DATA STRUCTURES AND ALGORITHMS

Year: Spring 2021

FINAL PROJECT REPORT

IMPLEMENTATION AND ANALYSIS OF STRING MATCHING ALGORITHMS FOR SEARCHING KEYWORDS IN PDF DOCUMENTS

Nitya Sathyavageeswaran (RUID:191007240)

Sudarshan Srinivasan (RUID: 182003539)

Github Link: https://github.com/nityas1997/DSA_21_group

Table of Contents

1 Abstract

String searching algorithms find all the occurrences of a particular pattern in a text file. Our aim is to implement the naive, KMP, Boyer Moore and Rabin Karp string matching algorithms and compare them based on the time taken and the number of comparisons needed. We find that Boyer Moore is the only algorithm with sublinear average case time complexity for the searching. The KMP algorithm turns out to be the fastest and has the least number of text comparisons needed. The report also has a table proving the space requirements for each of the four algorithms.

2 Introduction / Purpose

There is a large amount of data available in today's world and there is a need for algorithms to extract useful information from this data. The string searching algorithms is a class of algorithms that is useful in this regard. The algorithm searches for a particular pattern in the given data.

String matching algorithms are broadly classified into the following two types:

1. Exact String Matching Algorithms
2. Approximate String Matching Algorithms

In our project we are going to be focussing on exact string matching algorithms. The motivation of our project is to use these algorithms to search for a particular text match in pdf files, notepad/word documents and return the number of matches and the exact position of the match in the file.

Few other applications of string matching algorithms include plagiarism detection, detection of patterns in DNA sequencing and spam filtering [1].

The goal of an exact string matching algorithm is to find all the occurences of a particular pattern of length M in a text of length N. All the elements in the patter and text are taken from a finite alphabet set of size R.

The exact string matching algorithms are of two types:

1. Algorithms based on character comparison
    - Naive algorithm
    - Knuth Morris Pratt (KMP) Algorithm
    - Boyer Moore Algorithm
2. Algorithm based on hashing string matching
    - Rabin Karp Algorithm

We will be discussing the above mentioned algorithms in detail in the next section.

3 Algorithms

3.1) Naive Algorithm

This is a brute force implementation and is the most basic implementation. We use a sliding window mechanism in this algorithm. We just slide the pattern over the text one by one till a match is found. Once a match is found, we slide the pattern further to check for subsequent matches [2].

Pseudocode[3]:

```
Begin
   patLen := pattern Size
   strLen := string size

   for i := 0 to (strLen - patLen), do
      for j := 0 to patLen, do
         if text[i+j] ≠ pattern[j], then
            break the loop
      done

      if j == patLen, then
         display the position i, as there pattern found
   done
End
```

Figure 1: Pseudo code of Naive algorithm

Properties:

- No preprocessing of data needed
- Algorithm is slow when the pattern and data are of large lengths
- It is also slow when there are a lot of common elements between the pattern and text file.
- Average case time complexity : O(N) where N is the length of the text file.
- Worst case time complexity: O(MN) where N is the length of the text file and M is the length of the pattern.

When does the best case occur in the naive algorithm?

The best case occurs when the first letter of the pattern is not found in the text at all. An example for this could be when the text is say “Apple is red " and the pattern we want to search for is “fax”. The number of comparisons in this case is O(N).

When does the worst case occur in the naive algorithm?

This can happen in two ways:

1. When all the characters in the pattern and the text are the same like in the example given below:
   Text:”EEEEEEE”
   Patten:”EEE”
2. When the characters in the text differ from the characters in the pattern only in the last element. An example of this is given below:
   Text: “DDDDDDDD”
   Pattern: “DDDB”

The number of comparisons in this case is O(M*(N-M+1)). This is because from the pseudo code we see that in the worst case the outer ‘for’ loop runs “N-M+1” times and the inner ‘for’ loop runs ‘M’ times.

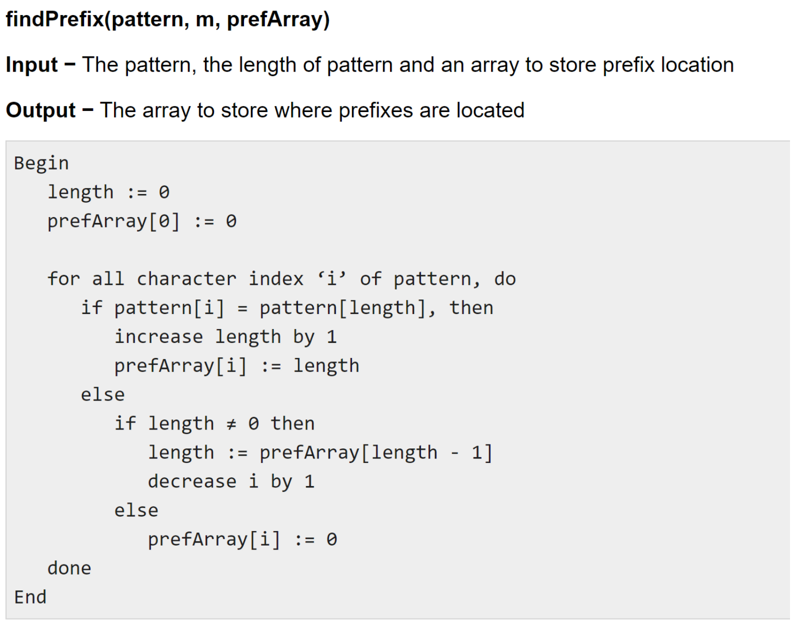
Drawback:

When there is a mismatch between the pattern and text, matching between the pattern and text starts by going back in the text.

3.2) KMP (Knuth Morris Pratt Algorithm)

We already mentioned that the Naive algorithm does not work well on cases where we see many matching characters followed by a mismatching character. [4] Hence, the KMP algorithm's basic idea is that whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. Thus, it bypasses re-examination of previously matched characters. [5]

Pseudocode [6]:



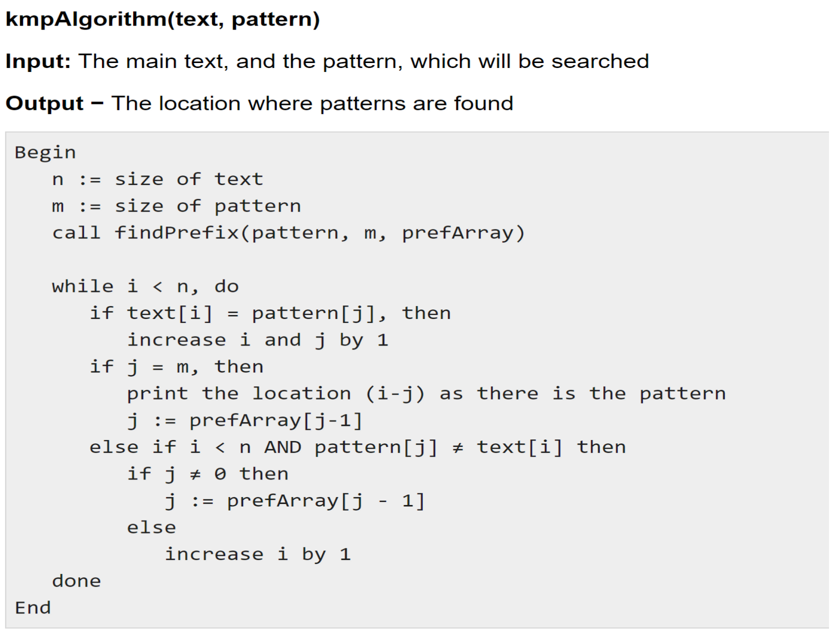
findPrefix(pattern, m, prefArray)

Input − The pattern, the length of pattern and an array to store prefix location

Output − The array to store where prefixes are located

```
Begin
   length := 0
   prefArray[0] := 0

   for all character index ‘i’ of pattern, do
      if pattern[i] = pattern[length], then
         increase length by 1
         prefArray[i] := length
      else
         if length ≠ 0 then
            length := prefArray[length - 1]
            decrease i by 1
         else
            prefArray[i] := 0
   done
End
```



kmpAlgorithm(text, pattern)

Input: The main text, and the pattern, which will be searched

Output − The location where patterns are found

```
Begin
   n := size of text
   m := size of pattern
   call findPrefix(pattern, m, prefArray)

   while i < n, do
      if text[i] = pattern[j], then
         increase i and j by 1
      if j = m, then
         print the location (i-j) as there is the pattern
         j := prefArray[j-1]
      else if i < n AND pattern[j] ≠ text[i] then
         if j ≠ 0 then
            j := prefArray[j - 1]
         else
            increase i by 1
   done
End
```

Figure 2: Pseudo code of KMP algorithm

Properties:

- Preprocessing of data performed
- Checks characters from left to right
- Uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) which helps in improving the time complexity compared to Naive algorithm
- Average case time complexity : O(MN) where N is the length of the text file.

Drawback:

This algorithm doesn’t work so well as the size of the alphabets increases due to which more chances of mismatch occurs.

3.3) Boyer Moore Algorithm

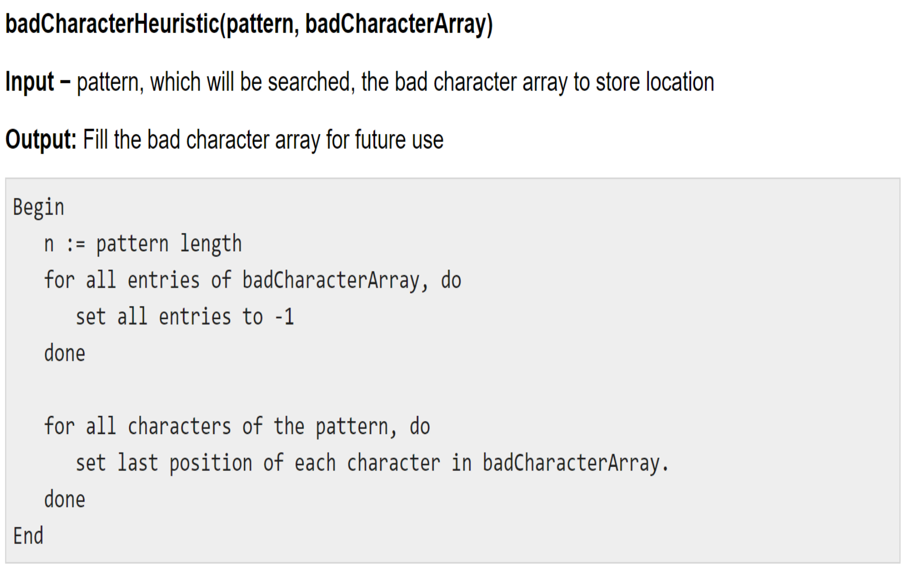
The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. The reason is that it works the fastest when the alphabet is moderately sized and the pattern is relatively long [7].

The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. This algorithm is a combination of two approaches:

1. The ***Good- Suffix Heuristic*** (also called matching shift) approach
2. The ***Bad-Character Heuristic*** (also called the occurrence shift) approach

For our project, we have implemented the bad character heuristic method. In this method we will try to find a **bad** character, that means a character of the main string, which is not matching with the pattern. When the mismatch has occurred, we will shift the entire pattern until the mismatch becomes a match, otherwise, the pattern moves past the bad character [8].

Pseudocode [8]:



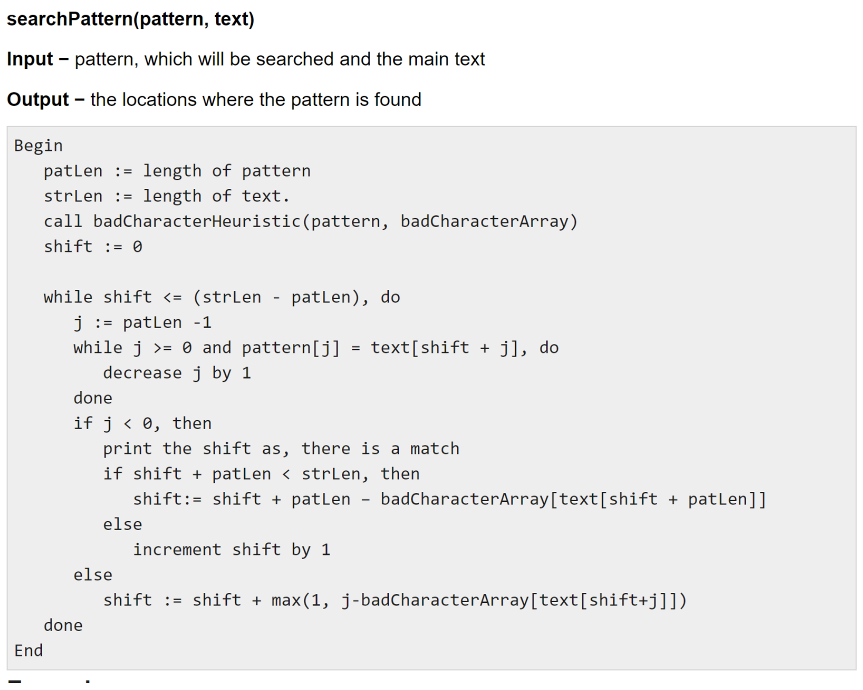
badCharacterHeuristic(pattern, badCharacterArray)

Input – pattern, which will be searched, the bad character array to store location

Output: Fill the bad character array for future use

```
Begin
   n := pattern length
   for all entries of badCharacterArray, do
      set all entries to -1
   done

   for all characters of the pattern, do
      set last position of each character in badCharacterArray.
   done
End
```



searchPattern(pattern, text)

Input – pattern, which will be searched and the main text

Output – the locations where the pattern is found

```
Begin
   patLen := length of pattern
   strLen := length of text.
   call badCharacterHeuristic(pattern, badCharacterArray)
   shift := 0

   while shift <= (strLen - patLen), do
      j := patLen -1
      while j >= 0 and pattern[j] = text[shift + j], do
         decrease j by 1
      done
      if j < 0, then
         print the shift as, there is a match
         if shift + patLen < strLen, then
            shift:= shift + patLen – badCharacterArray[text[shift + patLen]]
         else
            increment shift by 1
      else
         shift := shift + max(1, j-badCharacterArray[text[shift+j]])
   done
End
```

Figure 3: Pseudo code of Boyer Moore algorithm [8]

Properties:

- Preprocessing of data performed
- Performs string matching by starting at the right end of the string. The algorithm uses the best properties from both the KMP and Naive algorithm and therefore should be more efficient
- Best Case Time complexity - O(N/M) = when all characters of text and pattern are same
- Worst case Time Complexity - O(MN) = when all characters of text and pattern are different [8]

Drawback:

The main drawback of the Boyer-Moore algorithm is the preprocessing time and the space required, which depends on the alphabet size and/or the pattern size. For this reason, if the pattern is small (1 to 3 characters long) it is better to use the naive algorithm. If the alphabet size is large, then the Knuth-Morris-Pratt algorithm is a good choice. In all the other cases, in particular for long texts, the Boyer-Moore algorithm is better [9].

3.4) Rabin Karp Algorithm

The Rabin Karp Algorithm is also similar to the naive algorithm in the sense that, even here the pattern is slid over the text one by one. But in this case we first match the hash value of the pattern with the hash value of a substring of the text. Once the hash values of the pattern and substring of the text match, individual character by character comparison is then carried out[10].

Properties:

- The pattern is pre-processed here. Preprocessing done in order to find the hash value of the text and the pattern as described before.
- Rolling hash concept is used which is described with the help of an example below.
- Average case time complexity: O(M+N)
- Worst case time complexity: O(MN). This is just similar to the naive algorithm version that we have looked at earlier.

Let us look at the implementation of this algorithm in order to better understand it.

Consider the following example:

txt[]=”ABDCB”

pat[]=”DC”

Let q be equal to 11(some small prime number so that the computation is easier). Here M=2 and N=5.

From the code we see that, h=(1*256)%11=3

We need to calculate the hash value of the pattern and the first window of the text.

The ‘for’ loop calculation is as follows:

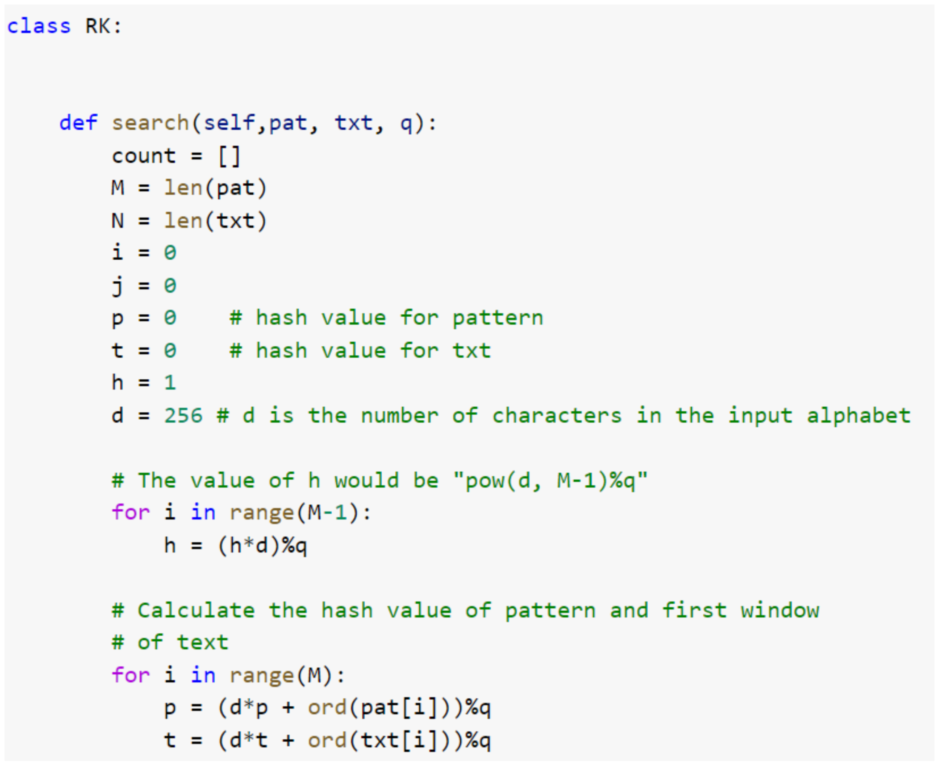
Iteration 1: p=(256*0+68)%11=2, t=(256*0+65)%11=10

Iteration 2: p=(256*2+67)%11=7, t=(256*10+66)%11=8

Two iterations because M=2.

In the above calculations 68,65,67,66 are the ASCII values of D, A ,C and B respectively.

We notice that the hash value of “DC” and “AB” do not match. We therefore slide the window by one and find the hash value of “BD”.



```
class RK:


    def search(self,pat, txt, q):
        count = []
        M = len(pat)
        N = len(txt)
        i = 0
        j = 0
        p = 0    # hash value for pattern
        t = 0    # hash value for txt
        h = 1
        d = 256 # d is the number of characters in the input alphabet

        # The value of h would be "pow(d, M-1)%q"
        for i in range(M-1):
            h = (h*d)%q

        # Calculate the hash value of pattern and first window
        # of text
        for i in range(M):
            p = (d*p + ord(pat[i]))%q
            t = (d*t + ord(txt[i]))%q
```

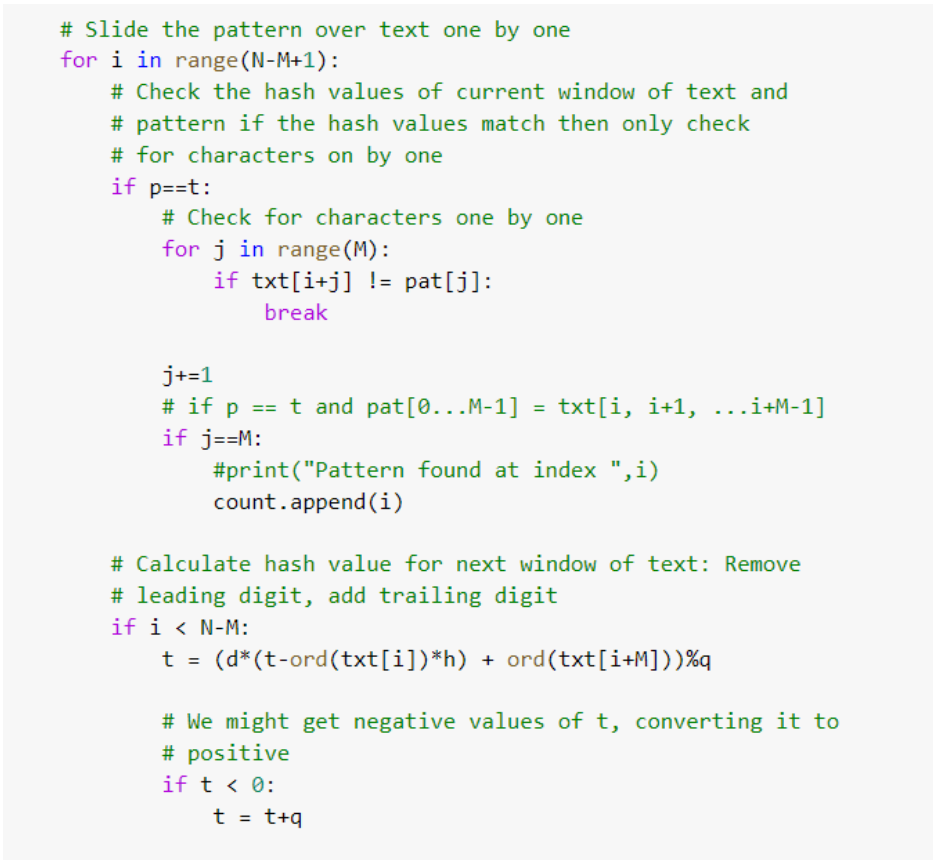
Figure 4a: Implementation of Rabin Karp Algorithm

We now have,
p=7, t=8,q=11,h=3

t=(256*(8-65*3)+68)%11=-9=-9+11=2

Continuing the calculation after sliding the window once more, we get:

t=(256*(2-66*3)+67)%11=-4=-4+11=7



```
# Slide the pattern over text one by one
for i in range(N-M+1):
    # Check the hash values of current window of text and
    # pattern if the hash values match then only check
    # for characters on by one
    if p==t:
        # Check for characters one by one
        for j in range(M):
            if txt[i+j] != pat[j]:
                break

        j+=1
        # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
        if j==M:
            #print("Pattern found at index ",i)
            count.append(i)

    # Calculate hash value for next window of text: Remove
    # leading digit, add trailing digit
    if i < N-M:
        t = (d*(t-ord(txt[i])*h) + ord(txt[i+M]))%q

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

Figure 4b: Implementation of Rabin Karp Algorithm(contd.)

The hash value of the text and the pattern finally match(equal to 7). We thus now do character by character comparison in order to find the match.

4 Experimental Setup

We consider five text files as our input data. The data sets include a collection of movie scripts, twitter lists,etc.

The data set can be found here: https://github.com/Phylliida/Dialogue-Datasets.

We run all the four algorithms on each of the data sets to compare their performance. We compare the time taken and the number of comparisons carried out by each of the algorithms.

5 Results and Analysis

Our implementation finds all the possible matches of a particular word in the text file and also gives the position where the match was found.

The time taken(in s) by the algorithms on each of the datasets is given below:

Datasets	**Naive**	**KMP**	**Rabin Karp**
BNCCorpus.txt	9.574026823	5.693263531	9.398285151
BNCSplitWordsCorpus.txt	9.736764908	5.703483582	9.456381559
MovieCorpus.txt	8.571478367	5.156430006	8.354653835
TwitterConvCorpus.txt	0.3179531097	0.1899700165	0.3161551952
TwitterLowerAsciiCorpus.txt	0.3104944229	0.1758773327	0.3099918365

Table 1a: Time Analysis

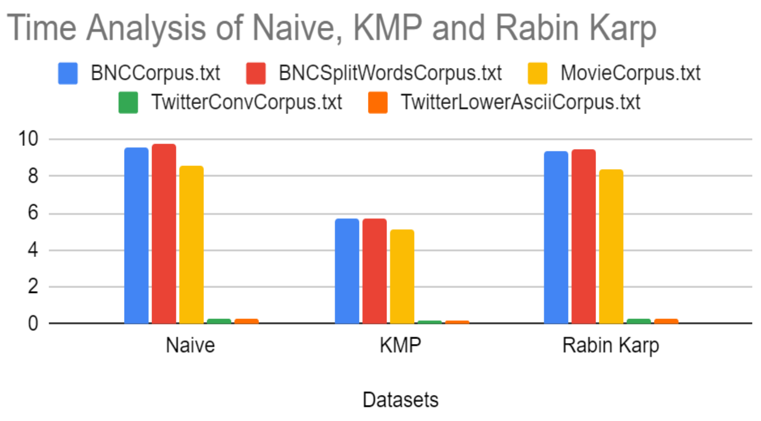


Figure 5a: Time Analysis

The number of comparisons is given below:

Datasets	Naive	KMP	Rabin Karp
BNCCorpus.txt	1117	1117	1511
BNCSplitWordsCorpus.txt	1052	1052	1489
MovieCorpus.txt	1468	1468	1470
TwitterConvCorpus.txt	61	61	62
TwitterLowerAsciiCorpus.txt	63	63	64

Table 1b: Number of comparisons

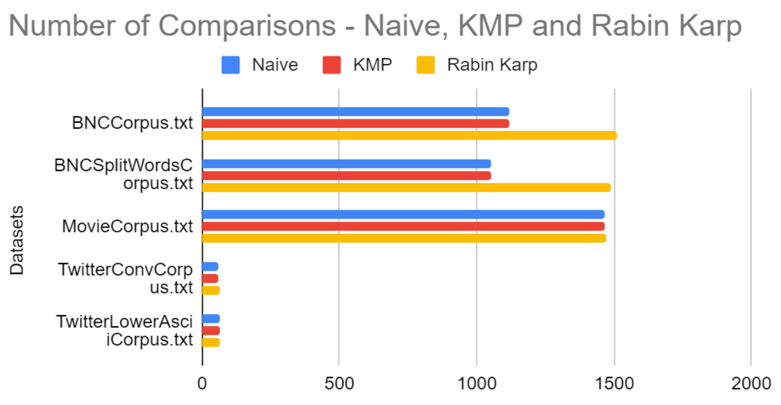


Figure 5b: Number of comparisons

Overall time complexity analysis:

Algorithm	**Average case searching time complexity**	**Worst case searching time complexity**	**Extra space requirement**
Naive	O(M)	O(MN)	O(1)
KMP	O(N)	O(N)	O(RM)
Boyer Moore	O(N/M)	O(MN)	O(R)
Rabin Karp	O(M+N)	O(MN)	O(1)

Table 1c: Time complexity

Here, M is the length of the pattern, N is the length of the text and R is the size of the alphabet(R).

6 Conclusions

From the experiments conducted on the five different data sets we conclude that the KMP algorithm is definitely faster than the naive and Rabin Karp algorithms although, the number of comparisons between the naive and KMP algorithm is comparable.
Rabin Karp algorithm takes the most time and this is because the number of comparisons is also maximum in this case.

The Boyer Moore algorithm has sublinear average case time complexity for the search time which is the best among the four algorithms. The implementation for the Boyer Moore algorithm also has low complexity.

Also in the Rabin Karp Algorithm we select a large prime number of size MN^2 to decrease the probability of false collisions by 1/N.

The KMP algorithm provides the best performance even when the text and pattern have a lot of common elements.

7 Acknowledgements

First of all, we would like to thank Professor Shantenu Jha and Ghadir Ayache for providing us this opportunity to successfully accomplish our project. We would also like to thank the grader Vikhyat Dhamija for his guidance and comments which made this work fruitful.

8 References

[1] https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/

[2] https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/

[3]https://www.tutorialspoint.com/Naive-Pattern-Searching

[4] https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

[5] https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

[6] https://www.tutorialspoint.com/Knuth-Morris-Pratt-Algorithm

[7] https://www-igm.univ-mlv.fr/~lecroq/string/node14.html

[8] https://www.tutorialspoint.com/Bad-Character-Heuristic

[9] http://orion.lcg.ufrj.br/Dr.Dobbs/books/book5/chap10.htm

[10] https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

[11] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," SIAM J. Computing 6, 323-350 (1977).

[12] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," Commun. ACM20, 762-772 (1977).

9 Individual Contributions

Nitya Sathyavageeswaran:

- Implementation(coding): Naive and Rabin Karp algorithm
- Dataset searching
- Time analysis
- Slides: Naive, Rabin Karp algorithm
- Report writing: Introduction, Naive, Rabin Karp, Experimental setup, Results and Analysis, Conclusion

Sudarshan Srinivasan:

- Implementation(coding): KMP and Boyer Moore
- Number of comparisons analysis
- Slides: Introduction, Boyer Moore, KMP, Results and Analysis
- Report writing : Cover page, Index, Abstract, KMP, Boyer Moore