CS110: Final Project

Designing a Plagiarism Detector

Minerva Schools at KGI

Designing a Plagiarism Detector

Problem Analysis

In this paper, we will build two versions of a simple plagiarism detector using two different hashing techniques and analyze their advantages and limitations by comparing and contrasting their implementation. A simple plagiarism detector is an algorithm, which can identify and quantify the similarity between two given strings. In this simple version, our outcome matric is the number of k-length substrings that is common in both the strings. More specifically, we will be given two strings x and y and a single integer k. Our algorithm should identify all the k-length common substrings in both strings and output the list of tuples which consists of the initial index of all such common substrings. We will use a hash table for this approach and two different hashing methods. First, we will implement the rolling hashing techniques to build the algorithm and later we will use a regular hashing method and compare both of them.

Hash Table

Hash table is a specific data structure optimized for dictionary operations (i.e. search, delete and insert). Suppose we have a hash table with m number of slots. If we want to insert an item we use a hash function, which takes an item and returns an integer. Then we use the output of the hash function as an index of the slot where we insert the item. So anytime we want to search an item, we just hash the item and look at the corresponding index, which works generally in O(1) time. Now, one limitation of the hash table is that, if two or more different item hashes to the same index, then we call it a conflict and there are several ways to deal with the conflict and we will see one later in the paper. If there is total n item to be hashed and total m slots in the hash

table, then on average we can expect $\frac{n}{m}$ items to be hashed in a single slot. This average-case largely depends on our choice of the hash function. A good function has to have two important properties. It should create as close as a random uniform distribution, which means every element has an equal chance to be hashed in any of the m slots. Secondly, a hash function should be quick as we call it many times for query or insert.

Rolling Hash

Rolling hashing is a hashing technique that can be used to hash each adjacent substrings of a string and by using the division method hash function. Suppose we want to hash each 3 length substring of the string: 'today is our final day' (i.e. 'tod', 'oda', … 'day'). The naive way could be going through each such substring and hash them one by one. But as they are adjacent substrings, they have a huge overlap and we can make use of this overlap by using division method hash function, i.e. 'h(k) = k mod q'. To understand how this works we can look at integer numbers like '254698'. If we know $h(254) = 254 \ mod \ 10 = 4$, we can find the next substrings' hash value by removing the first digit and adding the next digit:

$$h(546) = h(254 * 10 + 6 - 2 * 10^3) = ((h(254) * 10 + 6) \ mod \ 10 - 2 * 10^3) \ mod \ 10$$

So in this approach, we don't need to go through all the digits of the next substring, instead, we can use the previous hash value and compute the next hash value in constant (O(1)) time. Just like these decimal numbers, we can consider any string as a number of base *b* and apply the same method to use rolling hashing.

First Implementation

Now we will implement our first algorithm using the rolling hashing. As we saw in the last section, we need to consider the item (here, the substring) as a number and then hash it to get the index. To transform the string to a number, we can consider them as a digit of a b-base system. In python, every letter and symbol is expressed as an ASCII code. As there is 128 printable ASCII characters, we can define 128 as the basis of our system, then any substring consists of printable ASCII character will result in a different integer. We used the function *get_int()* to convert a substring into a 128-bit integer (see appendix for code).

Now another choice to make is the divider q in our hash function $h(k) = k \bmod q$. If we want a uniform random distribution, it's always better to take a prime number as they don't have any divisor. Then as long as it is optimal to use big spaces, we should use a large hash table because large hash tables mean low frequency of collision (i.e. on the average low number of items in a single slot). If we have n number of items, we should take at least n^2 total slots so that the frequency of collision becomes $\frac{n}{n^2} = \frac{1}{n}$. But having a large hash table unnecessarily can be wasteful as most of the place will be empty. So we used the divider as our table size because if our table size is smaller, we need to take the modulus again to find the corresponding index and if the table size is larger, then the slots larger than q and less than the table size will be always empty. We used a large prime number $10^6 + 3$ and take the maximum of this number and n^2 as our table size and the divider, where n is the length of the string x.

Now the major steps of our algorithm are the following:

1. We will find the hash value of the first substring of x using the division method, which will take O(k), where k is the length of each substring as we need to go through each element of the substring. Insert the item in the hash table
2. Then we will use the rolling hashing to find the hash value for all other substrings each in constant time, so total in O(n-k) time. Insert them to the hash table.
3. Now find the hash value for the first substring of string y and check the corresponding index of the hash table.
4. If there is anything stored, then check if the substring is equal to the substring of y, if yes store their first indices as a tuple in the answer list.
5. Use rolling hashing to find the hash value for other substrings of y and repeat step 4.

As a data structure, we used a python list where we can access the item later by index number which is simply the hash value of the item. Another choice could be a python dictionary, but there we need to store the index value as a string. In step 2, there are two extra things to consider. How should we handle the collision and how we will store the information of the first index of each substring in string x.

To have the information of the first index, we decided to add a tuple in the hash table, where the first element is the first index of the substring and the substring will be the 2nd element.

Here we used a chaining method to handle the collision. Other choices were open addressing or the cuckoo hashing. One problem with open addressing was if we had the same substrings at two different places, this addressing will put the words in a different slot as the first

element of the tuple is different. Then while searching, we cannot be sure just by checking one slot that there was no repetition on this word. Suppose, ‘todayis monday’, here the two tuples for ‘day’ will be (2, ‘day’) and (10, ‘day). Open addressing would put them in a different place and while searching we need to go through all possible slots to ensure that we get all the ‘day’. So they are better if we want to query a unique search but not in this case.

Example

Suppose our two given string x and y are the following:

x = ’todayisourfinalday’

y = ‘thisisfriday’

And k = 3

The algorithm steps will be the following:

1. Get an integer value for ‘tod’ using the *get_int()* function:

   int(‘tod’) = 1914852

2. Store it by taking mode with the divider, q = $10^6 + 3$

   key = 1914852 % $10^6 + 3$ = 914849

3. Store it in the hash table:

   Append (0, ‘tod’) in hash_table[914849]

4. Find all the next substrings using rolling hashing using the following equation:

   # adding the next letter

   new_key = (new_key * base + ord(x[k+i])) % size (here ord(x) is the ASCII for x)

   # removing the first letter

new_key = (new_key - ord(x[i])*(base**k % size)) % size

So we will find the hash value for ‘oda’, ‘day’ …. ‘day’ and append them with their initial index in the hash table.

5. Now, we will find the hash value for the first string of y:

   key = int(‘thi’) % q = 913958

6. Check if there is anything store in that index. If not then continue. If yes, then check the second element of the all tuples stored there. If any match found then add the x-index (first element of the tuple) and the y-index in a tuple and store it in the answer list.

Input: print(rh_get_match('todayisourfinalday', 'thisisfriday', 3))

Output: [(2, 9), (15, 9)]

As expected the only 3-length common substring is ‘day’ which occurs twice in x at 2nd and 15th index and once in y at 9th index.

Second Implementation

Now we will implement the same algorithm but now we will not use the rolling hashing or any division method hashing. But other than our hash function, everything should be the same. Again we need to change our substrings to an integer, which we did use the same function as the last one and we keep the base as 128. Then again to store the tuple with the substring and its initial index, we used the python list and we used the chaining method to handle any collision.

Now as a hash function we choose the multiplication method and used Knuth’s suggestion on using the golden ratio. The hash function is the following:

$h(k) = floor(m * frac(c * k))$

Where c is between 1 and 0 and taking it as a golden ratio works quite fine, $c = (\sqrt{5} - 1)/2$

As a hash function need to be quick and has to ensure random uniform distribution. I checked it with the precious division method and it was quite similar though not perfect, both shows

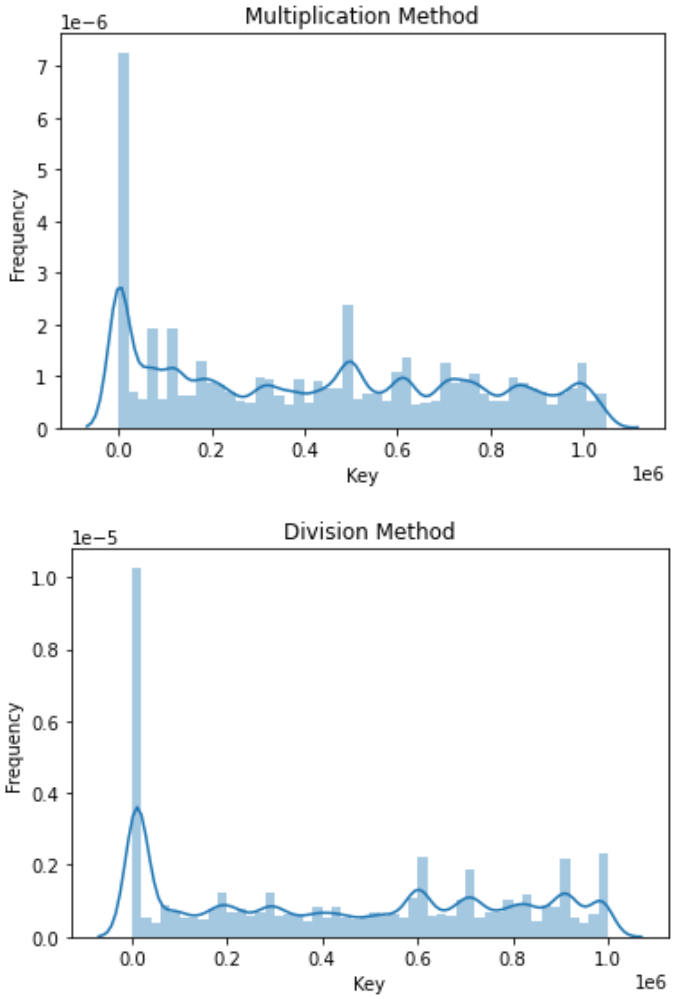


Figure 1: The histogram of both division and multiplication method showing their almost uniform distribution except near 0 value, where keys are concentrated

a large portion of the key near 0 value. We also checked how fast does it work and it is almost as fast as the division method. In the multiplication method, it is better to choose a size that is the power of 2. So we choose 2 to the power 20.

Here our major steps are the followings:

1. We will find the hash value of every substring of x using the division method, which will take O(k)*O(n) = O(nk), where k is the length of each substring as we need to go through each element of the substring. Insert them in the hash table.
2. Now find the hash value for the first substring of string y and check the corresponding index of the hash table.
3. If there is anything stored, then check if the substring is equal to the substring of y, if yes store their first indices as a tuple in the answer list.
4. Repeat 3 and 4 until all the substrings of y are checked.

Example

Suppose our two given string x and y are the following:

x = ’todayisourfinalday’

y = ‘thisisfriday’

And k = 3

The algorithm steps will be the following:

1. Get an integer value for ‘tod’ using the *get_int()* function:

   int(‘tod’) = 1914852
2. Store it by taking the hash function

   key = math.floor(size * frac(*get_int*('tod') * c))= 649514
3. Store it in the hash table:

Append (0, ‘tod’) in hash_table[649514]

4. Repeat 1 to 3 step using the multiplication hash function
5. Now, we will find the hash value for the first string of y:
6. Check if there is anything store in that index. If not then continue. If yes, then check the second element of the all tuples stored there. If any match found then add the x-index (first element of the tuple) and the y-index in a tuple and store it in the answer list
7. Repeat 5 and 6 until all substrings are finished.

.

Input: print(regular_get_match('todayisourfinalday', 'thisisfriday', 3))

Output: [(2, 9), (15, 9)]

Complexity Analysis

First Implementation

Let’s look at every step again for the first implementation:

1. We will find the hash value of the first substring of x using the division method, which will take O(k), where k is the length of each substring as we need to go through each element of the substring. Insert the item in the hash table
2. Then we will use the rolling hashing to find the hash value for all other substrings each in constant time, so total in O(n-k) time. Insert them to the hash table.
3. Now find the hash value for the first substring of string y and check the corresponding index of the hash table. it will again take O(k).

4. If there is anything stored, then check if the substring is equal to the substring of y, if yes store their first indices as a tuple in the answer list. Checking will take O(k)
5. Use rolling hashing to find the hash value for other substrings of y and repeat step 4. O(n)

Now step 4 is the crucial step here. On average for a large enough dataset, our table size will be at minimum n^2 , where n is the size of the string. So for a random string, on average there will be O(1/n) string. So total checking time for all O(n) substring of y will be O(n)*O(1/n)*O(k) = O(k)

But if both strings are actually totally similar, then every time it will find at least a substring in its hashed value: thus total substring per slot = O(1). Now the total checking time will be O(n)*O(1)*O(k) = O(kn) and it will dominate the asymptotic behavior.

Again if both strings are consist of the same letter, all of the substrings of x will be the same, so they all will be hashed at the same slot. Now as y string also consists of the same letters, they all will be checking that single slot. So, the total checking time will be O(n) for hashing all y substring, O(n) for all the substring on that slot and O(k) for checking each substring: together $O(kn^2)$ the dominant complexity and it will be way more inefficient when all or most of the substrings of both strings will be similar.

In a real-world context, it is very less likely to happen that both strings will have no other letters except only one. So, we may consider that this worst-case scenario will not be abundance

but the previous one is very much common when all or most of the strings will be similar. So it could be a good idea to consider adding some corner cases or improve the algorithm strategy to capture these special cases and decrease the time complexity of the algorithm.

In terms of memory complexity, we are using at least n^2 space for the hash table and we are wasting most of its space as we have at most O(n) substring. Though this large space complexity is a tradeoff for having a low average collision rate, not always it is feasible to have such a big space. So it would be worth thinking to find any other way to balance this tradeoff. One potential way is to add two hash functions instead of one and both using the rolling hashing (i.e. possibly changing the base and the divider). We can store the secondary hash value as another element of the tuple and store the three-element tuple in our hash table. While checking we will compute both the hash function and first hash value will give us the index of the hash table. Then we can check if the 2nd hash value matches the value in the tuple, only then the expensive O(k) checking will be done. In that case, we can have an O(n) hash table size, so there will be O(1) item in each slot. But we will compare the 2nd hash value first and the probability that both hash value will be matched is $1/n * 1/n$. So for all O(n) substring of y, the checking will be necessary only for $O(n * 1/n * 1/n) = O(1)$ time and the checking time is still O(k). So, we can still have O(k) total checking time while reducing so much space size by just adding a hash function.

2nd Implementation

Let's look at the major steps of the second implementation:

1. We will find the hash value of every substring of x using the multiplication method, which will take O(k)*O(n) = O(nk), where k is the length of each substring and n is the length of the string as we need to go through each element of the substring and we have a total of O(n) substring. Insert them in the hash table.
2. Now find the hash value for the first substring of string y and check the corresponding index of the hash table.O(k) for each of the hashing.
3. If there is anything stored, then check if the substring is equal to the substring of y, if yes store their first indices as a tuple in the answer list. Checking with one substring of y with one stored substring will take O(k).
4. Repeat 3 and 4 until all the substrings of y are checked. The total number of substring O(n).

Now again, the 3rd step is crucial here. Again as we have $O(n/n^2)$ number of slots, for a random string, on average we expect to have $O(n/n^2) = O(1/n)$ number of substrings in a single slot. But in this implementation, the hashing of all substrings of y takes O(nk) time just like it takes for all substrings of x. So, the total checking time becomes O(nk) hashing of y substrings, O(1/n) number of stored substring, and O(k) for checking one substring: together $O(nk) * O(1/n) * O(k) = O(k^2)$. The length of the substring, k is always smaller than the length of string, n. Thus, $k^2 < kn$. So, the dominant complexity will be the first step O(nk).

Then if we found two similar string, then there will be always at least O(1) substring stored in the hashed value of each y substrings. So for a total of O(n) y substrings, which takes O(nK) to hashes, will find O(1) stored substring and checking will take O(k) each: total complexity $O(nk) * O(1) * O(k) = O(nk^2)$ which will be dominant in the algorithm and will be responsible for its asymptotic growth.

Lastly, if we encounter similar both strings which consist of a single letter, then all of the O(n) substrings of x will hash at the same slot. Then each of the O(n) substrings of y will also be hashed there in O(kn) times. They will check all the stored substrings one by one and the total dominant complexity will be $O(nk) * O(n) * O(k) = O(n^2k^2)$.

Nature of the Input	Complexity of rolling hashing	Complexity of regular hashing
Both random strings	$O(n)$	$O(nk)$
Both strings are the same	$O(nk)$	$O(nk^2)$
Two same strings consist of the same letter	$O(n^2k)$	$O(n^2k^2)$

Table 1: Based on our above discussion, the theoretical time complexity of both implementations of the algorithm are dependent upon the nature of the input.

The discussion on having a huge space complexity of n^2 also applies in this implementation and we can use two different hash functions to balance the tradeoff of lower time complexity in random input.

Experimental Analysis

For experimental analysis, we generated the following different nature of inputs:

1. Shakespeare: I concatenate all the words from shakespeare.txt
2. Random: randomly generated words
3. Same Letter: Two different string each consists of only one letter
4. Random but same: Same two strings, which is generated randomly
5. All Same: Same two strings and same

We can extend the table 1 to include the expected asymptotic growth for a fixed substring length (k) or a fixed string length (n) (Table 2).

Nature of the Input	Complexity of rolling hashing			Complexity of regular hashing		
		n constant	k constant		n constant	k constant
Both random string	$O(n)$	$O(1)$	$O(n)$	$O(nk)$	$O(k)$	$O(n)$
Both strings are the same	$O(nk)$	$O(k)$	$O(n)$	$O(nk^2)$	$O(k^2)$	$O(n)$
Two same strings consist of the same letter	$O(n^2k)$	$O(k)$	$O(n^2)$	$O(n^2k^2)$	$O(k^2)$	$O(n^2)$

Table 2: The theoretical time complexity of both implementations of the algorithm depending on the nature of the input and when n or k is constant.

The 1st and 2nd input goes mainly in the first row, then the 4th and 5th input are respectively 2nd and 3rd row of the table. The 3rd type input has same letter in each string, so every substring will hash at the same slot but as the strings are different the checking will most probably go to the slot where substring from x is stored. So, it will be similar to the first row of the table.

Now to find the experimental growth, we change the number of string and substring and use all of the five input types to find the time for both regular and rolling hashing. We iterate

each set 10 times and take the average to get a smooth graph. As the time taken for each calculation was quite high, we take a set of six n_substring (from 10 to 60) and six n_string (from 100 to 600). (Appendix B: section I)

Before we start discussing, one important thing to note is that, surprisingly, all the generated graphs are too much inclusive of any pattern. One major reason for this could be, possibly we need more iteration and more continuous steps instead of six distinct value of n_string or n_substring. Another possible reason might be that we need more large number of string and substring to visualize the asymptotic behavior properly. But the second obstacle is hard to mitigate as both versions take a good amount of time to compute and from the theoretical analysis, it should increase for large n (and/or k for some cases too) also for very large number, the get_int() function started to showing error as we crossed the limit for storing an integer value.

We will discuss three graphs here, but all 10 graphs can be found on appendix B: section H. First for the random string, where n is constant, we expect to have O(k) for regular and O(1) for the rolling hash. But from the graph, the first thing to notice is that rolling takes a larger time than the regular which is unexpected (Figure 2). If our graphing was correct, then the possible reason could be the reasons we discussed above: low number of k or lack of more continuous steps. If we look closely, except the part from 20 to 30 n_substring, rolling hash seems to stay quite constant when n_substring is changing but regular hashing is increasing slightly other than the 20-30 part, where the decreased sharply.

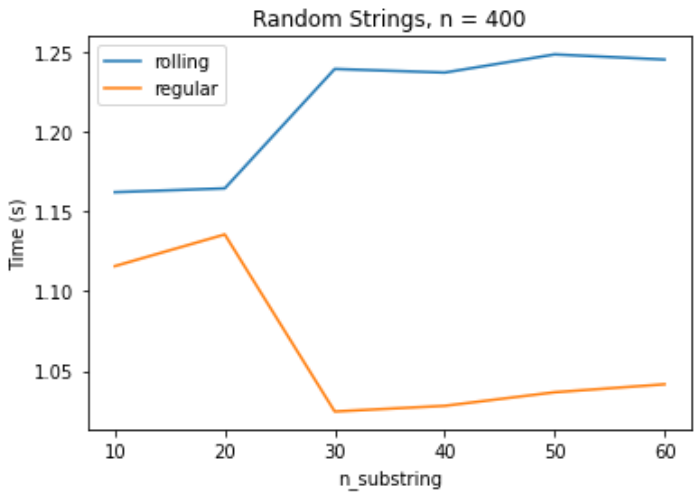


Figure 2: Rolling and regular hashing time for a constant n, when inputs are two random strings.

For same string consists of random letters (input type 4), we expect O(k) for rolling and $O(k^2)$ for regular hash for a constant n, but in the figure, the pattern is not obvious. The rolling hash seems to increase linearly upto k = 40, then it decreases a bit. But regular hashing doesn't show any continuous feature (Fig. 3). Also the time for the regular hash increases when the strings are similar.

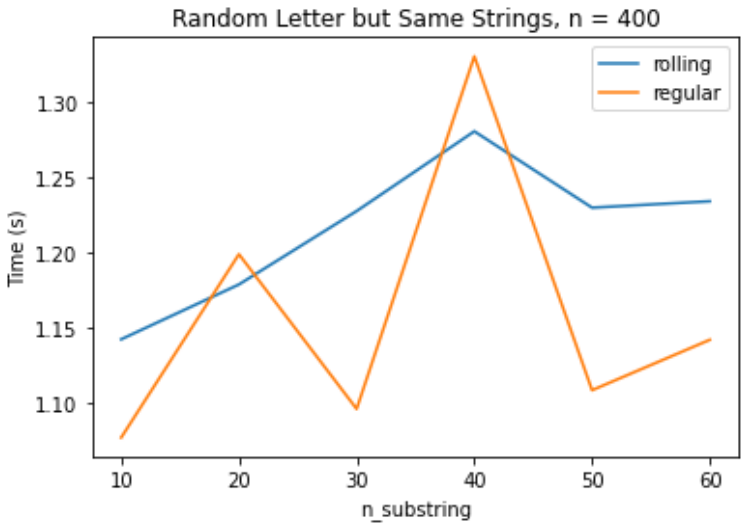


Figure 3: Rolling and regular hashing time for a constant n, when inputs are two same strings consist of random letters.

Now, for all_same strings (input type 5), the asymptotic growth for both hash should stay same as their growth for the same string (input type 4) when the n_string, n is constant: O(k) and $O(k^2)$ respectively. But in the graph, rolling hash shows almost constant feature, with a few slight linear growths, while the regular hash pattern is still inclusive. The time for regular hash increases more, while the rolling hash stays quite the same in the time range compare to other input types (Fig. 4).

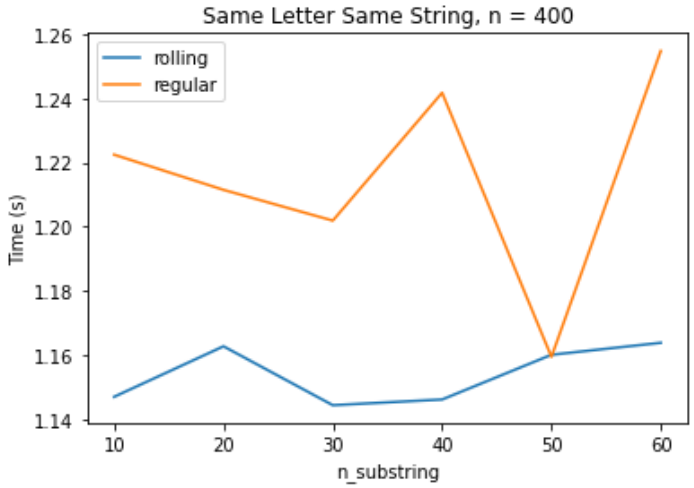


Figure 4: Rolling and regular hashing time for a constant n, when inputs are two same strings consist of the same letter.

For all input types, the regular and rolling hash should show a similar pattern when the number of substrings, k is constant: O(n) for the first four and $O(n^2)$ for the last one. Though they both broadly follow each other when k is constant, but the pattern is mostly inconclusive and not matched with the expectation, for example, for input 5, we can see almost linear growth. Another common feature for rolling hashing, when k is constant, that the time first increases with n_string followed by a fall in time. We should expect something like this pattern for rolling hash when n is constant and for input 1-3: instead of an O(1) it should decrease initially followed by

an increase. The reasoning is the complexity for this case is O(n-k) for hashing and O(k) for checking. So, when n is constant the dominant O(n-k) decreases and when (n - k) becomes less than k, O(k) starts dominating the growth. But no such pattern is found.

Real-Life Application

Two of the major limitations of both of the algorithm for a real-life use has already discussed on the previous section:

1. The time complexity of both implementation largely depends on the type of input and the average time complexity is mostly applicable foe random generated input. For a real-life application, especially when we don't know the nature of the input, we should prefer such algorithm which has low variability across different nature of output or the one which has the best worst-case time complexity. But if we know the specifications of the input, we may incorporate that information to our decision selection for choosing the most efficient one. Here, for all the inputs, rolling hashing is better than the regular hashing from at least the theoretical analysis.
2. The second major limitation is the use of excessive memory which might not be a suitable choice in real-life applications. As we already explained use of more than one hash function can help us mitigate the problem while maintaining low collision probability.
3. As we can see in figure 1, both the division method and the multiplication method generates the key close to 0 with a higher average frequency. This is bad for a hash function as it increases the chance of more collision thus slow the efficiency. So we could

try to find some more hash functions which can have a better uniform distribution while also maintaining the quick hashing time.

4. If we have a word more than once, it is stored as a separate tuple in the slot. Then while checking we need to check all possible stored tuple in the slots. One improvement could be to ensure that if the same substring comes twice, we don't insert it as another tuple, rather just store the information of its index in the same tuple. So, while checking we just need to match once and if it matches we can take all the index where it occurred in x string.

In order to use it in real life for plagiarism, we need to do some preprocessing. After taking the documents, we need to concatenate all the words without white spaces. Now, one decision choice is to define the substring length k. Having a very high number of k will miss the many common substrings. When there are many similarities, it will also speed up the time as the complexity will be similar to the 2nd column of the above-mentioned complexity table, thus depends on k. Then again, having a very low key will not give enough information on the similarities. Suppose, for k=1, we will always have 100% match, then if k=2, then there would still be many random matches, which can happen for any two string. So we need to choose an optimal length for the substring to get a better understanding, one possible way could be to find the average length of a word in that document and use it as our k.

Now another question is, how will we quantified the similarity. Our algorithm just outputs all the possible index of the common substrings. So we can find the length of the output list to find the total number of matched k-grams. If every word is similar, then we would expect

(n-k) total match, which could be used as our highest standard, because anything more than that could possibly mean that we have two similar string and some strings appear more than once in a string. The highest possible limit is when we will have a single letter in all values of both of the string, the length will be $(n - k + 1)^2$. But then one pitfall is, it is also possible that one string is consists of a single letter, thus all of its substrings are the same and there is only one substring in the second string that is similar to this substring. Now we cannot say that we have a good overall similarity between two strings, but still, the total length will be (n-k+1) as that 1 substring matches with (n-k+1) all substrings in first string.

So, though we can have a sense of similarity by examining the length of the common substring list, it can be an overestimate, if one string appears more than once in the string. But it will never be an underestimate as it will give a low number of matching only if the strings are not similar enough.

Suppose we have a source code and then we have the submission of 10 students which we want to match with the source code. We can find the number of matching (i.e. length of the output) for each pair (source code with one submission). We can rank them by setting the lowest match as 0 (0%) and the highest match as (n-k+1) (100%). Now, the pair who will have a higher ranking, they can be actually very similar or they can happen due to the repetitions of a common substring in either of the string. But the pair who will have low ranking, we can be certain that there is no plagiarism among them. So, we can set a cutoff based on the confounding variables (i.e. how common are the repetitions in the source code) and all pair whose matching number is lower than the cutoff, will have no plagiarism. For the pair with a high matching number, we can do a further investigation.

One possible further investigation can be to make the set of the number in the 1st element of all tuples in our output list and another set for the 2nd element. If we have repetitions, then the difference in the length of these two sets will be higher and if there is no repetitions, then the difference will be much lower. For example, if our total length of the matching is (n-k+1), then the highest possible difference is (n-k), where one set has (n-k+1) different index, where the second set has only 1 index, this will show that the higher number is due to repetitions. The lowest possible difference is 0 when there is no repetition. So, we can express the number of repetitions as the ratio of the number of (total matching-1) and the lower the ratio, the plagiarism is higher for that pair.

Appendix A:

HC index:

#designthinking: Learned the rolling hashing technique and incorporate them in my algorithm. Go through multiple iterations and debug to finish the first version using the previous knowledge of hash table and using the first version, I build up the 2nd version of the plagiarism checker.

#modelling: I used different hashing methods to model two algorithm and I analyzed the situations where the algorithm will be useful and where it will not. I also discussed the limitations, weaknesses, strengths, and advantages of both of the versions and the pitfalls to using in a real-life context.

#**critique**: Analyzed different nature of input and how they can influence efficiency. Analyzed the possible pitfalls and limitations of both of the algorithms, especially in real life. Explained how possibly we can improve upon that.

Appendix B: Code Snippets

Notebook Link:

https://colab.research.google.com/github/mahmud-nobe/CS110_Assignments/blob/master/CS110_Final_Project.ipynb

A: *get_int() function:*

```
# rolling hash Function
def get_int(x, base = 128):
    """
    Returns the integer value of a string
    using a specific base
    Input:
    x: a string
    base: the base of the integer we want to
    convert it into. Default is 128
    """

    int_x = 0
    n = len(x)-1

    # the first digit is the most significant digit
    # so will be multiplied by len(x)-1
    # the next letters are lesser significant
    for i in range(len(x)):
        int_x += ord(x[i])*base**n
        n -= 1

    return int_x
```

B: *rh_get_match() function:*

```
def rh_get_match(x, y, k):
    """
    Finds all common length-k substrings of x and y
    using rolling hashing on both strings.
```

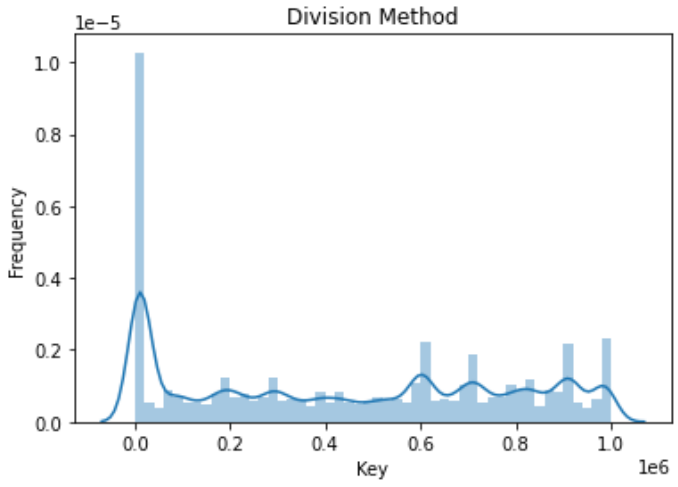
```
    Input:
    - x, y: strings
    - k: int, length of substring
    Output:
    - A list of tuples (i, j) where x[i:i+k] = y[j:j+k]
    """
    base = 128
    size = max(len(x)**2 - 1, 10**6 + 3)
    hash_table = [[] for i in range(size)]

    if(k > len(x) or k > len(y)):
        print('Warning: substring length is larger than the string
length')
        return []

    # hash function to hash item and store accordingly
    key = get_int(x[:k]) % size
    hash_table[key].append((0, x[:k]))

    new_key = key
    for i in range(len(x)-k):

        # adding the next letter
        new_key = (new_key * base + ord(x[k+i])) % size

        # removing the first letter
        new_key = (new_key - ord(x[i])*(base**k % size) ) % size

        # store the item at corresponding slot
        hash_table[new_key].append((i+1, x[i+1: k+i+1]))

    common_string = []

    ### checking with the substrings of y

    # compute the hash value for first substring in y
    checking_key = get_int(y[:k]) % size
```

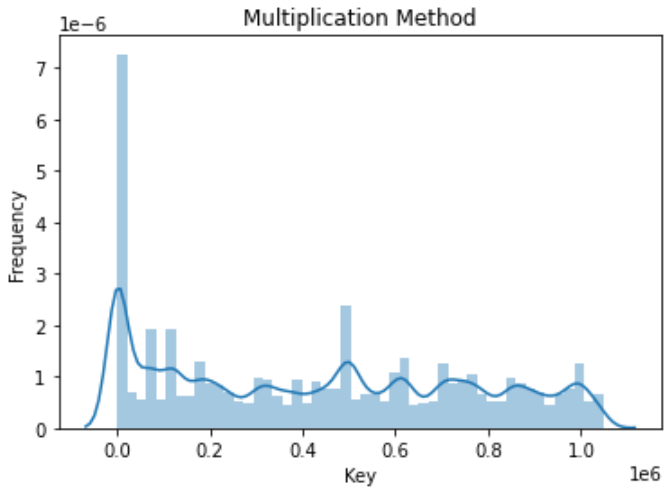
```
    # if any tuple exist in the slot, go through all the stored tuple
    # and check their 2nd element, if match, take their 'x-index' from the
    # first element and store it with 'y-index'.
    if(hash_table[checking_key]):

        for j in range(0, len(hash_table[checking_key])):
            stored_word = hash_table[checking_key][j]
            if stored_word[1] == y[:k]:

                common_string.append((stored_word[0], 0))

    # using rolling hashing to find the hash value for all other substring
    new_key = checking_key
    for i in range(len(y)-k):

        # adding the next letter
        new_key = (new_key * base + ord(y[k+i])) % size

        # removing the first letter
        new_key = (new_key - ord(y[i])*(base**k % size) ) % size


        # if any tuple exists in the slot, go through all the stored tuple
        # and check their 2nd element, if match, take their 'x-index' from
        # the first element and store it with 'y-index'.
        if(hash_table[new_key]):

            for j in range(0, len(hash_table[new_key])):
                stored_word = hash_table[new_key][j]
                if stored_word[1] == y[i+1: k+i+1]:

                    common_string.append((stored_word[0], i+1))

    return common_string
    ## your code here
```

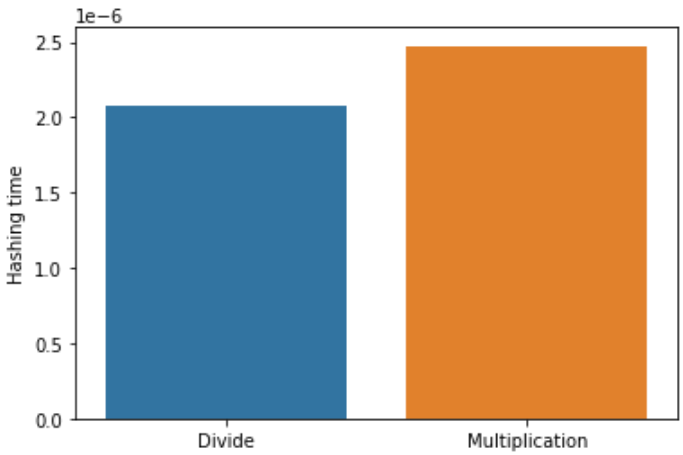
C: *Testing the rh_get_match()*

```
# test 1
x, y, k = 'todayisourfinalday', 'thisisfriday', 3
print(rh_get_match(x,y,k))
[(2, 9), (15, 9)]

# test 2 - same word
x, y, k = 'stayhometobesafe', 'stayhometobesafe', 3
print(rh_get_match(x,y,k))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8),
(9, 9), (10, 10), (11, 11), (12, 12), (13, 13)]

# test 3 - same word same letter
x, y, k = 'bbbbbbbb', 'bbbbbbbb', 4
print(rh_get_match(x,y,k))
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (0, 1), (1, 1), (2, 1), (3, 1),
(4, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (0, 3), (1, 3), (2, 3),
(3, 3), (4, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]

# test 4 - one string is empty
x, y, k = '','bbbbbbbb', 4
rh_get_match(x,y,k)
[]


# test 5 - the substring length is higher
x, y, k = 'asdfgh','bbbbbbbb', 8
rh_get_match(x,y,k)
[]
```

D: ***Comparing Hash Function:***







E: ***regular_get_match()***

```
import math

def frac(x):
    '''
    returns the fractional value of a float
    '''
    return x - math.floor(x)


def regular_get_match(x, y, k):
    """
    Finds all common length-k substrings of x and y
    not using rolling hashing on both strings.

    Input:
    - x, y: strings
    - k: int, length of substring
    Output:
    - A list of tuples (i, j) where x[i:i+k] = y[j:j+k]
    """

    # defining base, siz, c and hash table
    base = 128
    size = max(len(x)**2, 2**20)
    c = (math.sqrt(5) - 1)/2
    hash_table = [[] for i in range(size)]

    # hash function to hash the all substring and store acc. to the hash
value
    for i in range(len(x) - k + 1):

        key = math.floor( size * frac(get_int(x[i: k+i]) * c))
        hash_table[key].append((i, x[i: k+i]))

    common_string = []
```

```
    ### checking with the substrings of y

    # using regular hashing to find the hash value for all other substring
    for i in range(len(y) - k + 1):

        checking_key = math.floor( size * frac(get_int(y[i: k+i]) * c))

        # if any tuple exist in the slot, go through all the stored tuple
        # and check their 2nd element, if match, take their 'x-index' from
        # the first element and store it with 'y-index'.
        if(hash_table[checking_key]):

            for j in range(0, len(hash_table[checking_key])):
                stored_word = hash_table[checking_key][j]
                if stored_word[1] == y[i: k+i]:
                    common_string.append((stored_word[0], i))

    return common_string
    ## your code here
```

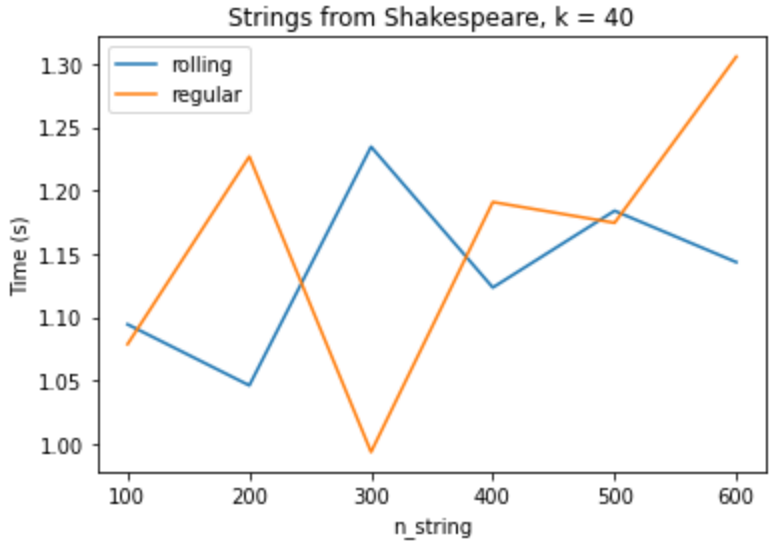
F: ***Testing the regular_get_match()***

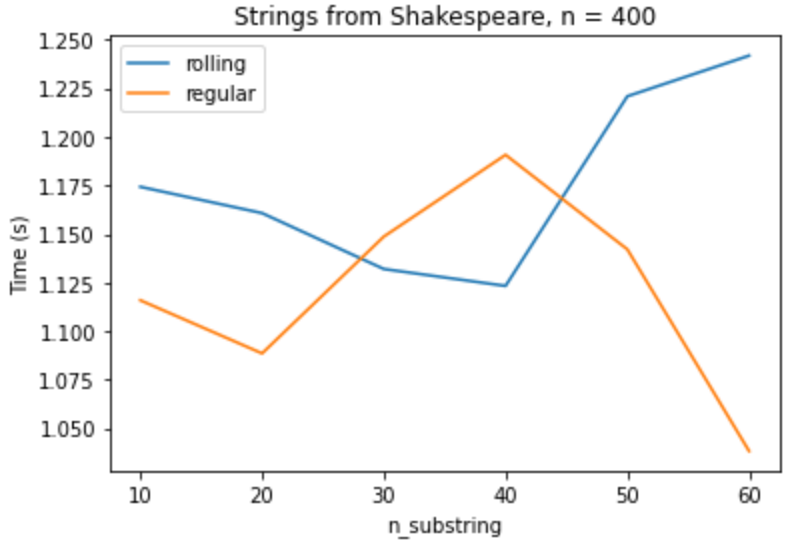
```
# test 1
x, y, k = 'todayisourfinalday', 'thisisfriday', 3
print(regular_get_match(x,y,k))
[(2, 9), (15, 9)]

# test 2 - same word
x, y, k = 'stayhometobesafe', 'stayhometobesafe', 3
print(regular_get_match(x,y,k))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8),
(9, 9), (10, 10), (11, 11), (12, 12), (13, 13)]

# test 3 - same word same letter
x, y, k = 'bbbbbbbb', 'bbbbbbbb', 4
print(regular_get_match(x,y,k))
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (0, 1), (1, 1), (2, 1), (3, 1),
(4, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (0, 3), (1, 3), (2, 3),
(3, 3), (4, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]

# test 4 - one string is empty
x, y, k = '','bbbbbbbb', 4
regular_get_match(x,y,k)
[]


# test 5 - the substring length is higher
x, y, k = 'asdfgh','bbbbbbbb', 8
regular_get_match(x,y,k)
[]
```

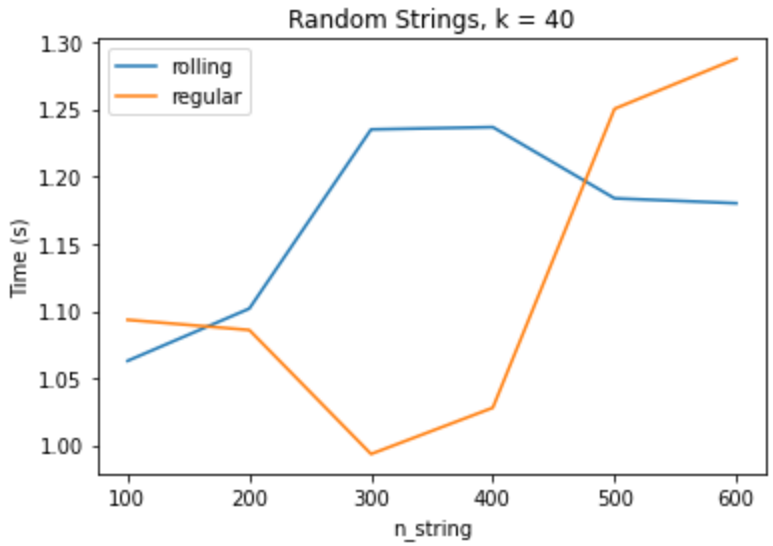
G: *String Generator of different Nature*

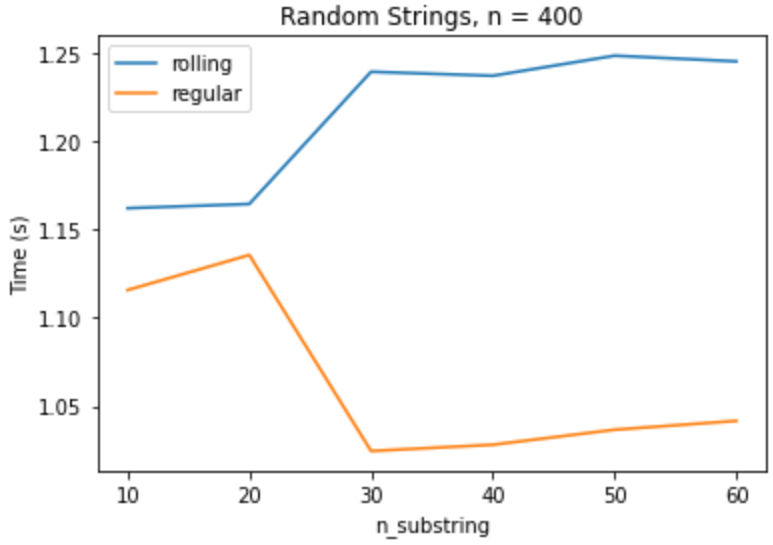
```
# reading the words from the shakespeare file

txt_file = open("t8.shakespeare.txt", "r")

entries = txt_file.read().split(' ')
lines = [string.replace('\n', '') for string in entries]
all_text = [line for line in lines if line != '']


# random word generator
def randomword(length):
    '''
    Return random word of given length
    '''
    return ''.join(random.choice(string.ascii_lowercase) for i in
range(length))

# same letter word generator
def sameletter(length):
    '''
    Return same letter word of given length
    '''
    letter = random.choice(string.ascii_lowercase)
    return ''.join(letter for i in range(length))
```

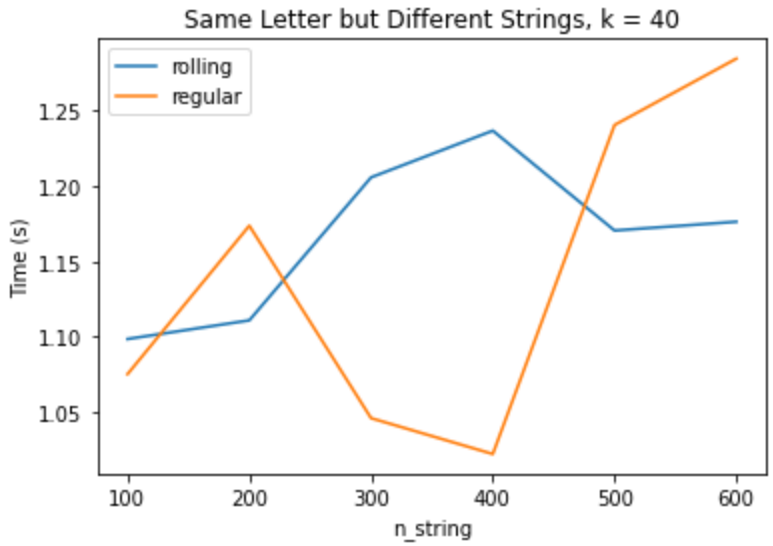
H: Experimental Complexity

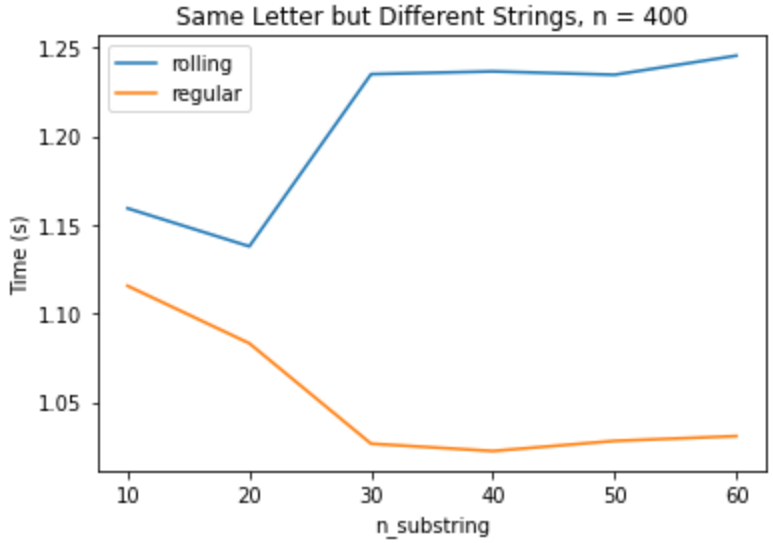


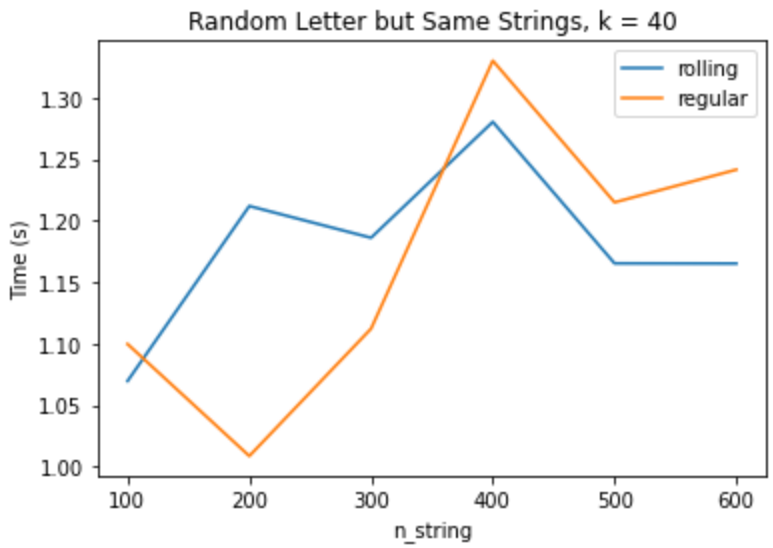


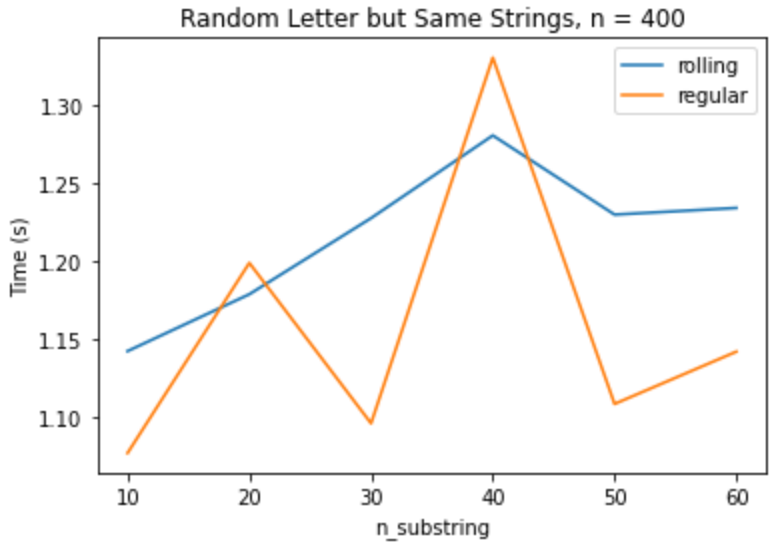

Random Strings, k = 40
rolling
regular
Time (s)
1.30
1.25
1.20
1.15
1.10
1.05
1.00
100
200
300
400
500
600
n_string

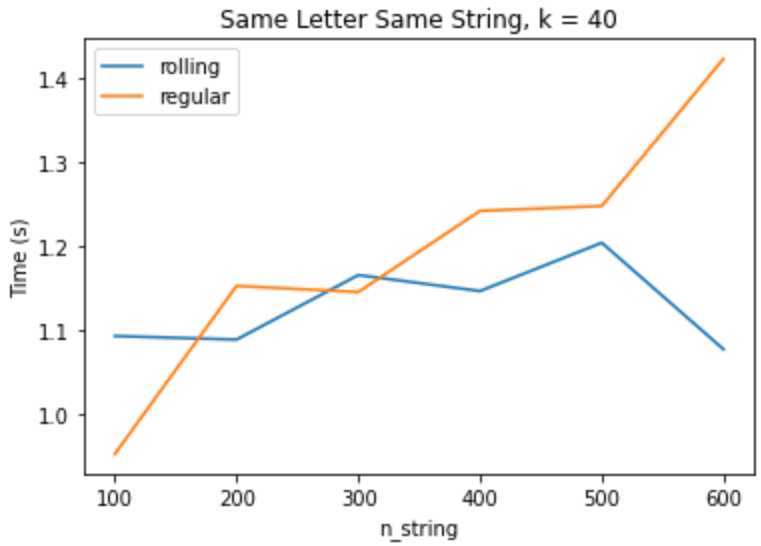

Random Strings, n = 400
rolling
regular
Time (s)
1.25
1.20
1.15
1.10
1.05
10
20
30
40
50
60
n_substring

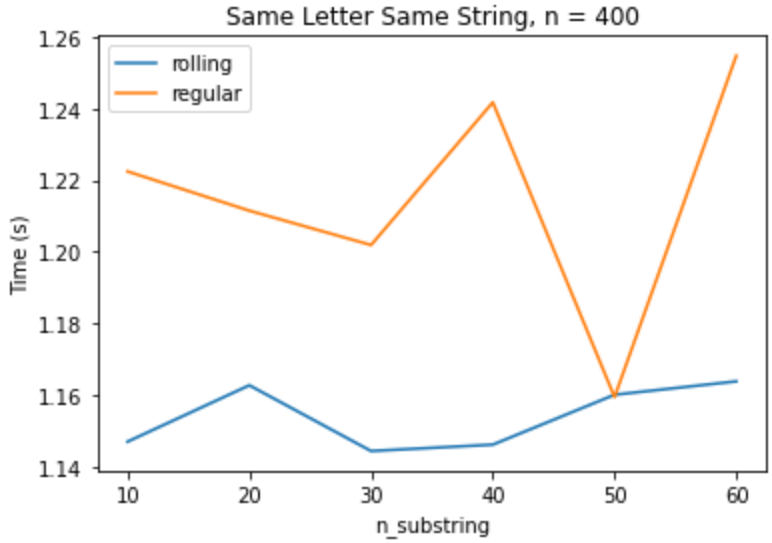

Same Letter but Different Strings, k = 40
rolling
regular
Time (s)
1.25
1.20
1.15
1.10
1.05
100
200
300
400
500
600
n_string


Same Letter but Different Strings, n = 400
rolling
regular
Time (s)
1.25
1.20
1.15
1.10
1.05
10
20
30
40
50
60
n_substring


Random Letter but Same Strings, k = 40
rolling
regular
Time (s)
1.30
1.25
1.20
1.15
1.10
1.05
1.00
100
200
300
400
500
600
n_string


Random Letter but Same Strings, n = 400
rolling
regular
Time (s)
1.30
1.25
1.20
1.15
1.10
10
20
30
40
50
60
n_substring


Same Letter Same String, k = 40
rolling
regular
Time (s)
1.4
1.3
1.2
1.1
1.0
100
200
300
400
500
600
n_string


Same Letter Same String, n = 400
rolling
regular
Time (s)
1.26
1.24
1.22
1.20
1.18
1.16
1.14
10
20
30
40
50
60
n_substring

I: Computing the time complexity for different Input:

```
## computing time of both function for experimental complexity

# to store the average time
rolling = [[] for i in range(5)]
regular = [[] for i in range(5)]



iteration = 10
n_list = np.array([100,200,300,400,500,600])
k_list = np.array([10,20,30,40,50,60])

for i in n_list:

    for k in k_list:

        # to store the total time
        rolling_hash_time = [0 for i in range(5)]
        regular_hash_time = [0 for i in range(5)]

        # 1 - strings from shakespear
        for _ in range(iteration):
            x = full_text[:i]
            y = full_text[-1*i:]

            # rolling
            start = time.time()
            rh_hash = rh_get_match(x,y,k)
            end = time.time()
            rolling_hash_time[0] += end - start

            # regular
            start = time.time()
            regular_hash = regular_get_match(x,y,k)
            end = time.time()
```

```
        regular_hash_time[0] += end - start

    rolling[0].append(rolling_hash_time[0]/iteration)
    regular[0].append(regular_hash_time[0]/iteration)


    # 2 - random different string
    for _ in range(iteration):
        x = randomword(i)
        y = randomword(i)

        # rolling
        start = time.time()
        rh_hash = rh_get_match(x,y,k)
        end = time.time()
        rolling_hash_time[1] += end - start

        # regular
        start = time.time()
        regular_hash = regular_get_match(x,y,k)
        end = time.time()
        regular_hash_time[1] += end - start

    rolling[1].append(rolling_hash_time[1]/iteration)
    regular[1].append(regular_hash_time[1]/iteration)


    # 3 - same letters but different string
    for _ in range(iteration):
        x = sameletter(i)
        y = sameletter(i)

        # rolling
        start = time.time()
        rh_hash = rh_get_match(x,y,k)
        end = time.time()
        rolling_hash_time[2] += end - start
```

```
        # regular
        start = time.time()
        regular_hash = regular_get_match(x,y,k)
        end = time.time()
        regular_hash_time[2] += end - start

    rolling[2].append(rolling_hash_time[2]/iteration)
    regular[2].append(regular_hash_time[2]/iteration)


    # 4 - random letters but same string
    for _ in range(iteration):
        x = randomword(i)
        y = x

        # rolling
        start = time.time()
        rh_hash = rh_get_match(x,y,k)
        end = time.time()
        rolling_hash_time[3] += end - start

        # regular
        start = time.time()
        regular_hash = regular_get_match(x,y,k)
        end = time.time()
        regular_hash_time[3] += end - start


    rolling[3].append(rolling_hash_time[3]/iteration)
    regular[3].append(regular_hash_time[3]/iteration)


    # 5 - same letters and same string
    for _ in range(iteration):
        x = sameletter(i)
        y = x
```

```
        # rolling
        start = time.time()
        rh_hash = rh_get_match(x,y,k)
        end = time.time()
        rolling_hash_time[4] += end - start

        # regular
        start = time.time()
        regular_hash = regular_get_match(x,y,k)
        end = time.time()
        regular_hash_time[4] += end - start

    rolling[4].append(rolling_hash_time[4]/iteration)
    regular[4].append(regular_hash_time[4]/iteration)

    print(i,k,'done')
```