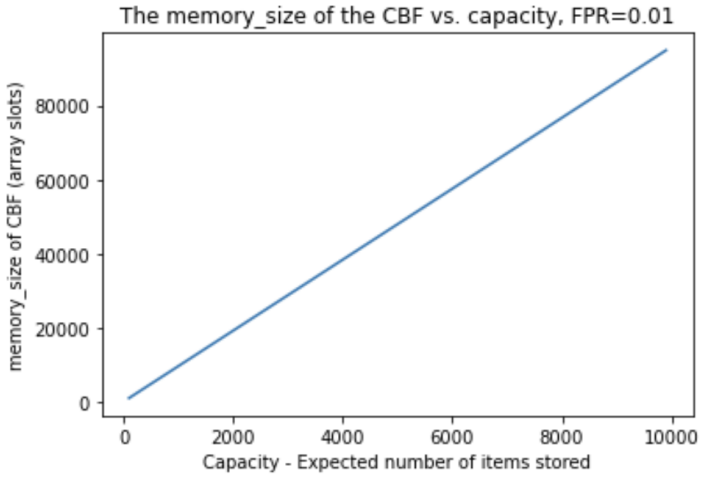
4. Verify the effectiveness of your CBF implementation by testing how well the implementation matches or diverges from the theoretical assumptions using data collected in all_text from Shakespeare's works (which can be retrieved from here). The following tasks will help you make this computational analysis, supported by plots you shall produce.

a. How does the memory size scale with FPR?



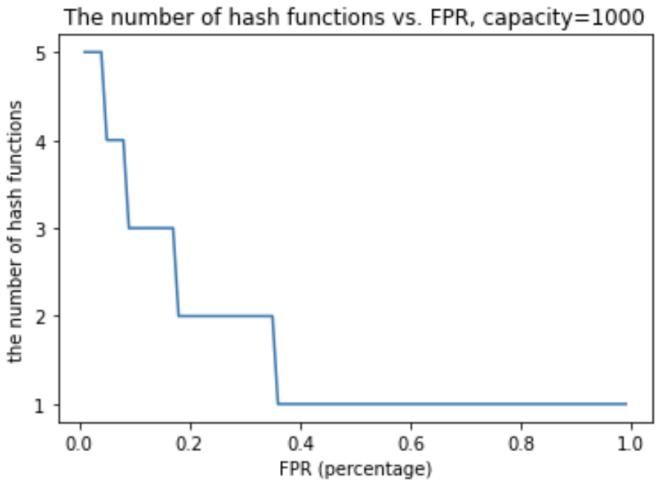
As FPR increases, the amount of memory used should decrease. This is due to the additional memory used to provide more freedom to avoid False Positive cases. As seen in the accompanying graph, the memory size decreased logarithmically as the FPR increased.

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



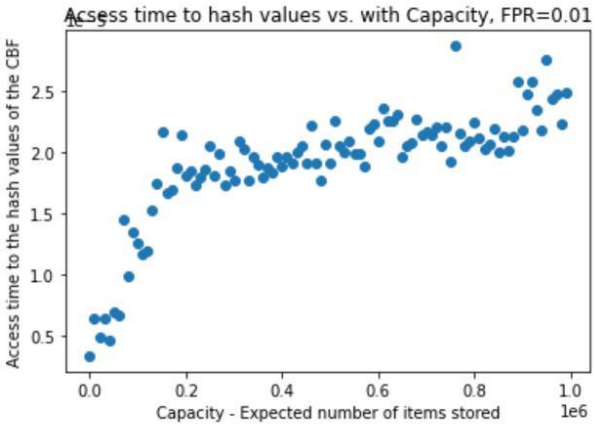
when attempting to minimize the False Positive Rate (FPR), the size of the Content-based Phrase Filtering (CPF) memory must increase proportionally with the number of items stored within it. This hypothesis is supported by the empirical experiment conducted here, which demonstrates a linear increase in memory size with text length, as seen in the figure above.

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



As the false positive rate (FPR) increases, the number of hash functions utilized decreases in a logarithmic fashion. This is due to a Counting Bloom Filter (CBF)'s design, which utilizes more hash functions to lower the FPR. This was observed in experiments where the number of inserted elements remained constant while the FPR increased.

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



The implementation of the Counting Bloom Filter (CBF) in this study dictates that the number of hash functions used is dependent on the ratio of n/m (number of elements to be inserted/size of the CBF array), which is in turn largely determined by the False Positive Rate (FPR). However I have implemented my CBF to have a limited number of hash functions between 1 to 5; thus, the

time needed to access hashed values of an element should scale linearly with the number of items stored. As more items are stored, the time to access each item will increase proportionally, but in this implementation, the number of hash functions does not go beyond 5, so we may need rehashing but the hashed values in my implementation scale linearly until 5 and then stay at the maximum possible number of hash functions (5).

5. Establish a new computational application for this data structure. Apply your CBF implementation to determine the extent of plagiarism between these three pieces of text (version_1, version_2, and version_3, defined below). You will need to:

a. Explicitly state how you would define plagiarism between two texts.

Plagiarism is stealing the work of others; thus if a document has taken work from another document, there have to be similar phrases between the two documents. The number of phrases that are shared between doc1 and doc2 divided by the number of phrases in doc1, is the percentage of doc1 that has been similar to others' work. If this percentage is above a threshold that I have experimentally defined to be:

Less than 10%: Green
More than 10% but less than 20%: Yellow
More than 20% but less than 20%: Orange
More than 30%: Red

These colors represent, probably not plagiarised, maybe plagiarised, likely plagiarised, highly likely plagiarised. These percentages can differ based on how long we choose a phrase to be, how is the form of the document (for example an assignment may include up to 50% similarity by just including the prompts of the assignment) thus we either have to make sure we clean our data promptly or take into account that these thresholds are just a matter of a signature rather than a determining factor. In order to actually know, one has to actually see what phrases are shared but these serve as a flag to gather one's attention; especially as the number of documents for comparison gets larger and as the length of each document increases.

We use the CBF to know which documents are above a certain threshold and check only those that are most probable to actually include meaningful similarities.

b. Argue for the strengths of utilizing CBFs to detect plagiarism while making explicit references to any limitations/failure modes. Specifically, why don't we compare word by word and use hashing instead?

Comparing word by word may flag plagiarism in many documents; it is an unreliable and ineffective method for detecting plagiarism. This is because it does not take into account the context of the words or phrases being compared, and simply comparing words does not take into account the idea or concept behind the writing. This can lead to false positives as well as false negatives, making it difficult to accurately identify plagiarism.

Another strong method that can be used, is rolling hashing for the CBF; it would be basically the Rabin Karp Algorithm but with CBFs.[13]

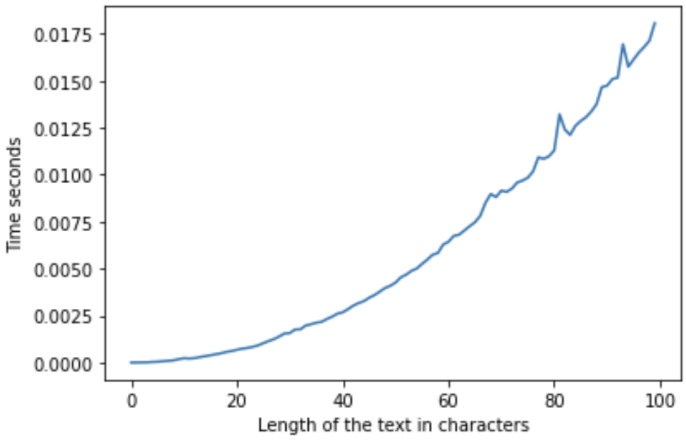
As already discussed, Counting Bloom Filter (CBF) is an effective tool for identifying potential plagiarism due to its quick determination of whether a phrase is present in a set of known phrases. The CBF technique requires just O(length of this doc) to convert a doc to a CBF and O(length of the other doc) to convert another document's phrases to CBF, resulting in a total running time of O(length of doc1+length of doc2) worst-case, while the sliding window algorithm requires O(n^2) in the worst-case. CBF can be used to determine if the similarity between two texts exceeds a threshold, in which case the other non-probabilistic techniques should be implemented to compare the text phrase by phrase. Thus, CBF can drastically reduce the number of computations necessary in cases where the two texts do not match.

In addition, CBF algorithms can sometimes produce false positives, meaning that they may flag text as being plagiarized even when it is not. This can be particularly problematic when the text being compared is similar in content but not actually plagiarized.

c. Identify what other algorithmic strategies could be implemented to detect plagiarism, and explain why. Be sure to make your comparison as thorough as possible, using any experimental analysis setup you deem appropriate.

Another method is the naive algorithm for pattern searching:
Naive algorithm for pattern searching is a search algorithm used to locate a pattern of characters in a larger string. It works by comparing each character of the pattern to each character of the string, one by one until it finds a match. If a match is found, it starts searching for the next character of the pattern in the same location in the string. If a mismatch is found, it starts searching for the first character of the pattern in the next location in the string. This process is repeated until all characters of the pattern have been found or until the end of the string is reached. The algorithm is considered to be naive because it is not particularly efficient.

[13]which cross my mind too late to be able to implment it.



Time complexity of the naive algorithm

As we can see above this algorithm has the worst-case time complexity of O(n^2) as we can see when the length of the text doubles the time it takes to compare using this algorithm gets multiplied by 4.

A better approach would be the Rabin-Karp algorithm which uses hashing to search for patterns in large bodies of text. It works by first creating a hash of the pattern to be searched, then creating a hash of each substring of the text, and then comparing the hashes. If the hashes match, then the algorithm checks if the two patterns are actually the same. If they are, then plagiarism has been detected. The algorithm is efficient and can detect plagiarized text even if there are spelling variations or other changes in the text.
This algorithm has an average and best-case running time of O(n+m) where n is the number of characters in the text and m is the number of characters in the pattern.
The worst-case for this algorithm has a time complexity of O($m \times n$).

Longest Common Subsequence (LCS) is an algorithm for finding similarities between two strings. It has a time complexity of O(mn), where m and n are the lengths of the two strings. There are advantages to using this algorithm, such as its relative simplicity and its ability to measure the degree of relatedness between two strings. However, it is not able to effectively handle large strings due to its time complexity of O(mn) if we use dynamic programming, and it is also not able to capture the context of the strings, thus making it difficult to accurately measure the degree of relatedness.

The fastest of all, CBFs, in the context of plagiarism detection, a counting Bloom filter can be used to quickly and efficiently compare two documents to determine if they are similar. The counting Bloom filter works by converting each document into a set of hash values and then comparing the sets for common elements. If any common elements are found, then the documents are likely to contain similar content and may be flagged for further investigation. The

advantage of using counting Bloom filters for plagiarism detection is that they are extremely fast and efficient, and can detect even small similarities between documents.

6. List all the LOs and HCs you have exercised while working on this final assignment and a thorough justification of their application in about 50 words per LO and HC:

LOs:
#PythonProgramming:
I made sure to test which made an important difference in my hash functions and the method I used; I also tested the CBF and its method; additionally I tested the plots by using theoretical and practical methods together and compared the results. I made functional codes that I reused.
Word count: 50

#AlgoStratDataStruct:
I included many contexts and easy-to-understand and jargon-less explanations of the data structure to the best of my ability by bringing a history of the other data structures that solved a similar problem and building up on the bloom filters and counting bloom filters.
Word count: 47

#ComputationalCritique:
I kept my analysis concise by not including all the plots in the main report, instead opting to include only the necessary results and explain the key important points, how I could only include those points or graphs, etc. I explained which complexity was better and in what context and why.
Word Count: 51

#ComplexityAnalysis:
I made sure to include an interpretation of the growth rate, for example, by looking at the double-sized input. I also made certain to include Big Omega and Big Theta through lower and upper bounds, but I took care to clarify them further.
Word count: 44

#Professionalism:
I attempted to shorten my labels, titles, and legend labels; and formatted my references more professionally based on the feedback. I revised the report and structuralized it; I cited sources using APA style; I included Addendum, additional footnote explanations, and equations to better represent the hash functions.
Word count: 46

#CodeReadability:

I maintained the appropriate variable names and comments which explained the strategy behind my code, in order to enhance #codereadability, I made sure to provide doc strings for all functions and classes and to space out the code more.
HCs: 39

#optimization
explored how to minimize the false positive rate (objective function), via decision variables: the number of hash function memory size and capacity while on another level optimizing them if possible; tested how these variables scale with each other and discussed which has more effect on the formula and why.
Word count: 49

#Dataviz
Titled and labeled plots for the experiments; I created plots based on theory, experiment, and experiment of the prompt of using Shakespeare to make sure the plots are easily understandable and correct, as far as I could tell.
Word count: 38

#Algorithm
I coded multiple approaches, processed the input to serve the algorithm, explained my approach in the report, and compared the algorithms in the report, also explained the use of the algorithm and how we can combine them to be even more efficient and accurate (CBFs first then Rabin-Karp)
Word count: 49

Addendum

A bitset:

is a data structure that is used to store and manipulate a sequence of bits. It is called a bitset because it uses a fixed-size array of bits to represent the sequence of bits. Each bit in the array can be accessed individually, which allows for fast manipulation of the bits in the sequence.

Bitsets are very fast and memory-efficient for several reasons. First, because a bitset uses a fixed-size array of bits, it does not need to dynamically allocate memory as other data structures do. This means that it can access and manipulate bits in the array without incurring the overhead associated with dynamic memory allocation.

Second, a bitset can store and manipulate a large number of bits in a small amount of memory. This is because each bit in a bitset is represented using a single bit, which is the smallest unit of memory that can be accessed. As a result, a bitset can store and manipulate a large sequence of bits using only as much memory as is necessary to store the bits themselves.

Third, a bitset provides fast access to individual bits in the sequence. This is because each bit in the array is stored at a fixed location, which can be calculated using the index of the bit. As a result, a bitset can access individual bits in the sequence in constant time, without the need for searching or other expensive operations.

Why use probabilistic data structures?

With the increasing amount of data generated each day, companies are facing the challenge of understanding it quickly in order to gain valuable insights and take action. Traditional technologies such as data structures and algorithms are no longer effective in tackling Big Data. To solve this issue, Probabilistic Data Structures are the best solution. If the data set is too large for the available memory, using a deterministic data structure is not possible. Probabilistic Data Structures offer approximate answers, with a reliable method to measure possible errors. Despite these drawbacks, they have the advantage of low memory usage, consistent query time, and scalability with data size, which more than makes up for the potential inaccuracies.

Sources

Kim, K., Jeong, Y., Lee, Y., & Lee, S. (2019a). Analysis of Counting Bloom Filters Used for Count Thresholding. *Electronics*, *8*(7), 779. https://doi.org/10.3390/electronics8070779

Kim, K., Jeong, Y., Lee, Y., & Lee, S. (2019b). Analysis of Counting Bloom Filters Used for Count Thresholding. *Electronics*, *8*(7), 779. https://doi.org/10.3390/electronics8070779

Manning Publications. (2020, September 22). *All About Bloom Filters*. Manning. https://freecontent.manning.com/all-about-bloom-filters/

This code is defining a class called CountingBloomFilters. It the class is used to create a counting bloom filter data structure, which is a probabilistic data structure that is used to test whether an element is a member of a set. The class has several methods, including __init__, hash_cbf, insert, delete, and search.

The __init__ method is used to initialize the class and assign the optimal values for the attributes based on the inputs. It takes two parameters: num_item, which is the expected size of the input, and fpr, which is the desired false positive probability. The method then uses these inputs to calculate the optimal size of the array and the optimal number of hashing functions to be used. It also initializes the array attribute with the optimal size and sets all of the elements to 0.

The hash_cbf() method takes an element and returns a list of keys that can be used to insert or search for the element in the CBF. The insert() method takes an element and uses the keys generated by hash_cbf() to insert the element into the CBF. The delete() method takes an element and uses the keys generated by hash_cbf() to delete the element from the CBF (if it exists). The search() method takes an element and uses the keys generated by hash_cbf() to search for the element in the CBF. It returns True if the element is in the CBF, and False if it is not.

Hash function
a good hash function should distribute the elements evenly across the range of possible hash values. This can help to avoid collisions, where multiple elements are mapped to the same hash value.

In the code, the hash_cbf() method uses the built-in hash() function to generate the keys for an element. The hash() function is a general-purpose hash function that is included in Python, but it may not be well-suited for all applications; thus we have created our own hashing function on top of it.

In addition to using a good hash function, it is important to use multiple hash functions in a CBF. This helps to reduce the chance of false positives, where the CBF incorrectly indicates that an element is in the set when it is not. The hash_cbf() method in the code uses 5 different hash functions, however depending on the other parameters not all of them may be used.

In [106]:

```
#importing the libraries
import math
#creating a class for the counting bloom filters
class CountingBloomFilters:
    """
    This class is used to create a counting bloom filter data structure
    The class has the following methods:
    1- __init__ : This method is used to initialize the class
    2- hash_cbf : This method is used to create the hash functions
    3- insert : This method is used to insert elements into the data structure
    4- delete : This method is used to delete elements from the data structure
    5- search : This method is used to search for an element in the data structure
    """

    def __init__(self, num_item, fpr):
        """
        This method is used to initialize the class and
        assignes the optimal values for the attributes based on the inputs

        Parameters:
        1- num_item : The expected size of the input (c), int
        2- fpr : The desired false positive probability (p), float

        The CBF has the following attributes:
        num_item : The expected size of the input (c), int
        fpr : The desired false positive probability (p), float
        memory_size : The optimal size of the array (m), int
        array : The array that will hold the elements, list
        num_hashfn : The optimal number of hashing functions (k), int
        """
        self.num_item = num_item
        self.fpr = fpr

        #getting the optimal memory_size (m) based on the fpr and the number of items
        self.memory_size = -1*round ((self.num_item * math.log (self.fpr)) / (math.log (2))**2)
        self.array = [0] * self.memory_size
        #create the array with the optimal size with 0s to be used as counters

        #getting the optimal number of hashing functions
        #(k) based on the fpr and the number of items
        self.num_hashfn = round ((self.memory_size / self.num_item ) * math.log (2))

        #check if the number of hashing functions is more than 5
        #or less than 1, in order to set the limit
        if (self.num_hashfn < 1):
            self.num_hashfn = 1
        elif (self.num_hashfn > 5):
            self.num_hashfn = 5


    def hash_cbf(self, element):
        """
        This method is used to get hash values for the element

        Parameters:
        element : The element that will be hashed, int

        Returns:
        keys : The list of keys for the element, list
        """
        keys = [] #Create an empty list to hold the keys
        #Find the hash values for the element
        hashed_value1 = (hash (element) * 31 + 7) % self.memory_size
        hashed_value2 = (hash (element) * 11 + 5) % self.memory_size
        hashed_value3 = (hash (element) * 19 + 11) % self.memory_size
        hashed_value4 = (hash (element) * 29 + 7) % self.memory_size
        hashed_value5 = (hash (element) * 17 + 3) % self.memory_size

        #Convert the hash to integer and add it to the list
        keys.append(int (hashed_value1))
        keys.append(int (hashed_value2))
        keys.append(int (hashed_value3))
        keys.append(int (hashed_value4))
        keys.append(int (hashed_value5))
        return keys


    def insert(self, element):
        """
        This method is used to insert elements into the data structure

        Parameters:
        element : The element that will be inserted, int

        Returns:
        None
        """
        key=0 #define the key variable
        keys = self.hash_cbf (element) #get the keys for the element

        for index in range (self.num_hashfn): #for every key in the list up to the limit
            key = int (keys [index] % self.memory_size) #get the key moduls from the list size
            #in the array of cbf at the key index, increase the counter value by one
            self.array [key] += 1


    def delete(self, element):
```

```
        """
        This method is used to delete elements from the data structure

        Parameters:
        element : The element that will be deleted, int

        Returns:
        None
        """
        key = 0 #define the key variable
        keys = self.hash_cbf(element) #get the keys for the element

        if (self.search(element)):#Check if the element exists in the data structure
            for index in range(self.num_hashfn):#for every key in the list up to the limit
                key = int(keys[index] % self.memory_size)#get the key moduls from the list size
                #in the array of cbf at the key index, decrease the counter value by one
                self.array[key] = self.array[key] - 1

        else: #If the element doesn't exist
            print("Element is not in the CBF!") #Error message

    def search(self, element):
        """
        This method is used to search for an element in the data structure

        Parameters:
        element : The element that will be searched for, int

        Returns:
        exists : The result of the search, bool
        """
        key = 0 #define the key variable
        exists = True #define the exists variable
        keys = self.hash_cbf (element) #get the keys for the element

        for index in range (self.num_hashfn): #for every key in the list up to the limit
            key = int (keys [index] % self.memory_size) #get the key moduls from the list size

            if self.array[key] == 0: #If the counter value is 0
                exists = False #The element doesn't exist

        return exists
```

In [108]:

```
#testing the hash functions
#if two elements are equal they should have the same hash values
assert((hash ("1") * 31 + 7)% 101) == ((hash (str(1)) * 31 + 7)% 101)
assert((hash (45) * 11 + 5)% 101) == ((hash (int("45")) *  11 + 5)% 101)
assert((hash (266) * 19 + 11)% 101) == ((hash (int(266)) * 19 + 11)% 101)
assert((hash (266) * 29 + 7)% 101) == ((hash (float(266)) * 29 + 7)% 101)
assert((hash (2.5) * 17 + 3)% 101) == ((hash (5/2) *  17 + 3)% 101)

#testing the cbf by reating bloom filters of strigns
#and checking if the elements are in the bloom filter or not
cbf = CountingBloomFilters(10, 0.01)

#inserting 10 elements
for i in range(10):
    cbf.insert(str(i))

#checking if the elements are in the bloom filter
for i in range(10):
    assert(cbf.search(str(i)) == True)

#checking if the elements are not in the bloom filter
for i in range(11, 15):
    assert(cbf.search(str(i)) == False)
#assert(cbf.search(str(16)) == False) gives True due to #False positive


#testing the delete function

#deleting an element that is in the bloom filter
cbf.delete(str(0))
#checking if the element is in the bloom filter
assert(cbf.search(str(0)) == False)
```

In [24]:

```
url_version_1 = 'https://bit.ly/39MurYb' # URL for the first version of the text file
url_version_2 = 'https://bit.ly/3we1QCp' # URL for the second version of the text file
url_version_3 = 'https://bit.ly/3vUecRn' # URL for the third version of the text file
from requests import get # import the get function from the requests library
def get_txt_into_list_of_words(url):
    '''
    a function that takes a URL of the text as input
    and 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 = [';', ',', '.', '?', '!', '_', '[', ']', '(', ')', '*'] # list of characters to remove
    data = get(url).text # get the text from the URL
    data = ''.join(c for c in data if c not in bad_chars) # remove the bad characters defined above
```

```
    data_without_newlines = ''.join(c if (c not in ['\n', '\r', '\t']) else " " for c in data) # replace newlines with spaces
    data_just_words = [word for word in data_without_newlines.split(" ") if word != ""] # split the text into words
    data_just_words_lower_case = [word.lower() for word in data_just_words] # convert all words to lower case
    return data_just_words_lower_case # return the list of words
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)
```

In [193]:

```
#get the phrases of the documents
def get_phrases(list_of_words): #defined as every 3 words
    """
    This function takes a list of words as input and returns a list of phrases.
    Each phrase is a string of 3 words separated by a space.
    Input
    ----------
    list_of_words : list
    A list of words.
    Returns
    -------
    phrases : list
    A list of phrases.
    """
    phrases = [] #define the phrases list
    for i in range(len(list_of_words)-2): #loop through the list of words up to the last 3 words
        phrases.append(list_of_words[i] + " " + list_of_words[i+1] + " " + list_of_words[i+2]) #append the phrase to the list
    return phrases

phrases_version_1 = get_phrases(version_1)
phrases_version_2 = get_phrases(version_2)
phrases_version_3 = get_phrases(version_3)

#create a counting bloom filter from the phrases of each document
def create_cbf(list_of_phrases, fprate=0.01):
    """
    This function takes a list of phrases as input and returns a counting bloom filter.
    Input
    ----------
    list_of_phrases : list
    A list of phrases.
    Returns
    -------
    cbf : CountingBloomFilters
    A counting bloom filter.
    """
    cbf = CountingBloomFilters(len(list_of_phrases), fprate)
    for phrase in list_of_phrases:
        cbf.insert(phrase)
    return cbf

#create the cbf for each document
cbf_version_1 = create_cbf(phrases_version_1)
cbf_version_2 = create_cbf(phrases_version_2)
cbf_version_3 = create_cbf(phrases_version_3)


def count_common_phrases(phrases_list, cbf):
    """
    This function takes a list of phrases and a counting bloom filter as input and returns the number of common phrases.
    Input
    ----------
    phrases_list : list
    A list of phrases.
    cbf : CountingBloomFilters object
    A counting bloom filter.
    Returns
    -------
    counter : int
    The number of common phrases.
    """
    counter=0 #define the counter
    for phrase in phrases_list: #loop through the phrases
        if cbf.search(phrase): #if the phrase is in the cbf
            counter+=1 #increment the counter
    return counter

#for the three documents, count the number of common phrases with each other
print("count of common phrases of version one with itself, version two, and version three respectively:")
print(count_common_phrases(phrases_version_1, cbf_version_1))
print(count_common_phrases(phrases_version_1, cbf_version_2))
print(count_common_phrases(phrases_version_1, cbf_version_3))
print("count of common phrases of version two with itself, version two, and version three respectively:")

print(count_common_phrases(phrases_version_2, cbf_version_2))
print(count_common_phrases(phrases_version_2, cbf_version_1))
print(count_common_phrases(phrases_version_2, cbf_version_3))
print("count of common phrases of version three with itself, version two, and version three respectively:")

print(count_common_phrases(phrases_version_3, cbf_version_3))
print(count_common_phrases(phrases_version_3, cbf_version_2))
print(count_common_phrases(phrases_version_3, cbf_version_1))
```

```
count of common phrases of version one with itself, version two, and version three respectively:
8459
883
907
count of common phrases of version two with itself, version two, and version three respectively:
8459
842
930
count of common phrases of version three with itself, version two, and version three respectively:
```

8552
912
865

In [194]:

```
doc1_common1=(count_common_phrases(phrases_version_1, cbf_version_1))
doc1_common2=(count_common_phrases(phrases_version_1, cbf_version_2))
doc1_common3=(count_common_phrases(phrases_version_1, cbf_version_3))
print("potential plagiarism of version 1 of version 1", round(doc1_common1/doc1_common1)*100, "percentage")
print("potential plagiarism of version 1 of version 2", (doc1_common2/doc1_common1)*100, "percentage")
print("potential plagiarism of version 1 of version 3", (doc1_common3/doc1_common1)*100, "percentage")

doc2_common2=(count_common_phrases(phrases_version_2, cbf_version_2))
doc2_common1=(count_common_phrases(phrases_version_2, cbf_version_1))
doc2_common3=(count_common_phrases(phrases_version_2, cbf_version_3))
print("potential plagiarism of version 2 of version 2", (doc2_common2/doc2_common2)*100, "percentage")
print("potential plagiarism of version 2 of version 1", (doc2_common1/doc2_common2)*100, "percentage")
print("potential plagiarism of version 2 of version 3", (doc2_common3/doc2_common2)*100, "percentage")


doc3_common3=(count_common_phrases(phrases_version_3, cbf_version_3))
doc3_common2=(count_common_phrases(phrases_version_3, cbf_version_2))
doc3_common1=(count_common_phrases(phrases_version_3, cbf_version_1))
print("potential plagiarism of version 3 of version 3", (doc3_common3/doc3_common3)*100, "percentage")
print("potential plagiarism of version 3 of version 2", (doc3_common2/doc3_common3)*100, "percentage")
print("potential plagiarism of version 3 of version 1", (doc3_common1/doc3_common3)*100, "percentage")
```

```
potential plagiarism of version 1 of version 1 100 percentage
potential plagiarism of version 1 of version 2 10.438586121290932 percentage
potential plagiarism of version 1 of version 3 10.722307601371321 percentage
potential plagiarism of version 2 of version 2 100.0 percentage
potential plagiarism of version 2 of version 1 9.953895259486938 percentage
potential plagiarism of version 2 of version 3 10.994207353115025 percentage
potential plagiarism of version 3 of version 3 100.0 percentage
potential plagiarism of version 3 of version 2 10.664172123479888 percentage
potential plagiarism of version 3 of version 1 10.114593077642658 percentage
```

In [195]:

```
def plagiarism_detector(list_of_words_doc, list_of_words_others):
    """
    This function takes two lists of words as input and returns the percentage of plagiarism.
    Input
    ----------
    list_of_words1 : list
    A list of words.
    list_of_words2 : list
    A list of words.
    Returns
    -------
    percentage : float
    The percentage of plagiarism.
    """
    phrases1 = get_phrases(list_of_words_doc) #get the phrases of the first document
    phrases2 = get_phrases(list_of_words_others) #get the phrases of the second document
    cbf1 = create_cbf(phrases1) #create the cbf of the first document
    cbf2 = create_cbf(phrases2) #create the cbf of the second document
    common_phrases_base = count_common_phrases(phrases1, cbf1) #count the common phrases
    common_phrases = count_common_phrases(phrases1, cbf2) #count the common phrases
    percentage = (common_phrases/common_phrases_base)*100 #calculate the percentage
    return percentage #this is the percentage of doc that is potentially plagiarized form other
```

In [114]:

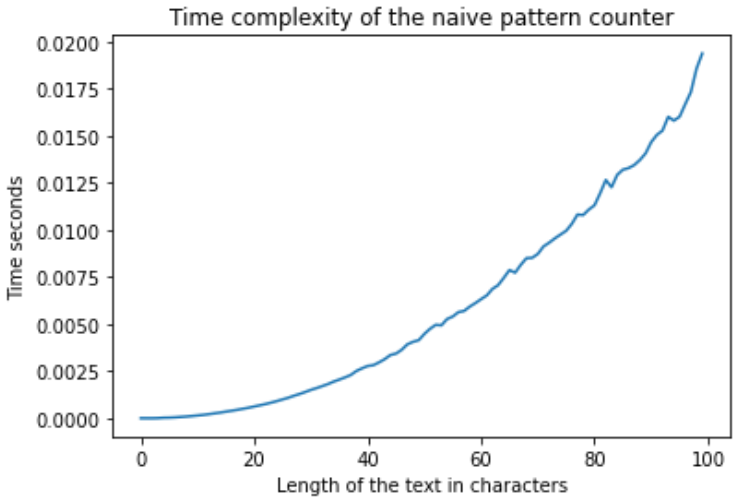
```
#naive pattern searching algorithm
def naive_pattern_counter(pattern, text):
 """
 This function takes a pattern and a text as input and returns the
 number of times the pattern appears in the text.
 Input
 ----------
 pattern : string
 A pattern.
 text : string
 A text.
 Returns
 -------
 pattern_count : int
 The number of times the pattern appears in the text.
 """
 Pattern_length = len(pattern)
 text_length = len(text)
 pattern_count=0

 for i in range(text_length - Pattern_length + 1):
  #loop through the text for the pattern window
  j = 0

  while(j < Pattern_length): #loop through the pattern
   if (text[i + j] != pattern[j]): #if the pattern does not match the text window
    break #break the loop
   j += 1 #increment the pattern index

  if (j == Pattern_length): #if the pattern matches the text window
   pattern_count+=1 #increment the pattern count
 return pattern_count


#Test
text = "ALLISGOODALLISGOOD"
pattern = "ALL"
```

```
assert naive_pattern_counter(pattern, text) == 2
assert naive_pattern_counter("GOOD", text) == 2
assert naive_pattern_counter("A", text) == 2
assert naive_pattern_counter("ALLISGOOD", text) == 2
assert naive_pattern_counter("L", text) == 4
```

In [209]:

```
#counting the number of times each phrase appears in each document using the naive pattern counter
#making a list of words a joined string of text
def get_string(list_of_words):
    """
    This function takes a list of words as input and returns a string of the words.
    Input
    ----------
    list_of_words : list
    A list of words.

    Returns
    -------
    string : string
    A string of the words.
    """
    string = "" #define the string
    for word in list_of_words: #loop through the words
        string += word + " " #add the word to the string with a space after each word
    return string #return the string

def count_phrases_naive(phrases_list, text):
    """
    This function takes a list of phrases and a text as input and returns
    the number of times each phrase appears in the text.

    Input
    ----------
    phrases_list : list
    A list of phrases.
    text : string
    A text.

    Returns
    -------
    counter : int
    The number of times each phrase appears in the text.
    """
    counter=0
    for phrase in phrases_list: #loop through the phrases
        counter+=naive_pattern_counter(phrase, get_string(text))
        #increment the counter by the number of times the phrase appears in the text
    return counter

#print(count_phrases_naive(phrases_version_1, version_2))
#The above is possbile but it is not efficient because it is O(n^2)
#where n is the number of words in the text.

#counting the number of times each phrase appears in each document using the naive pattern counter
#creating a plot to show that it is O(n^2)
import matplotlib.pyplot as plt
import time
import random
import string

def random_string(stringLength=10):
    """
    Generate a random string of fixed length
    Parameters
    ----------
    stringLength : int
    The length of the string.
    Returns
    -------
    string
    A random string.
    """
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters) for i in range(stringLength))

def get_random_text(length):
    """
    This function takes a length as input and
    returns a random text of the length.
    Input
    ----------
    length : int
    The length of the text.

    Returns
    -------
    text : list
    A random text of the length.
    """
    text = [] #define the text
    for i in range(length): #loop through the length
        text.append(random_string()) #append a random word to the text
    return text #return the text

def get_random_phrases(length):
    """
    This function takes a length as input and
    returns a random list of phrases of the length.
    Input
    ----------
```

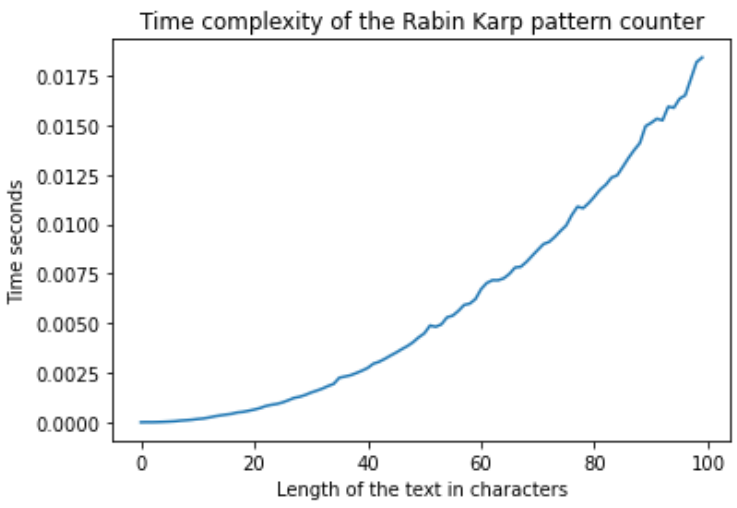
```
    length : int
    The length of the list of phrases.

    Returns
    -------
    phrases : list
    A random list of phrases of the length.
    """
    phrases = [] #define the phrases
    for i in range(length): #loop through the length
        phrases.append(random_string()) #append a random phrase to the phrases
    return phrases #return the phrases

def get_random_text_phrases(length):
    """
    This function takes a length as input and returns a
    random text and a random list of phrases of the length.
    Input
    ----------
    length : int
    The length of the text and the list of phrases.

    Returns
    -------
    text : list
    A random text of the length.
    phrases : list
    A random list of phrases of the length.
    """
    text = get_random_text(length) #get a random text of the length
    phrases = get_random_phrases(length) #get a random list of phrases of the length
    return text, phrases #return the text and the phrases

def get_time(text, phrases):
    """
    This function takes a text and a list of phrases as input and
    returns the time it takes to count the number of times each phrase appears in the text.
    Input
    ----------
    text : list
    A text.
    phrases : list
    A list of phrases.

    Returns
    -------
    time : float
    The time it takes to count the number of times each phrase appears in the text.
    """
    experiments=[]
    for i in range(31): #loop through 100 times
        start_time = time.process_time() #get the start time
        count_phrases_naive(phrases, text) #count the number of times each phrase appears in the text
        end_time = time.process_time() #get the end time
        experiments.append(end_time - start_time) #append the time it takes to count the
    return sum(experiments)/30 #return the time it takes to count the
    #number of times each phrase appears in the text

def get_time_list(length):
    """
    This function takes a length as input and returns a list of times it takes
    to count the number of times each phrase appears in the text for a random
    text and a random list of phrases of the length.

    Input
    ----------
    length : int
    The length of the text and the list of phrases.

    Returns
    -------
    time_list : list
    A list of times it takes to count the number of times each phrase appears in
    the text for a random text and a random list of phrases of the length.
    """
    time_list = [] #define the time list
    for i in range(length): #loop through the length
        text, phrases = get_random_text_phrases(i) #get a random text and a random
                                                   #list of phrases of the length

        time_list.append(get_time(text, phrases)) #append the time it takes to count the number of times each phrase
                                #appears in the text for the random text and the random list of phrases of the length
    return time_list #return the time list

def plot_time_list(time_list):
    """
    This function takes a list of times as input and plots the list of times.
    Input
    ----------
    time_list : list
    A list of times.
    """
    plt.plot(time_list) #plot the list of times
    plt.xlabel("Length of the text in characters")
    plt.ylabel("Time seconds")
    plt.title("Time complexity of the naive pattern counter")
    plt.show()

#plot the list of times it takes to count the number of times each phrase appears in the text
# for a random text and a random list of phrases of the length

plot_time_list(get_time_list(100))
```



In [211]:

```
#Rabin Karp Algorithm for Pattern Searching
d = 256 #number of characters in the alphabet of the input text
def pattern_counter(pattern, text, prime_number):
    """
    This function takes a pattern, a text and a prime number as input and
    returns the number of times the pattern appears in the text.
    Input
    ----------
    pattern : string
    A pattern.
    text : string
    A text.
    prime_number : int
    A prime number.

    Returns
    -------
    pattern_count : int
    The number of times the pattern appears in the text.
    """
    pattern_length = len(pattern)
    text_length = len(text) #length of text
    i = 0 # index of text[]
    j = 0 # index of pattern[]
    p = 0 # pattern's hash value
    t = 0 # text's hash value
    pattern_count=0


    h = 1 #defining h value
    for i in range(pattern_length-1):
        h = (h*d) % prime_number #calculating value of
        #h would be "pow(d, pattern_length-1)%q"


    for i in range(pattern_length): # getting the hash value of pattern and first window of text
        p = (d*p + ord(pattern[i])) % prime_number
        t = (d*t + ord(text[i])) % prime_number

    for i in range(text_length-pattern_length+1): #sliding the pattern over text one by one
        if p == t: #if the hash values match then only check for characters on by one
            #go through the pattern and text and check if they match one by one
            for j in range(pattern_length):
                #if the characters don't match then break
                if text[i+j] != pattern[j]:
                    break
                else: #if the characters match then increment j
                    j += 1


            if j == pattern_length: #if j is equal to the length of the pattern
                #then it means the pattern was found
                    pattern_count+=1

      #getting the hash value of the next window of text by sliding the window and removing
            # the hash value of the previous char removed from the window
            if i < text_length-pattern_length:
                t = (d*(t-ord(text[i])*h) + ord(text[i+pattern_length])) % prime_number

       # We might get negative values of t, converting it to
       # positive
                if t < 0:
                    t = t+prime_number

        return pattern_count

#testing
text = "THIS TEXT IS A TEST TEXT"
pattern = "TEXT"
prime = 101
#test the pattern_counter function
assert(pattern_counter(pattern, text, prime)== 2)

def count_phrases(phrases_list, list_of_words):
    """
    This function takes a list of phrases and a list of words as input and
    returns the number of times each phrase appears in the list of words.
    Input
    ----------
    phrases_list : list
    A list of phrases.
```

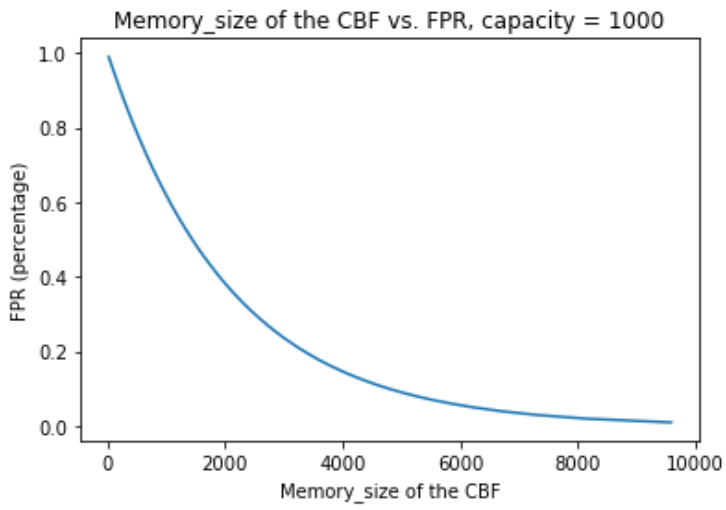
```
    list_of_words : list
    A list of words.

    Returns
    -------
    counter : int
    The number of times each phrase appears in the list of words.
    """
    counter=0
    for phrase in phrases_list:
        counter+=pattern_counter(phrase, get_string(list_of_words),prime)
    return counter

#Testing the plaigarism with this example

# print("count of common phrases of version one with itself, version two, and version three respectively:")
# print(count_phrases(phrases_version_1, version_1))
# print(count_phrases(phrases_version_1, version_2))
# print(count_phrases(phrases_version_1, version_3))
# print("count of common phrases of version two with itself, version two, and version three respectively:")

# print(count_phrases(phrases_version_2, version_2))
# print(count_phrases(phrases_version_2, version_1))
# print(count_phrases(phrases_version_2, version_3))
# print("count of common phrases of version three with itself, version two, and version three respectively:")

# print(count_phrases(phrases_version_3, version_3))
# print(count_phrases(phrases_version_3, version_2))
# print(count_phrases(phrases_version_3, version_1))


#plot the list of times it takes to count the number of times each phrase appears in the text
# for a random text and a random list of phrases of the length
plt.plot(get time list(100)) #plot the list of times
plt.xlabel("Length of the text in characters")
plt.ylabel("Time seconds")
plt.title("Time complexity of the Rabin Karp pattern counter")
plt.show()
```

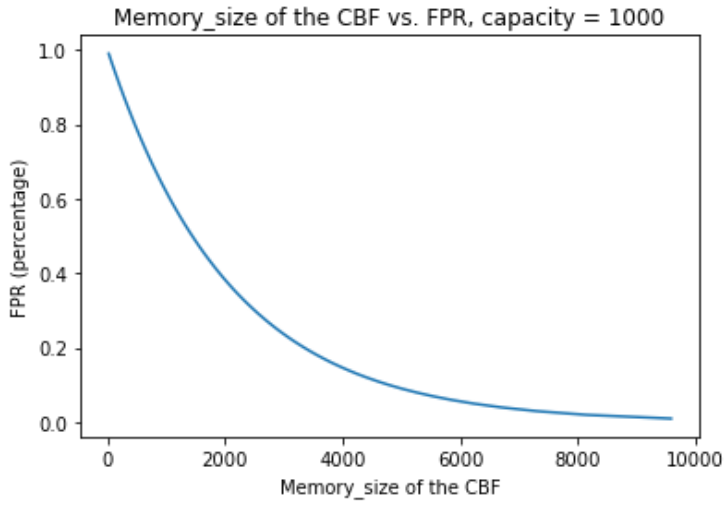


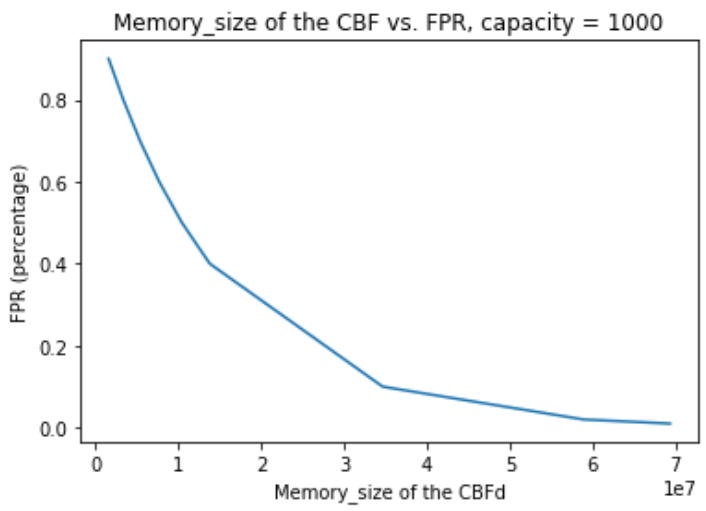
In [187]:

```
#create a plot to show how does the memory_size of cbf scale with fpr theoretically

import matplotlib.pyplot as plt
import numpy as np
capacity=1000
def plot_cbf_memory_size(fpr):
    plt.plot([(-1*round ((capacity * math.log (i)) / (math.log (2))**2)) for i in fpr],fpr)
    plt.ylabel("FPR (percentage)")
    plt.xlabel("Memory_size of the CBF")
    plt.title("Memory_size of the CBF vs. FPR, capacity = 1000")
    plt.show()

plot_cbf_memory_size([i/100 for i in range(1, 100)])

#creating the plot using bloom filters class defined above

capacity=1000
def plot_bf_memory_size(fpr):
    plt.plot([CountingBloomFilters(capacity, i).memory_size for i in fpr],fpr)
    plt.ylabel("FPR (percentage)")
    plt.xlabel("Memory_size of the CBF")
    plt.title("Memory_size of the CBF vs. FPR, capacity = 1000")
    plt.show()

plot_bf_memory_size([i/100 for i in range(1, 100)])

capacity=1000

URL="https://gist.githubusercontent.com/raquelhr/78f66877813825dc344efefdc684a5d6/raw/361a40e4cd22cb6025e1fb2baca3bf7e166b2ec6/"
#using this link and make it into a list of words
list_of_words=get_txt_into_list_of_words(URL)
phrases=get_phrases(list_of_words)
cbf_test_1=create_cbf(phrases, fprate=0.01)
cbf_test_2=create_cbf(phrases, fprate=0.02)
cbf_test_3=create_cbf(phrases, fprate=0.1)
cbf_test_4=create_cbf(phrases, fprate=0.4)
cbf_test_5=create_cbf(phrases, fprate=0.5)
cbf_test_6=create_cbf(phrases, fprate=0.6)
cbf_test_7=create_cbf(phrases, fprate=0.7)
cbf_test_8=create_cbf(phrases, fprate=0.8)
cbf_test_9=create_cbf(phrases, fprate=0.9)

list_cbf_experiment=[cbf_test_1, cbf_test_2, cbf_test_3, cbf_test_4, cbf_test_5,
```

```
cbf_test_6, cbf_test_7, cbf_test_8, cbf_test_9]

import sys
fpr=[0.01,0.02,0.1,0.4,0.5,0.6,0.7,0.8,0.9]
sizes=[]
for cbf in list_cbf_experiment:
    sizes.append(sys.getsizeof(cbf.array))

plt.plot(sizes,fpr)
plt.ylabel("FPR (percentage)")
plt.xlabel("Memory_size of the CBFd")
plt.title("Memory_size of the CBF vs. FPR, capacity = 1000")
plt.show()
```



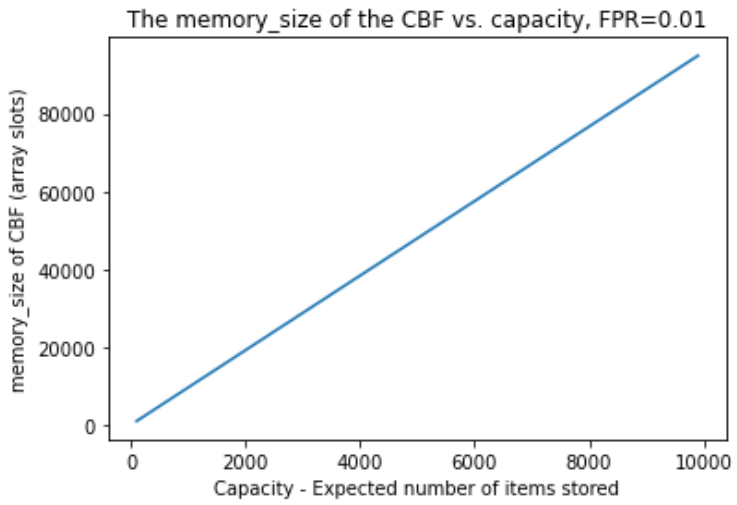


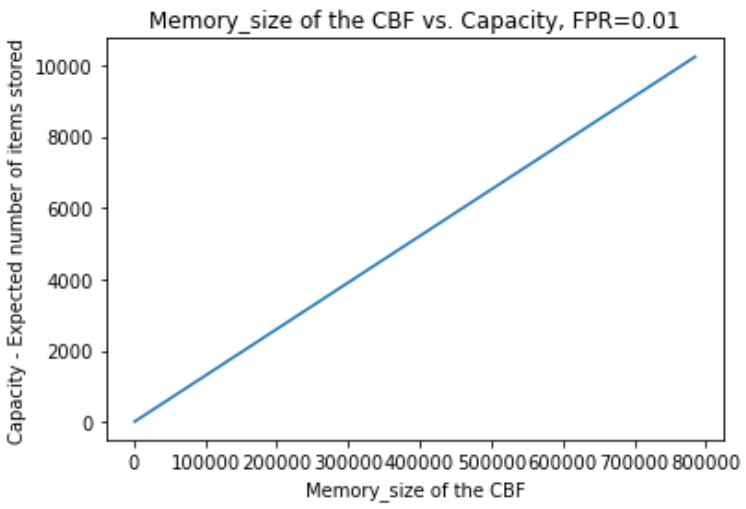


In [197]:

```
#plot how the memory_size scale with the number of items stored for a fixed fpr

def plot_memory_memory_size(fpr, capacity):
    plt.plot(capacity, [CountingBloomFilters(i, fpr).memory_size for i in capacity])
    plt.xlabel("Capacity - Expected number of items stored")
    plt.ylabel("memory_size of CBF (array slots)")
    plt.title("The memory_size of the CBF vs. capacity, FPR=0.01")
    plt.show()

plot_memory_memory_size(0.01, [i for i in range(100, 10000, 100)])

cbf_test_1=create_cbf(phrases[:20], fprate=0.01)
cbf_test_2=create_cbf(phrases[:40], fprate=0.01)
cbf_test_3=create_cbf(phrases[:80], fprate=0.01)
cbf_test_4=create_cbf(phrases[:160], fprate=0.01)
cbf_test_5=create_cbf(phrases[:320], fprate=0.01)
cbf_test_6=create_cbf(phrases[:640], fprate=0.01)
cbf_test_7=create_cbf(phrases[:1280], fprate=0.01)
cbf_test_8=create_cbf(phrases[:2560], fprate=0.01)
cbf_test_9=create_cbf(phrases[:5120], fprate=0.01)
cbf_test_10=create_cbf(phrases[:10240], fprate=0.01)


list_cbf_experiment=[cbf_test_1, cbf_test_2, cbf_test_3, cbf_test_4, cbf_test_5,
cbf_test_6, cbf_test_7, cbf_test_8, cbf_test_9, cbf_test_10]



import sys
sizes=[]
for cbf in list_cbf_experiment:
    sizes.append(sys.getsizeof(cbf.array))
```

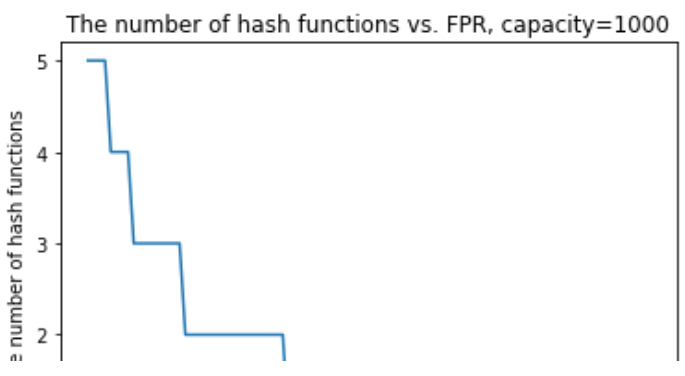
```
plt.plot(sizes,[20,40,80,160,320,640,1280,2560,5120,10240])
plt.ylabel("Capacity - Expected number of items stored")
plt.xlabel("Memory_size of the CBF")
plt.title("Memory_size of the CBF vs. Capacity, FPR=0.01")
plt.show()
```

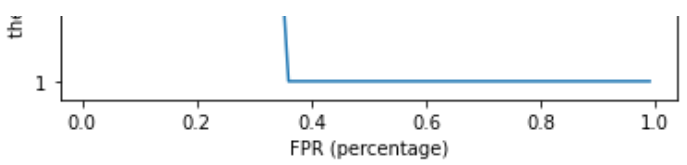


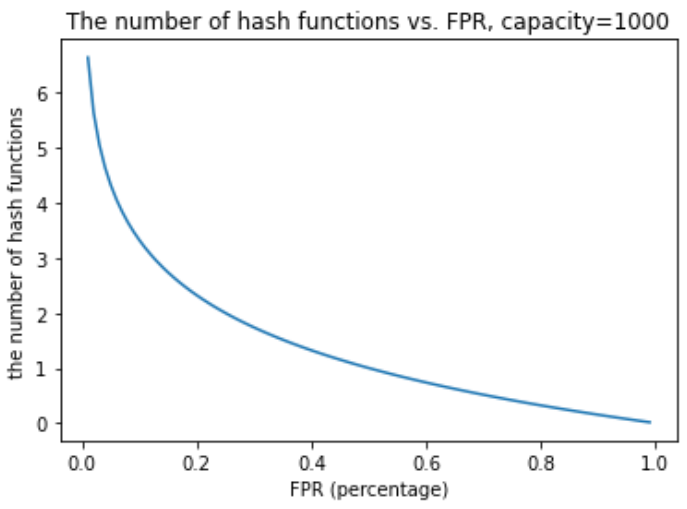


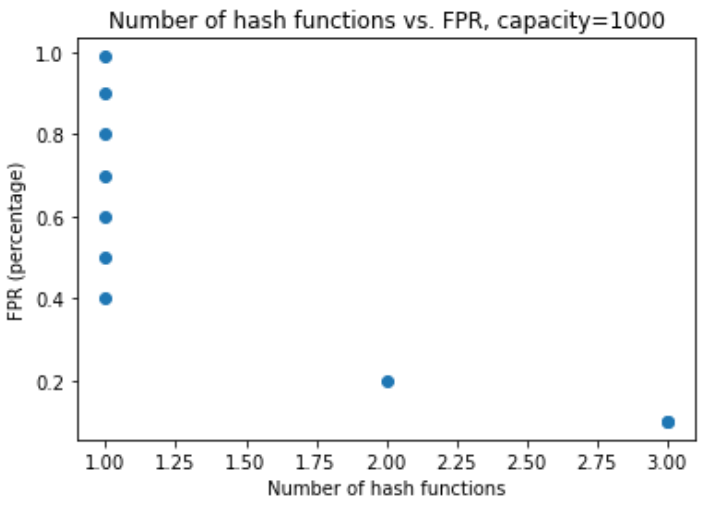
In [200]:

```
#plot how does the fpr scale with the num_hashfn in the counting bloom filter

def plot_fpr(fpr, capacity):
    plt.plot(fpr, [CountingBloomFilters(capacity, i).num_hashfn for i in fpr])
    plt.xlabel("FPR (percentage)")
    plt.ylabel("the number of hash functions")
    plt.title("The number of hash functions vs. FPR, capacity=1000")
    plt.yticks(np.arange(1, 6, step=1))
    plt.show()

plot_fpr([i/100 for i in range(1, 100)], 1000)

#plot how does the fpr scale with the num_hashfn in the counting bloom filter using that num_hashfn= - log base 2 of fpr

def plot_fpr(fpr, capacity):
    plt.plot(fpr, [-1*math.log(i, 2) for i in fpr])
    plt.xlabel("FPR (percentage)")
    plt.ylabel("the number of hash functions")
    plt.title("The number of hash functions vs. FPR, capacity=1000")
    plt.show()

plot_fpr([i/100 for i in range(1, 100)], 1000)

cbf_test_1=create_cbf(phrases, fprate=0.1)
cbf_test_2=create_cbf(phrases, fprate=0.2)
cbf_test_3=create_cbf(phrases, fprate=0.1)
cbf_test_4=create_cbf(phrases, fprate=0.4)
cbf_test_5=create_cbf(phrases, fprate=0.5)
cbf_test_6=create_cbf(phrases, fprate=0.6)
cbf_test_7=create_cbf(phrases, fprate=0.7)
cbf_test_8=create_cbf(phrases, fprate=0.8)
cbf_test_9=create_cbf(phrases, fprate=0.9)
cbf_test10=create_cbf(phrases, fprate=0.99)

list_cbf_experiment=[cbf_test_1, cbf_test_2, cbf_test_3, cbf_test_4, cbf_test_5,
cbf_test_6, cbf_test_7, cbf_test_8, cbf_test_9, cbf_test10]

number_of_h_func=[]
for cbf in list_cbf_experiment:
    number_of_h_func.append((cbf.num_hashfn))

plt.scatter(number_of_h_func,[0.1,0.2,0.1,0.4,0.5,0.6,0.7,0.8,0.9,0.99])
plt.ylabel("FPR (percentage)")
plt.xlabel("Number of hash functions")
plt.title("Number of hash functions vs. FPR, capacity=1000")
plt.show()
```









In [208]:

```
#create a function to  time the access time of to the hash values of the counting bloom filter using time.process_time()

import time
def time_access_time(fpr, capacity):

    experiemnts=[]
    for i in range(100):
        cbf_exp=CountingBloomFilters(capacity, fpr)
        start=time.process_time()
        cbf_exp.search("anything")
        end=time.process_time()
        experiemnts.append(end-start)
    return sum(experiemnts)/len(experiemnts)

#create a plot for the access time of the hash values of the counting bloom filter vs the capacity



def plot_access_time(fpr, capacity):
    plt.scatter(capacity, [time_access_time(fpr, i) for i in capacity])
    plt.xlabel("Capacity - Expected number of items stored")
    plt.ylabel("Access time to the hash values of the CBF")
    plt.title("hash values acsess time vs. Capacity, FPR=0.01")
    plt.show()


plot_access_time(0.01, [i for i in range(100, 1000000, 10000)])

#run the above experiemnt 60 times and average the results and plot those
```