[CS2223 Final Project Report]

Submitted By

[Tae Hyun Je]

[Shundong Li]

[Yongcheng Liu]

[Xiao Xiao]

[Yicheng Yang]

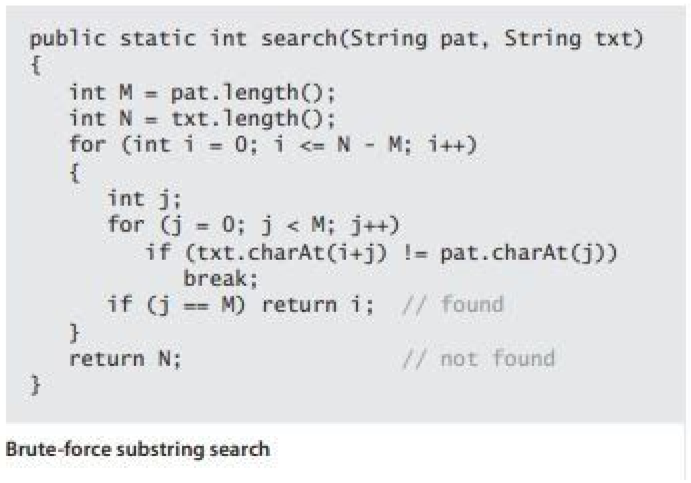
Date Submitted: [5.12.2020]

Table of Contents

Introduction

What is substring search? Substring search can be defined as attempting to find a desired pattern, within a String. For example, the String “TOMATO” and a substring search algorithm tries to find “MA”; the algorithm would return the index 2, since that is when the desired string starts. While the task of finding a substring may seem easy, finding the correct approach is difficult. A String could be short, 10 characters, while another could be over a million characters long. It becomes apparent that finding an efficient algorithm is key.

The general approach that many people use is the brute force algorithm. The basic premise of this algorithm is to check each element in some data structure (see Figure 1). If you check each subsequent character that matched the first character of the substring, the worst case time complexity would be equivalent to O(N*M). However, say, you have a String that is over 1 million characters long. Obviously, it will work well enough for short Strings but longer ones will take a significant amount of time.



```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
   for (int i = 0; i <= N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt.charAt(i+j) != pat.charAt(j))
            break;
      if (j == M) return i;  // found
   }
   return N;                 // not found
}
```

Brute-force substring search

Figure 1. Brute Force Algorithm for Substring Search

The second algorithm, which is significantly more efficient in some circumstances, is the KMP algorithm. This algorithm was devised by Knuth, Morris, and Pratt, who published their technique and findings in 1977 called *Fast Pattern Matching in Strings*.[1] However, they were not the only ones to publish a substring algorithm. Around the same time, R.S. Boyer and J. S. Moore discovered a different algorithm that was more efficient in some cases because it only probed a small sector of the String. However, both algorithms had one thing in common that many people noticed, they were difficult to understand.[2] So, while the many variations of substring algorithms might be efficient, the implementation of them can be difficult to understand. The published KMP is built upon an interesting finding that "whenever we detect a mismatch, we already know some of the characters in the text (being searched string)".[3] By utilizing this knowledge of known characters, it has a worst case time complexity of O(N+M). The KMP algorithm can be broken down into two steps: building a helper array based on pattern and then running a search for the string with the built array.

There are many other substring algorithms that have been created; however, this report will focus only on two particular subjects: understanding how KMP is implemented and the performances of it and the brute force algorithm.

[1] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing* 6, no. 2 (1977): pp. 323-350, https://doi.org/10.1137/0206024.

[2] Sedgewick, Robert, and Kevin Wayne. *Algorithms, Fourth Edition*. Addison-Wesley Professional, 2011., 759

[3] Sedgewick, Robert, and Kevin Wayne. *Algorithms, Fourth Edition*. 762

KMP implementation

In order to offer better consistency with the actual implementation in code, “Text” will be used to notate the “being searched string” for the rest of the report. Meanwhile, “Pattern” will be used to notate the “goal string”, or “seeking string”, for the rest of the report. For example, “searching substring of ‘abcde’ in the context string of ‘abcde fghij’ ” can be translated as “searching Pattern ‘abcde’ in Text ‘abcde fghij’ ”. The behavior of finding substrings can thus be translated as “finding Patterns in Text”.

The essence of KMP implementation is to pre-build a helper array to store backup points in case a mismatch between the Pattern and searched string happens. Based on this idea, there are many variations in the implementation of KMP. In order to better understand the KMP algorithm, we have chosen two representative ones to run experiments. The first one is the KMP implementation in the algorithm textbook *Algorithms*, which can be supposed to be the original implementation from Knuth, Morris and Pratt.[4] The second one is a simplified implementation found online, which has some small differences in Pattern processing and searching to the first one.[5] For convenience, for the rest of the report “KMP1” will be used to denote the implementation from *Algorithms*, while “KMP2” will be used to denote the implementation from the internet.

As mentioned in introduction, KMP algorithm has two general steps:

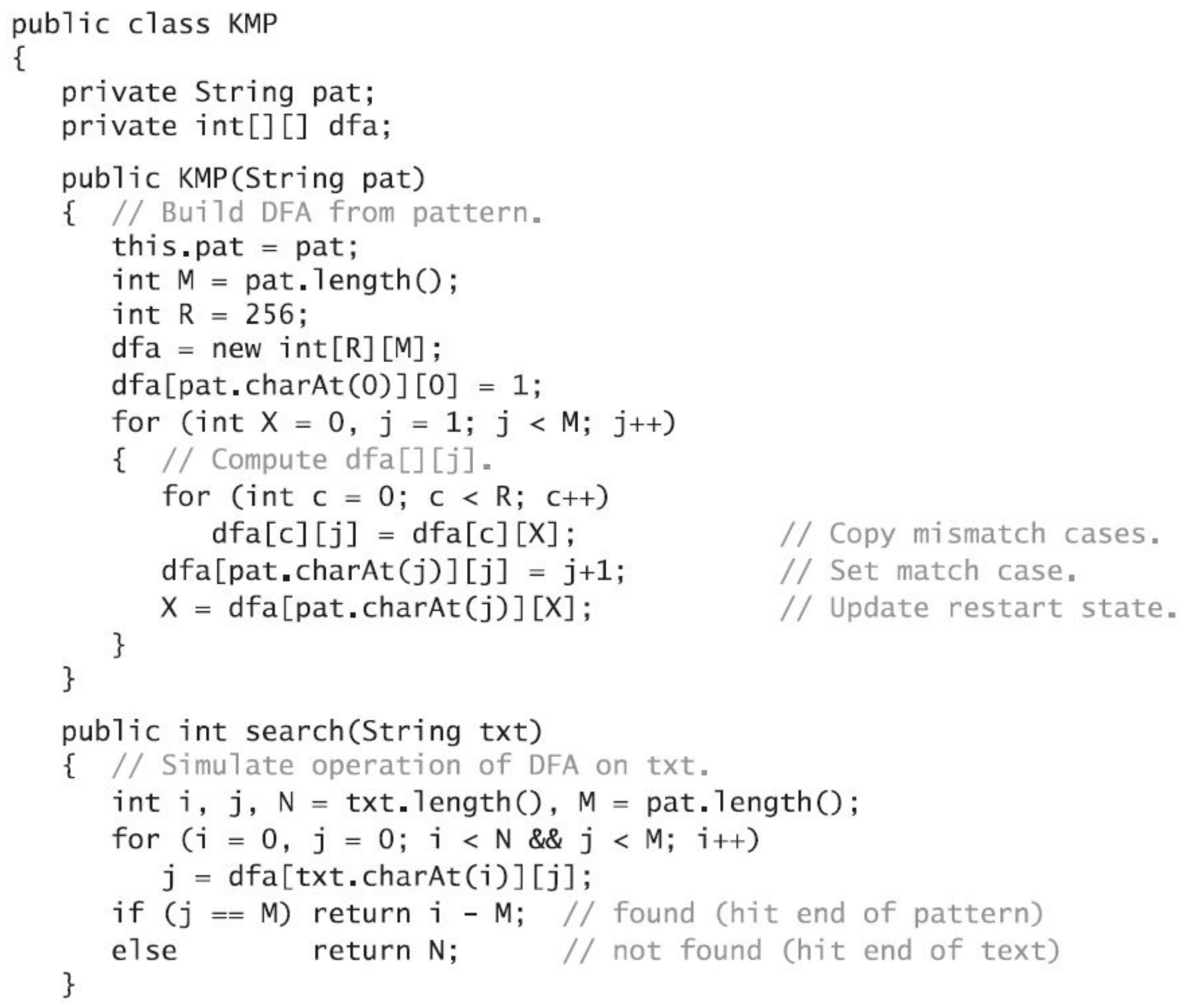
1. Build a helper array based on the Pattern
2. Run the search for the string with the built array

Thus, detailed analysis of the KMP implementations will mainly focus on how each implementation approaches each step, as well as some potential benefits and drawbacks of such implementation. In terms of code, the first step is being coded as the constructor for a KMP class, and the second step is being coded as an instance method for a KMP class. Consequently, in this section, the report is going to analyze the “constructor” and “search” method of each KMP implementation.

[4] Sedgewick, Robert, and Kevin Wayne. *Algorithms, Fourth Edition*. Addison-Wesley Professional, 2011., 768
[5] Mission-Peace, “interview,” *Github*, accessed May 3, 2020. https://github.com/mission-peace/interview/blob/master/src/com/interview/string/SubstringSearch.java

KMP1:

Full Implementation:



```
public class KMP
{
   private String pat;
   private int[][] dfa;

   public KMP(String pat)
   {  // Build DFA from pattern.
      this.pat = pat;
      int M = pat.length();
      int R = 256;
      dfa = new int[R][M];
      dfa[pat.charAt(0)][0] = 1;
      for (int X = 0, j = 1; j < M; j++)
      {  // Compute dfa[][j].
         for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X];            // Copy mismatch cases.
         dfa[pat.charAt(j)][j] = j+1;         // Set match case.
         X = dfa[pat.charAt(j)][X];           // Update restart state.
      }
   }

   public int search(String txt)
   {  // Simulate operation of DFA on txt.
      int i, j, N = txt.length(), M = pat.length();
      for (i = 0, j = 0; i < N && j < M; i++)
         j = dfa[txt.charAt(i)][j];
      if (j == M) return i - M;  // found (hit end of pattern)
      else        return N;      // not found (hit end of text)
   }
```

Figure 2. KMP Implementation From Algorithm, 4th Edition

Constructor For KMP1:

The constructor for KMP1 Implementation builds a DFA from a Pattern string. DFA is a finite state machine for processing strings. It gives KMP1 the ability to decide which state it should go to for a certain character. In this implementation, the functionality of the DFA is to help each character at each Pattern position to decide what should be the closest backup point. To ensure that every possibility is included, the implementation builds a 2D Array of size [256][Pattern.length]. 256 is the number of potential characters in AscII set, while Pattern.length is the smallest size for ensuring that each Pattern index position has a state. For example, dfa['a'][1] (which is dfa[97][1]) represents the next state for getting an 'a' at Pattern index 1 when comparing the Pattern to Text at some position.

Specifically, the implementation uses two for loops to construct the actual dfa. The outer for loop is for traversing all the Pattern indexes to ensure there are 256 states for each Pattern index (all possibilities for AscII set). The inner for loop is for traversing all potential characters at a fixed Pattern index to build the actual state for a specific character. It completes the task by copying the states of the same character from the previous Pattern index, and sets the state for current Pattern character to j+1(j is the previous state value) , which is the state for "found" at the Pattern position. Lastly, it resets restart state value for ensuring correct states will be copied in the next loop.

Overall, the time complexity of construction is f (n) = (Pattern.length * 256), which can be viewed as O (n) for n >>256, or nelectable for Text.length >> 256. The space complexity is f (n) = (Pattern.length * 256), which can be viewed as O (n).

Search For KMP1:

The search implementation for KMP1 is very straightforward. It traverses through the Text and updates the state value in real time. When the state value accumulated to Pattern.length, then it will break the loop, and tell the program that Pattern exists in Text. If it never achieves Pattern.length, that means Pattern does not exist in Text.

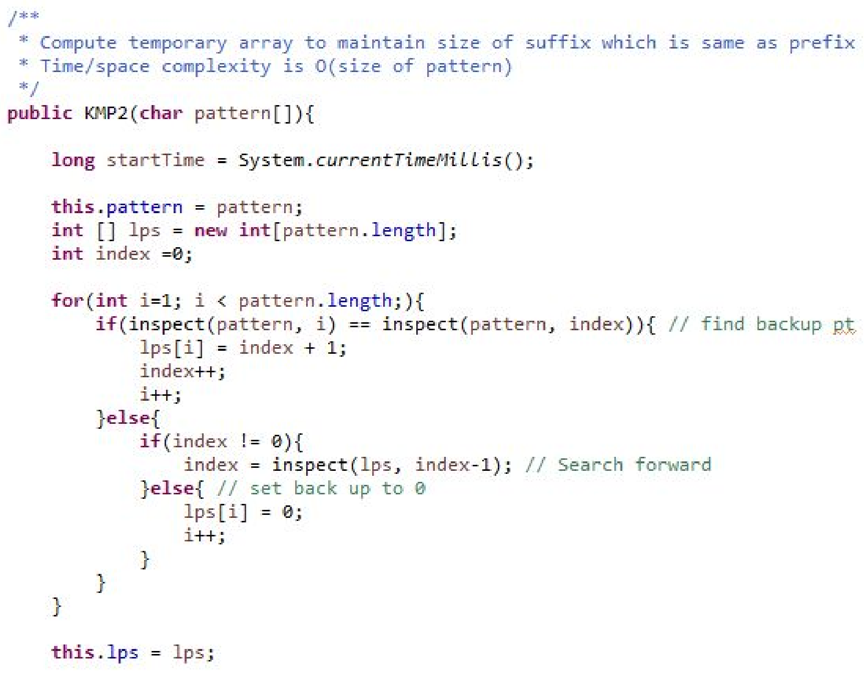
Time Complexity : f(n) = Text.length, which can be view as O(n)

Potential Benefit / Drawbacks for KMP1:

The implementation of KMP1 guarantees worst case Text.length times of traversal for searching, which is crazy. However, it assumes that the Pattern is way shorter than the Text, so the cost of constructing the helper array would be neglectable. However, in most cases, that might not be the case. So for longer Patterns, KMP1 is not a good search algorithm.

KMP2:

Full Implementation:



```
/**
 * Compute temporary array to maintain size of suffix which is same as prefix
 * Time/space complexity is O(size of pattern)
 */
public KMP2(char pattern[]){

    long startTime = System.currentTimeMillis();

    this.pattern = pattern;
    int [] lps = new int[pattern.length];
    int index =0;

    for(int i=1; i < pattern.length;){
        if(inspect(pattern, i) == inspect(pattern, index)){ // find backup pt
            lps[i] = index + 1;
            index++;
            i++;
        }else{
            if(index != 0){
                index = inspect(lps, index-1); // Search forward
            }else{ // set back up to 0
                lps[i] = 0;
                i++;
            }
        }
    }

    this.lps = lps;
```

Figure 3. KMP2 Constructor Implementation From Tushar Roy

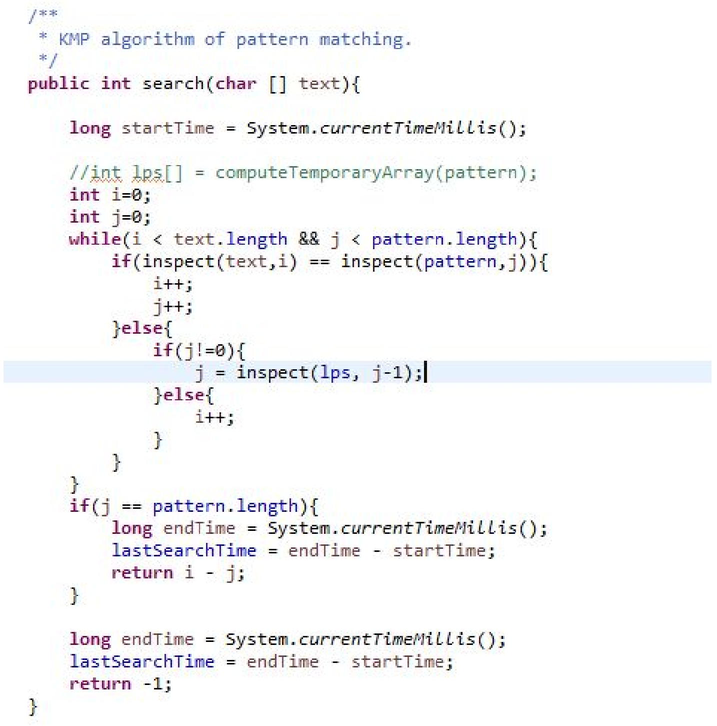
Constructor For KMP2:

The constructor for KMP2 Implementation builds a 1D Array from a Pattern string. The goal is to enable search to get stored backup indexes where the last potential restart index could be based on the repetition patterns within Pattern in the Array. Suppose we are trying to find Pattern “ABCDEFG” in Text. If a mismatch is received when the pattern index is at G, it is implied that the previous character must be “F” (or it will not go further to G). However, as there are no repeated characters in the Pattern, there is no need to start from any of the Pattern Index again. We can directly restart our search after G. Another example could be a Pattern with inner repetitions. Suppose we are trying to find Pattern “ABCABCG” in Text. If a mismatch is received when the program is trying to compare the final G in the Pattern to some character in the Text, there is a possibility that the compared character from the Text is A, which will make the fourth A in the pattern a potential start. In order to achieve this, KMP2 implementation used some tricky way to indicate repetitions in the pattern. Take Pattern “ABCABCG” as example again, the 1D helper array generated will then be {0, 0, 0, 1, 2, 3, 0}. What this means is basically that if a mismatch is found at Pattern index of 7, the program should go check if there is a match between the Text character and Pattern[array[7-1]] (A, which is a potential start within the Pattern).

Specifically, the implementation uses two pointers to construct the array. Pointer i is used to traverse through the whole array, while Pointer index is used to keep track of the last possible restart index. So when a match within pattern is found between Pattern[i] and Pattern[index], that means there must be an ongoing repetition starting at i. If a mismatch is found when index is not zero, which indicates an being tracked repetition, it will go check if Pattern[i] matches the Pattern[lps[index-1]], which is previous "restarting index with same offset (which is tracked by the index as well)".

Overall, the time complexity of construction is f (n) ~ O(n), and the space complexity is f (n) = Pattern.length, which can be viewed as O (n) or neglectable if Text.length >> Pattern.length.

Search For KMP2:



```
/**
 * KMP algorithm of pattern matching.
 */
public int search(char [] text){

    long startTime = System.currentTimeMillis();

    //int lps[] = computeTemporaryArray(pattern);
    int i=0;
    int j=0;
    while(i < text.length && j < pattern.length){
        if(inspect(text,i) == inspect(pattern,j)){
            i++;
            j++;
        }else{
            if(j!=0){
                j = inspect(lps, j-1);
            }else{
                i++;
            }
        }
    }
    if(j == pattern.length){
        long endTime = System.currentTimeMillis();
        lastSearchTime = endTime - startTime;
        return i - j;
    }

    long endTime = System.currentTimeMillis();
    lastSearchTime = endTime - startTime;
    return -1;
}
```

Figure 4. KMP2 Constructor Implementation From Tushar Roy

The search implementation for KMP2 is consistent with design ideas for the constructor. The main search body is the while loop at line 7. The logic is that while it is searching, compare the current Text character with Pattern character. If it is a match, both point increments. However, if it is not a match, before Pattern point goes back to zero (by reading what should be the last “repetition starting point”), compare the current Text character with all potential starting points in Pattern previously. If a match is found, then continue with the Pattern index set to the “potential restarting point”. If the Pattern index goes back to zero, which means there is no potential restarting point, then move on to the next character in the Text.

Time Complexity : f(n) ~ O(n)

Potential Benefit / Drawbacks for KMP2:

Compared to KMP1, the implementation of KMP2 constructs a 1D helper array of size Pattern.length rather than dfa of size [256 * Pattern.length]. It uses less space and time for construction. However, it will compare more at the time of searching. But overall, it also possesses O(M+N) Time Complexity, where M = Pattern.length, N = Text.length. Similarly, if the pattern is relatively longer, KMP2 would not be an ideal approach.

Worst-case Strings

KMP1

To find out the worst case scenario for the KMP1, we discussed and agreed that the following variables are some potential significant factors impacting final performance greatly:

1. Length : Either Pattern.length or Text.Length
2. Mode / Dictionary : The dictionary Text and Pattern is using.

Thus, we set up our experiment with "control variable" method, and the following is our final decision on the selection of cases for experimenting:

Cases for Text dictionary

1. Condition 1: Binary Dictionary - Text All 0
2. Condition 2: Binary Dictionary - Text Random
3. Condition 3: Condition 2 + Text Length 8192
4. Condition 4: Binary Dictionary - Text 101010...
5. Condition 5: Random Text with A to Z

Cases for Pattern dictionary

1. Condition 1: all As (or 1s)
2. Condition 2: (N-2) A + B + A / (N-2) 0 + 1 + 0
3. Condition 3: (N-1) A + B + (N-1)A / (N-1) 0 + 1 + (N-1)0
4. Condition 4: Random pattern

Extra case: Finding ABB in ABABAB...

Thus, there will be 5 x 4 + 1 = 21 total cases, and we believe that can cover all circumstances we came up with.

We first need to find some pattern about search inspect times and construct inspect times with respect to text length, pattern length, text and pattern formats. The following graph shows that:

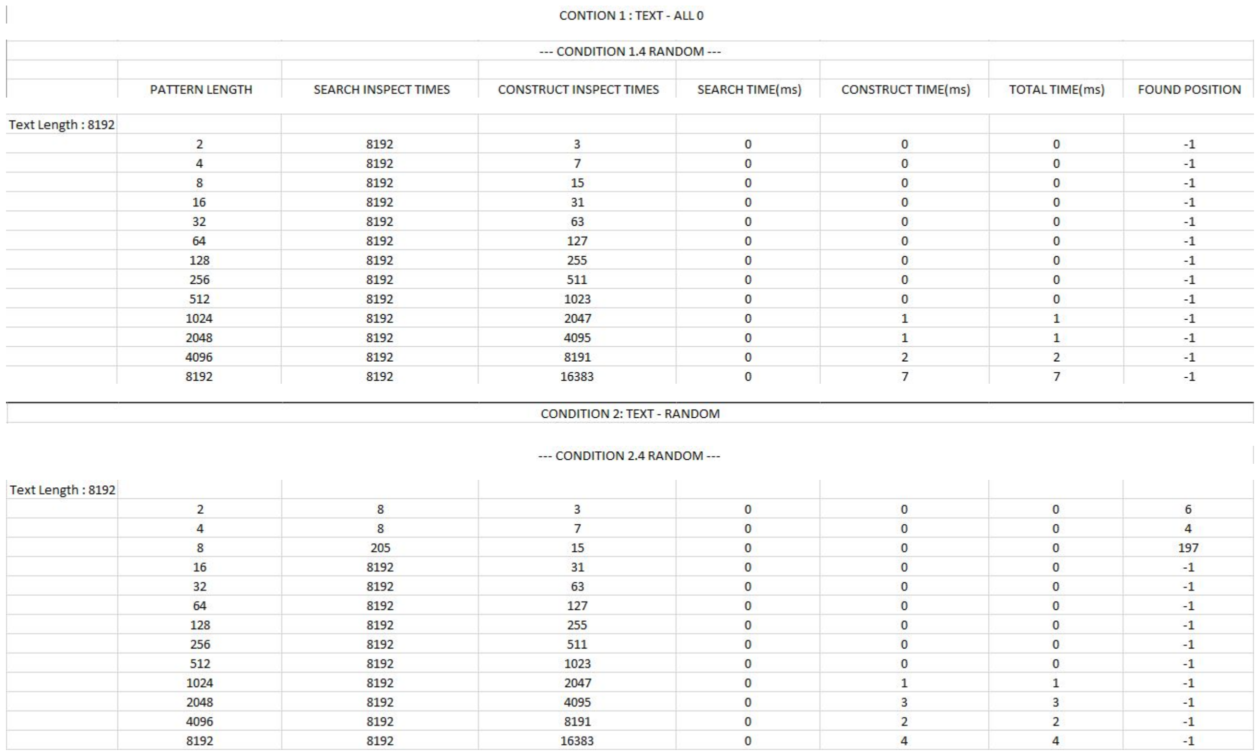
First, the KMP1 method will spend most time when it cannot find the pattern and transverse through the whole Text String compared to similar cases that found the pattern. As all search inspect times which found the pattern are lower in such cases.

Second, given different text formats and random pattern, the construct inspect time and construct inspect time is the same for the same text length and the same pattern length, we can make the hypothesis that, given the pattern is not found in the certain text, the search inspect time is equal to the text length and the construct inspect time is 2*(pattern length) - 1 , and both times are not affected by the pattern format or the text format.

Third, for actual runtime in milliseconds, the construct time has much more weight than the inspect time. From the two millisecond columns, even though the search inspect time is locked at 8192 because of the text length, the total search time is always below 1ms while construct time takes much longer due to the enormous space the dfs matrix requires. The findings above are only derived from the graphs we choose down below to give a clear perspective as well as controlling variables, but it is also valid across the board within our test results concluded in KMP1_WorstCaseExperiments.xlsx.

Thus the worst case Text String for KMP1 is any String that doesn’t contain the Pattern String. Simply put, this will happen when the Text String does not contain the Pattern String. For example, suppose a Text String of “XYAXYA...XYA” and The Pattern String of “XYZ”, you can never find this Pattern in the Text, then such Text String can be supposed to be a worst case Text String.

Another way of constructing the Text String generally is: if the pattern length is n, take the substring of the first n-1 letters and insert a letter that is not the last letter of the pattern, repeat until it reaches the desired length of the worst case string.



CONTION 1 : TEXT - ALL 0

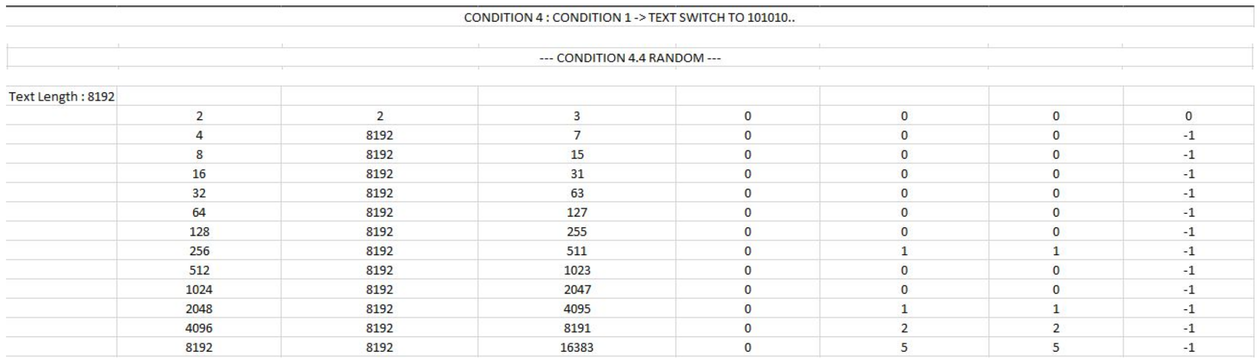
--- CONDITION 1.4 RANDOM ---

	PATTERN LENGTH	SEARCH INSPECT TIMES	CONSTRUCT INSPECT TIMES	SEARCH TIME(ms)	CONSTRUCT TIME(ms)	TOTAL TIME(ms)	FOUND POSITION
Text Length : 8192							
	2	8192	3	0	0	0	-1
	4	8192	7	0	0	0	-1
	8	8192	15	0	0	0	-1
	16	8192	31	0	0	0	-1
	32	8192	63	0	0	0	-1
	64	8192	127	0	0	0	-1
	128	8192	255	0	0	0	-1
	256	8192	511	0	0	0	-1
	512	8192	1023	0	0	0	-1
	1024	8192	2047	0	1	1	-1
	2048	8192	4095	0	1	1	-1
	4096	8192	8191	0	2	2	-1
	8192	8192	16383	0	7	7	-1

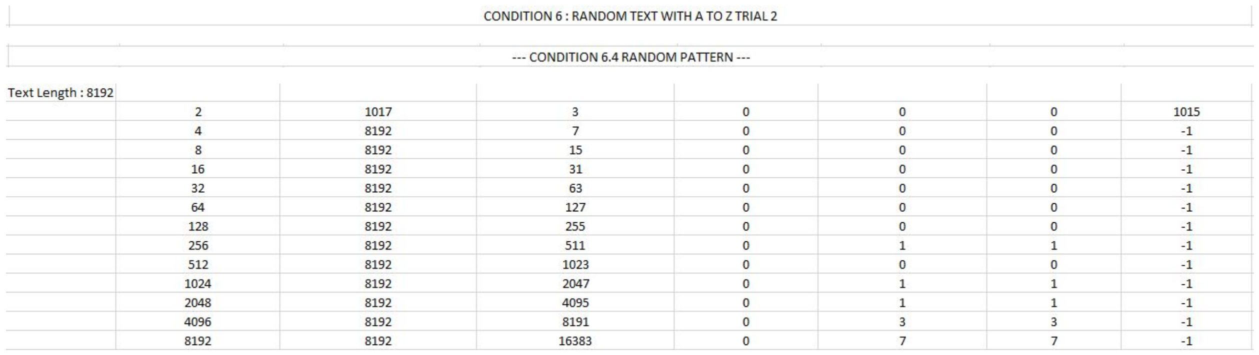
CONDITION 2: TEXT - RANDOM

--- CONDITION 2.4 RANDOM ---

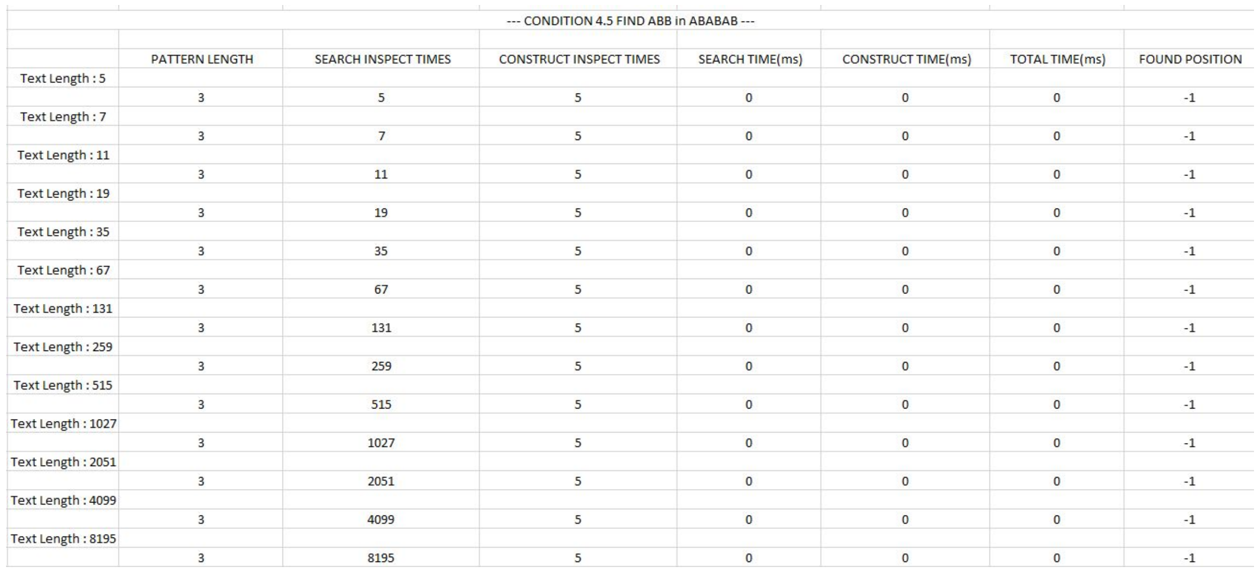
Text Length : 8192							
	2	8	3	0	0	0	6
	4	8	7	0	0	0	4
	8	205	15	0	0	0	197
	16	8192	31	0	0	0	-1
	32	8192	63	0	0	0	-1
	64	8192	127	0	0	0	-1
	128	8192	255	0	0	0	-1
	256	8192	511	0	0	0	-1
	512	8192	1023	0	0	0	-1
	1024	8192	2047	0	1	1	-1
	2048	8192	4095	0	3	3	-1
	4096	8192	8191	0	2	2	-1
	8192	8192	16383	0	4	4	-1



CONDITION 4 : CONDITION 1 -> TEXT SWITCH TO 101010..							
--- CONDITION 4.4 RANDOM ---							
Text Length : 8192							
	2	2	3	0	0	0	0
	4	8192	7	0	0	0	-1
	8	8192	15	0	0	0	-1
	16	8192	31	0	0	0	-1
	32	8192	63	0	0	0	-1
	64	8192	127	0	0	0	-1
	128	8192	255	0	0	0	-1
	256	8192	511	0	1	1	-1
	512	8192	1023	0	0	0	-1
	1024	8192	2047	0	0	0	-1
	2048	8192	4095	0	1	1	-1
	4096	8192	8191	0	2	2	-1
	8192	8192	16383	0	5	5	-1



CONDITION 6 : RANDOM TEXT WITH A TO Z TRIAL 2							
--- CONDITION 6.4 RANDOM PATTERN ---							
Text Length : 8192							
	2	1017	3	0	0	0	1015
	4	8192	7	0	0	0	-1
	8	8192	15	0	0	0	-1
	16	8192	31	0	0	0	-1
	32	8192	63	0	0	0	-1
	64	8192	127	0	0	0	-1
	128	8192	255	0	0	0	-1
	256	8192	511	0	1	1	-1
	512	8192	1023	0	0	0	-1
	1024	8192	2047	0	1	1	-1
	2048	8192	4095	0	1	1	-1
	4096	8192	8191	0	3	3	-1
	8192	8192	16383	0	7	7	-1



--- CONDITION 4.5 FIND ABB in ABABAB ---

	PATTERN LENGTH	SEARCH INSPECT TIMES	CONSTRUCT INSPECT TIMES	SEARCH TIME(ms)	CONSTRUCT TIME(ms)	TOTAL TIME(ms)	FOUND POSITION
Text Length : 5							
	3	5	5	0	0	0	-1
Text Length : 7							
	3	7	5	0	0	0	-1
Text Length : 11							
	3	11	5	0	0	0	-1
Text Length : 19							
	3	19	5	0	0	0	-1
Text Length : 35							
	3	35	5	0	0	0	-1
Text Length : 67							
	3	67	5	0	0	0	-1
Text Length : 131							
	3	131	5	0	0	0	-1
Text Length : 259							
	3	259	5	0	0	0	-1
Text Length : 515							
	3	515	5	0	0	0	-1
Text Length : 1027							
	3	1027	5	0	0	0	-1
Text Length : 2051							
	3	2051	5	0	0	0	-1
Text Length : 4099							
	3	4099	5	0	0	0	-1
Text Length : 8195							
	3	8195	5	0	0	0	-1

KMP2

For KMP2, there are also some findings that we can get from the experiments.

First, for the same pattern of the same length, the construct time for the helper matrix is always the same due to the construction method itself.

Second, the search inspection time is also the largest when it couldn't find the pattern, but this time the text pattern matters because for the same pattern and the same text length, the search inspect time is different for different text patterns due to the largely different search implementation between KMP1 and KMP2. But for the same text, same pattern length, the pattern format has a rather small impact on the total inspect times (difference within 2%).

Performance Evaluation

To evaluate the performance of the algorithms, we used two different conditions and three different alphabets for testing. The first condition uses original Text to find a pattern, so it is a guaranteed find. The second condition uses a random generator to generate both Text and Pattern, so it is highly possible that no substring could be found. The text, as well as the pattern, will belong to one of three alphabets --- DNA sequencing, lower case "a-z" or binary strings("01").

Although the instructions only require us to evaluate the performance "on random patterns and random text strings that range from 2^{14} (that is 16384) to 2^{22} (that is, 4,194,304)", we decided to expand the range to (1,024 - 4,194,304) so that we could also see the performance on searching short strings. As a result, for each evaluate trial(eg. KMP1-condition1-"a-z"), we would test 13 different text length(2^{10} , 2^{11} .... 2^{22}). And for each text length 2^k we will test k-9 pattern length(2^k , 2^{k-1} ... 2^{10}). In addition, we would run each trial three times to get the general performance.

Performance Evaluation Result

All test results were recorded and output to excel files which can be found in appendix. In order to clean up the datasets, we also wrote a python file to calculate the average processing time and inspection counts among three trails. The figures below show the performance of four algorithms in condition1 alphabet AtoZ and max text length(4194304).

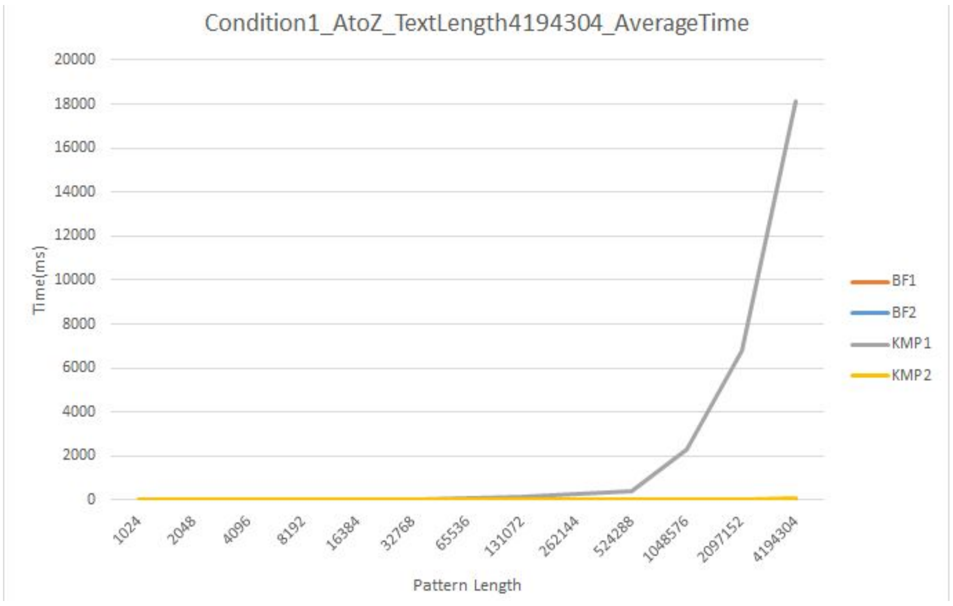


Figure 5. Average Time for Condition1 AtoZ Text Length 4194304

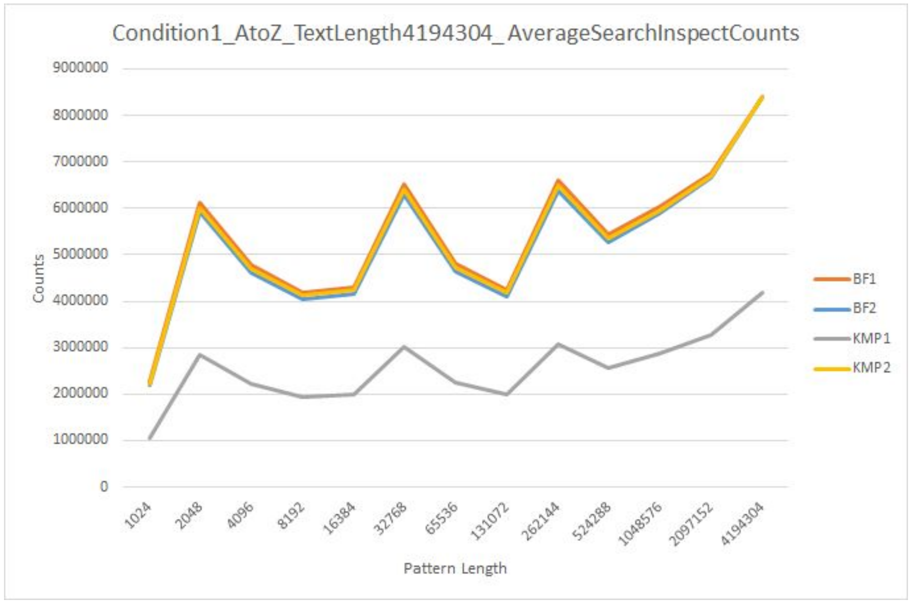


Figure 6. Average Search Inspect Counts for Condition1 AtoZ Text Length 4194304

For the same condition, we also generated the graph for min text length(16384).

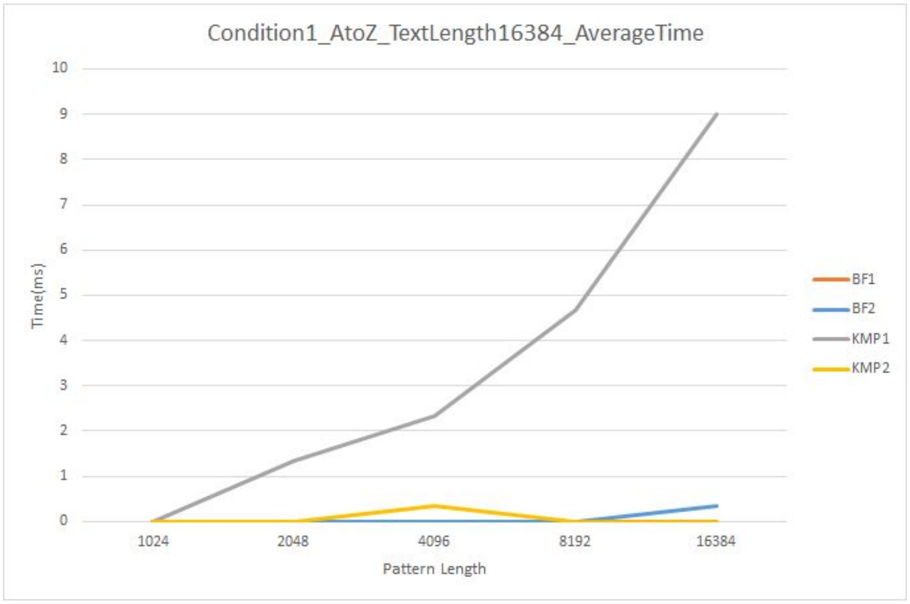


Figure 7. Average Time for Condition1 AtoZ Text Length 16384

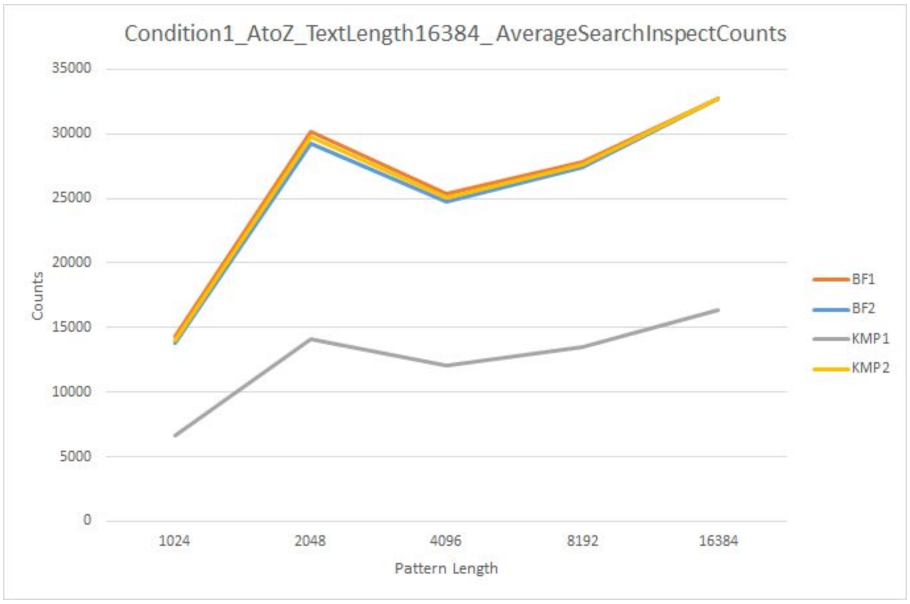


Figure 8. Average Search Inspect Counts for Condition1 AtoZ Text Length 16384

Then, we were also curious about what if the pattern is not in the text. So we generated figures for condition 2 with the same other parameters.

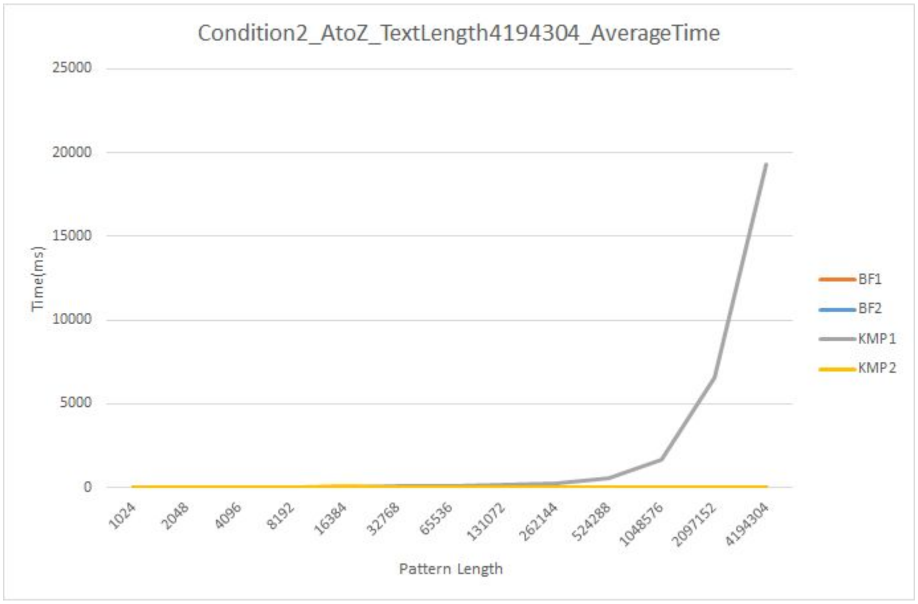


Figure 9. Average Time for Condition2 AtoZ Text Length 4194304

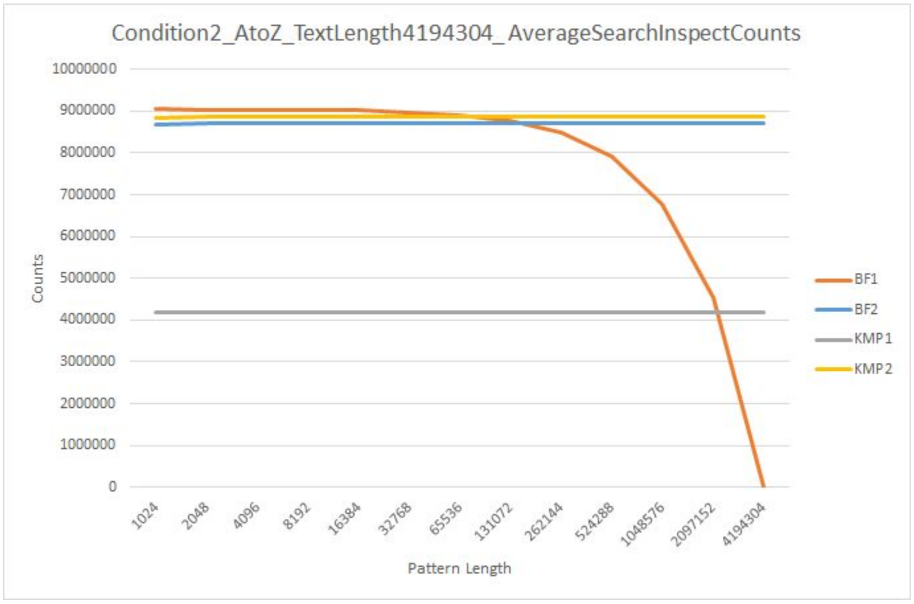


Figure 10. Average Search Inspect Counts for Condition2 AtoZ Text Length 4194304

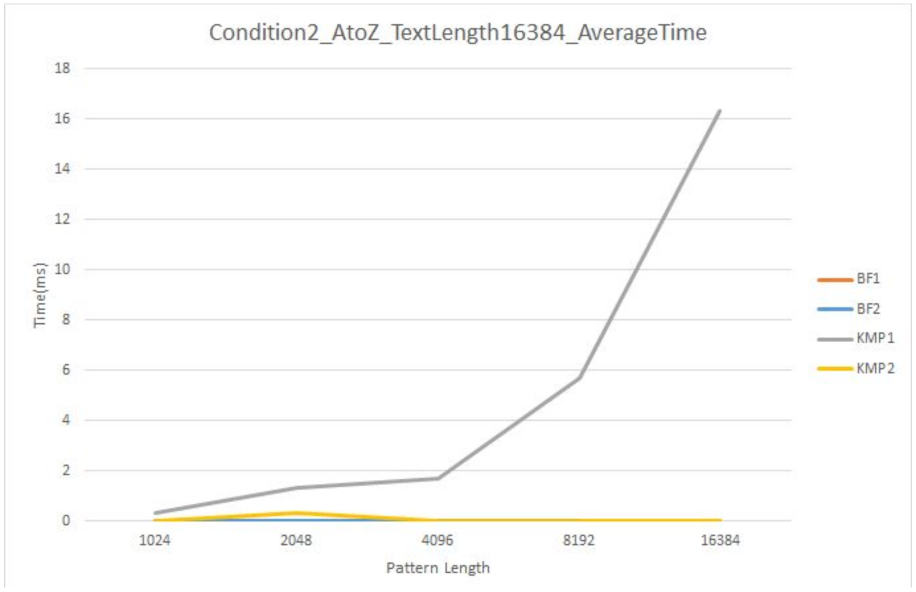


Figure 11. Average Time for Condition2 AtoZ Text Length 16384

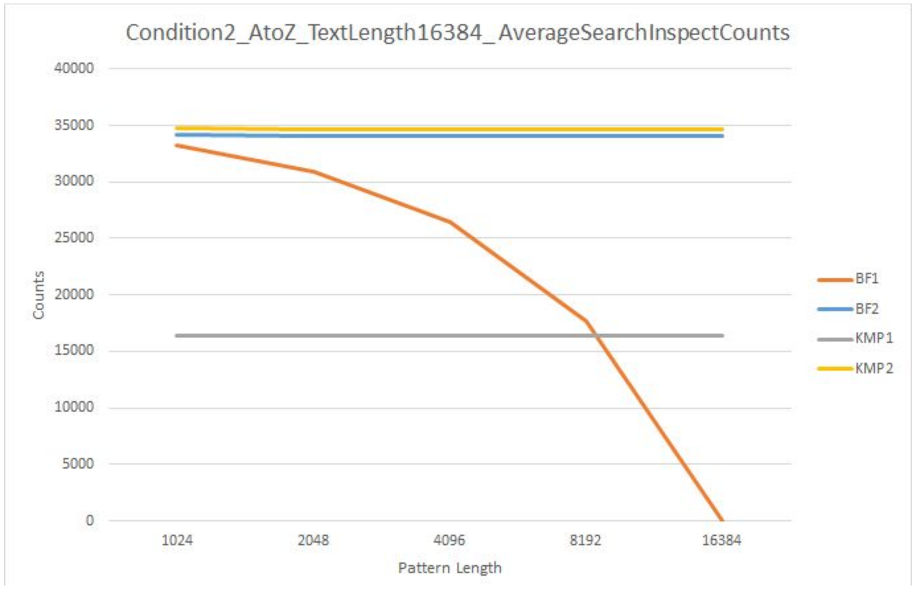


Figure 12. Average Search Inspect Counts for Condition2 AtoZ Text Length 16384

We also wanted to provide some figures without KMP1 so we can see more details.

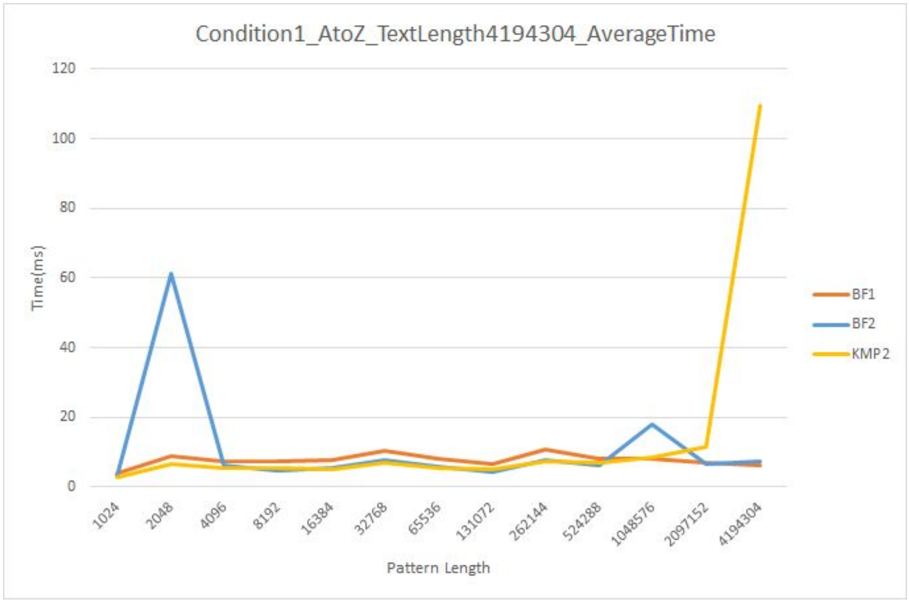


Figure 13. Average Time for Condition1 AtoZ Text Length 4194304 without KMP1

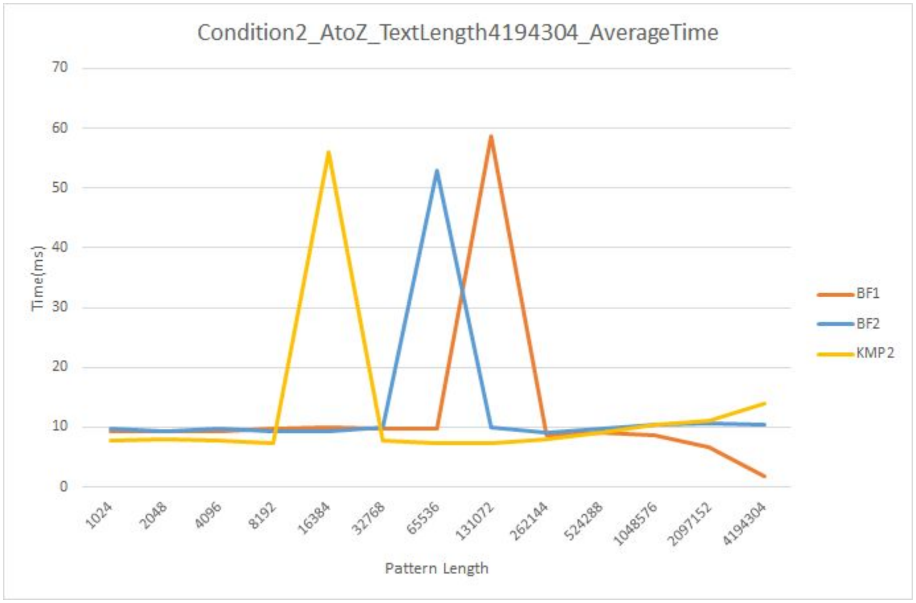


Figure 14. Average Time for Condition2 AtoZ Text Length 4194304 without KMP1

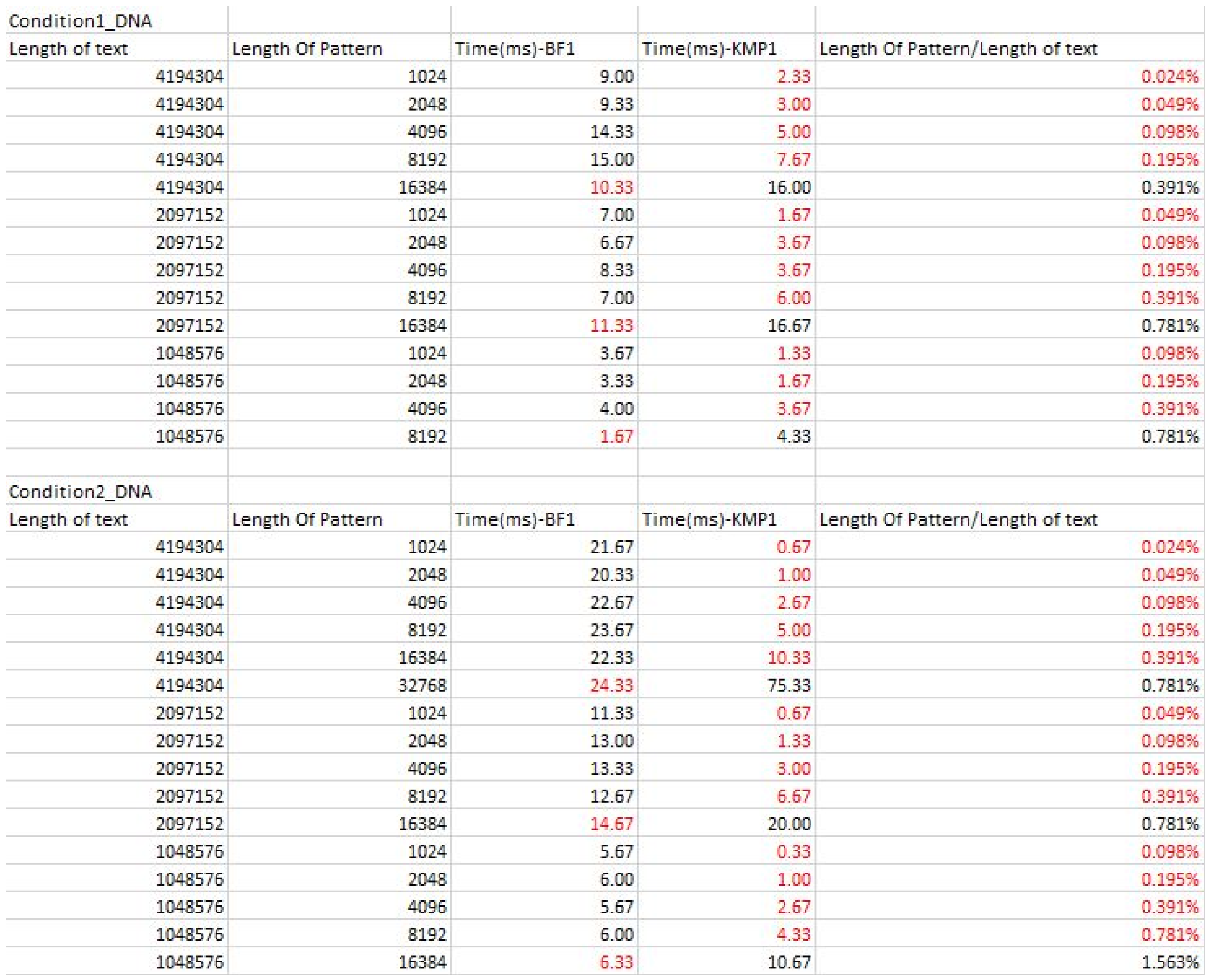
Although we did not include the figure for the other two alphabets, the data trends are similar. In addition, we selected some pieces of data to compare the performance between BF1 and KMP1.

Condition1_AtoZ				
Length of text	Length Of Pattern	Time(ms)-BF1	Time(ms)-KMP1	Length Of Pattern/Length of text
4194304	1024	4.00	1.67	0.024%
4194304	2048	9.00	5.67	0.049%
4194304	4096	7.33	6.33	0.098%
4194304	8192	7.33	12.00	0.195%
2097152	1024	3.00	1.33	0.049%
2097152	2048	3.33	2.33	0.098%
2097152	4096	4.33	4.67	0.195%
1048576	1024	1.33	1.00	0.098%
1048576	2048	1.67	1.67	0.195%

Condition2_AtoZ				
Length of text	Length Of Pattern	Time(ms)-BF1	Time(ms)-KMP1	Length Of Pattern/Length of text
4194304	1024	9.33	6.67	0.024%
4194304	2048	9.33	8.00	0.049%
4194304	4096	9.33	10.33	0.098%
4194304	8192	9.67	20.67	0.195%
2097152	1024	4.00	3.67	0.049%
2097152	2048	4.00	4.00	0.098%
2097152	4096	4.67	6.00	0.195%
1048576	1024	2.00	2.33	0.098%
1048576	2048	1.67	2.33	0.195%

Condition1_Binary				
Length of text	Length Of Pattern	Time(ms)-BF1	Time(ms)-KMP1	Length Of Pattern/Length of text
4194304	1024	13.67	2.67	0.024%
4194304	2048	13.33	3.00	0.049%
4194304	4096	18.33	5.00	0.098%
4194304	8192	14.33	7.00	0.195%
4194304	16384	8.33	11.00	0.391%
4194304	32768	26.67	34.00	0.781%
2097152	1024	9.00	2.33	0.049%
2097152	2048	11.00	2.67	0.098%
2097152	4096	5.00	3.33	0.195%
2097152	8192	9.33	6.00	0.391%
2097152	16384	5.67	10.00	0.781%
2097152	32768	15.00	22.00	1.563%
1048576	1024	4.67	1.00	0.098%
1048576	2048	4.00	3.00	0.195%
1048576	4096	5.00	2.67	0.391%
1048576	8192	7.33	4.67	0.781%
1048576	16384	6.00	15.00	1.563%

Condition2_Binary				
Length of text	Length Of Pattern	Time(ms)-BF1	Time(ms)-KMP1	Length Of Pattern/Length of text
4194304	1024	30.00	6.00	0.024%
4194304	2048	30.00	6.67	0.049%
4194304	4096	30.33	8.00	0.098%
4194304	8192	31.00	11.00	0.195%
4194304	16384	31.00	16.00	0.391%
4194304	32768	31.67	74.67	0.781%
2097152	1024	14.67	3.33	0.049%
2097152	2048	14.67	4.00	0.098%
2097152	4096	15.00	6.00	0.195%
2097152	8192	15.67	8.33	0.391%
2097152	16384	15.00	17.67	0.781%
1048576	1024	8.33	2.33	0.098%
1048576	2048	9.00	2.67	0.195%
1048576	4096	8.33	4.00	0.391%
1048576	8192	9.33	6.33	0.781%
1048576	16384	8.33	13.67	1.563%



Condition1_DNA

Length of text	Length Of Pattern	Time(ms)-BF1	Time(ms)-KMP1	Length Of Pattern/Length of text
4194304	1024	9.00	2.33	0.024%
4194304	2048	9.33	3.00	0.049%
4194304	4096	14.33	5.00	0.098%
4194304	8192	15.00	7.67	0.195%
4194304	16384	10.33	16.00	0.391%
2097152	1024	7.00	1.67	0.049%
2097152	2048	6.67	3.67	0.098%
2097152	4096	8.33	3.67	0.195%
2097152	8192	7.00	6.00	0.391%
2097152	16384	11.33	16.67	0.781%
1048576	1024	3.67	1.33	0.098%
1048576	2048	3.33	1.67	0.195%
1048576	4096	4.00	3.67	0.391%
1048576	8192	1.67	4.33	0.781%

Condition2_DNA

Length of text	Length Of Pattern	Time(ms)-BF1	Time(ms)-KMP1	Length Of Pattern/Length of text
4194304	1024	21.67	0.67	0.024%
4194304	2048	20.33	1.00	0.049%
4194304	4096	22.67	2.67	0.098%
4194304	8192	23.67	5.00	0.195%
4194304	16384	22.33	10.33	0.391%
4194304	32768	24.33	75.33	0.781%
2097152	1024	11.33	0.67	0.049%
2097152	2048	13.00	1.33	0.098%
2097152	4096	13.33	3.00	0.195%
2097152	8192	12.67	6.67	0.391%
2097152	16384	14.67	20.00	0.781%
1048576	1024	5.67	0.33	0.098%
1048576	2048	6.00	1.00	0.195%
1048576	4096	5.67	2.67	0.391%
1048576	8192	6.00	4.33	0.781%
1048576	16384	6.33	10.67	1.563%

Figure 14. Relationship between Search Time and Length Ratio

Performance Evaluation Conclusion

In conclusion, the performance of algorithms usually depends on the conditions. It is obvious that when the length of the pattern is large KMP1 will need much more time than the other algorithms. This makes sense since it will take KMP much more time to construct a helper array based on pattern before starting search. However, it only did about less than half the number of inspections compared to other algorithms. The performance of the other three algorithms(BF1, BF2, KMP2) are similar in condition 1. There were only several little peaks which might just be special cases. But in condition 2, BF1 performs much better, especially when the length of the pattern is large. This is because BF1 has a special feature that if the length of the rest of the text is smaller than the length of the pattern, it will exit early.
The other thing we discovered is the advantage of the KMP1 algorithm. KMP1 performs very high efficiency when we have a long text and a short pattern while BF1 seems consistent on processing time. From figure 14, we can see that the smaller the ratio pattern and text have, the faster KMP1 will perform. There exists a ratio that determines whether BF1 performs better or KMP1 performs better.

Alphabet	Condition	Dividing Ratio
a-z	1	0.1%
a-z	2	0.075%
Binary	1	0.6%
Binary	2	0.65%
DNA	1	0.39%
DNA	2	0.5%

As a result, if you have a very long pattern, KMP should not be your first choice. If you have a long text and a short pattern, like searching for a word in a book, you should always choose KMP1.

Reference

Knuth, Donald E., Jr. James H. Morris, and Vaughan R. Pratt. “Fast Pattern Matching in Strings.” *SIAM Journal on Computing* 6, no. 2 (1977): 323–50. https://doi.org/10.1137/0206024.

Mission-Peace, “interview,” *Github*, accessed May 3, 2020. https://github.com/mission-peace/interview/blob/master/src/com/interview/string/SubstringSearch.java

Sedgewick, Robert, and Kevin Wayne. *Algorithms, Fourth Edition*. Addison-Wesley Professional, 2011.