SUBSTRING SEARCH

CS2223 D20 - Final Project

*

Team Leiopsammodius
Hailey Anderson, Evelyn Tran, Cindy Trac, Sitsanok Young

I. KMP SUBSTRING SEARCHING ALGORITHM

A fundamental operation on strings that is often utilized in everyday life is *substring search*. As the title itself suggests, it looks for a particular piece of text inside a larger text. Similarly, in mathematics, when looking for a subset of a given set, the subset is just a part of the set. When given a text string and a pattern string, the operation looks to find an occurrence of the pattern within the text. Whether it be using the find tool on a webpage or simply looking up something on Google, substring search is used by computer scientists and regular people alike.

The naive approach to substring search would be to use brute force. The brute force approach goes character-by-character through the text, checking to see if the given pattern is within the text. It does not implement any sort of strategy when looking for the substring; it just goes in blind with a plan of iterating through all of the positions of the text. If the pattern is found sooner, the algorithm will be finished and will not need to process the remainder of the text; however, if it ends up not finding the pattern, or the pattern ends up being the last character(s) of the text, then the running time and number of comparisons made are increased. Thankfully, there are other implementations of substring search.

The Knuth-Morris-Pratt (KMP) substring searching algorithm is the strategy we will be focusing on in this report. This is a much more efficient method compared to the brute force approach. The brute force approach has a performance of O(M*N), whereas the KMP method provides a worst-case performance of O(M+N). The KMP algorithm is based on the idea that when a mismatch occurs, the previous characters have already been checked. With this in mind, we can avoid backtracking and moving the text pointer over previous characters. It does this by preprocessing a given pattern and constructs a deterministic finite-state automaton (dfa) array to track the pattern progress to decide the next character desired in the text. “This program does the same job as the brute-force method, but it runs faster for patterns that are self-repetitive” (Sedgewick 768). In the rest of the report, we will be discussing our findings about the KMP substring search algorithm.

II. WORST-CASE STRINGS

In the worst case, the KMP algorithm makes M + N comparisons: M being the length of the pattern and N being the length of the text. We found that the type of pattern/text pairs that make KMP work the hardest are the ones where 1.) each position in the pattern is found in the text, except the very last position in which there is a mismatch, and 2.) the pattern and length are exactly the same. Starting the discussion with our first point, if there are multiple sets of mismatch patterns, then this forces the DFA to reset and start its search all over again. If the text is simply the pattern except the last character, then that makes the DFA go through each position in the pattern and the last position prompts an array inspection since the DFA needs to check if the text character and the pattern character match. As for the second point, it is the same case as the previous sentence; every character in the pattern has been checked in the pattern. The table below includes examples of worst case scenarios that we tested. The green rows correspond with our observations for the first point and the blue row corresponds with the second. If we wanted to find a pattern/text pair that caused a worst case scenario for the KMP with a large number of comparisons, we could easily code a program where a pattern is given and the text is repeatedly created with that pattern except for its last character. Since the worst case is M + N, we can never exceed it. The largest number of comparisons using the KMP could be infinitely large, but it’ll always be M + N.

Pattern	Text	M	N	M + N	Actual
aaaaa	aaaabaaaataa aasaaaa	5	19	24	24
abc	abgabtabc	3	9	12	12
qwertyuiop	qwertyuiol	10	10	20	20
2223	22212224222 3	4	12	16	16
sec	sec	3	3	6	6

III. PERFORMANCE OF KMP ALGORITHM AND BRUTE FORCE

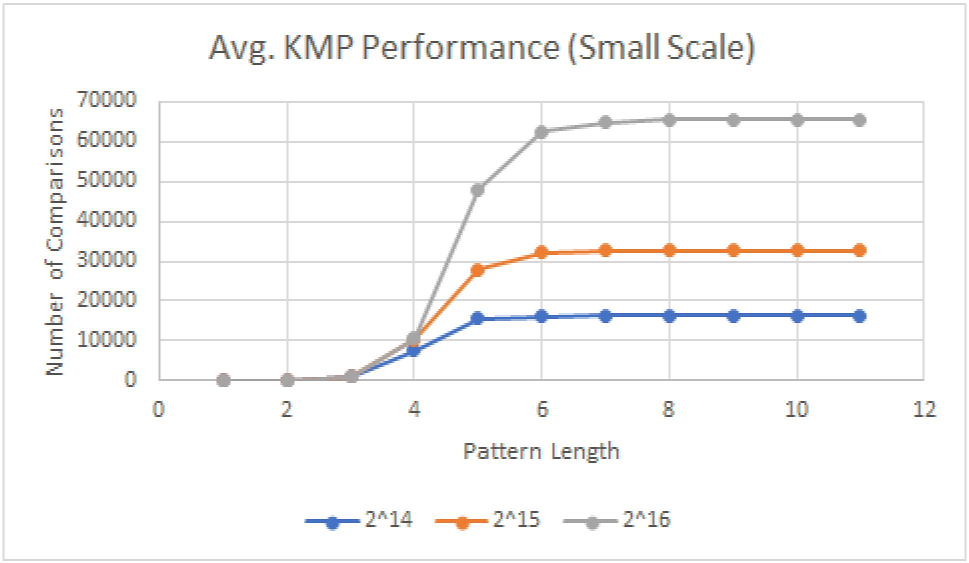
For the sake of readability, we have combined our evaluations of the KMP and brute force approaches into one section.

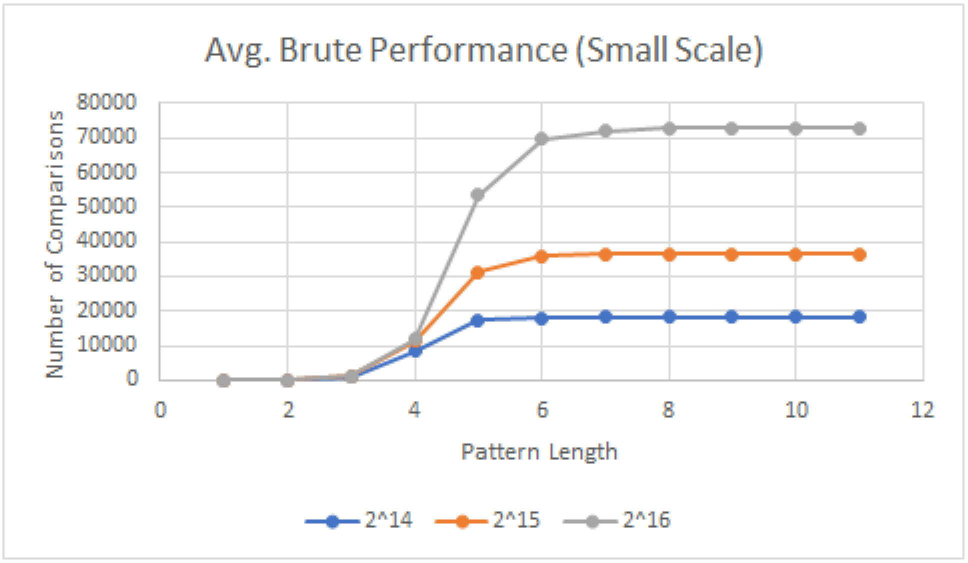
To evaluate our algorithm and the brute force performance, we used random patterns and text strings that range from 2^{14} (that is 16384) to 2^19 (that is, 5,242,88), each varying between the different alphabets. We used the following four alphabets when constructing the patterns and strings to be tested: digits of pi, DNA sequencing, standard "a-z" lower case letters, and binary numbers.

DIGITS OF PI

The digits of pi alphabet consists of digits from the decimal numeral system (numbers ranging from 0-9). Due to the alphabet size, there are ten different possible characters that the pattern's prefix needs to match. In other words, the chances of finding the first character of the pattern in the text will not be as high as DNA sequencing and binary number alphabets as discussed later on in this report. From our findings for the digits of pi of lengths of 2^{14}, 2^{15}, and 2^{16}, it appears that the number of comparisons increases as the text size increases. We believe this may have to do with the fact that the text string for pi is instead specific rather than more random like the others below.

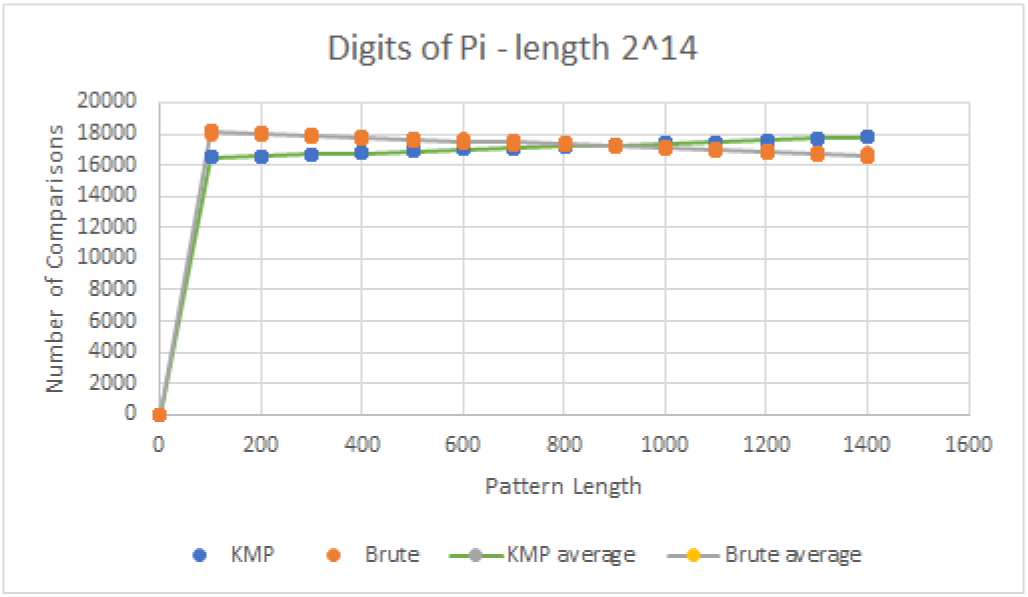
Towards the beginning when the pattern length is less than 4, the number of comparisons are very close to one another between the three text lengths; however, once we get to a pattern length of 4, the number of comparisons are a little more different. At the length of 5, the number of comparisons for all three text lengths skyrockets and are noticeably distinct, and at the length of 6, the number of comparisons increase just a bit more. Finally, with a pattern length of 7 and greater, there seems to be something similar to a plateau, but not quite since the number of comparisons is still slightly increasing. When looking at the data for the digits of pi, we noticed that the number of comparisons for 2^{14} was about half the number of comparisons for 2^{15}, and the same observation was made with 2^{15} and 2^{16}. We believe that as the text size doubles, the number of comparisons doubles as well.

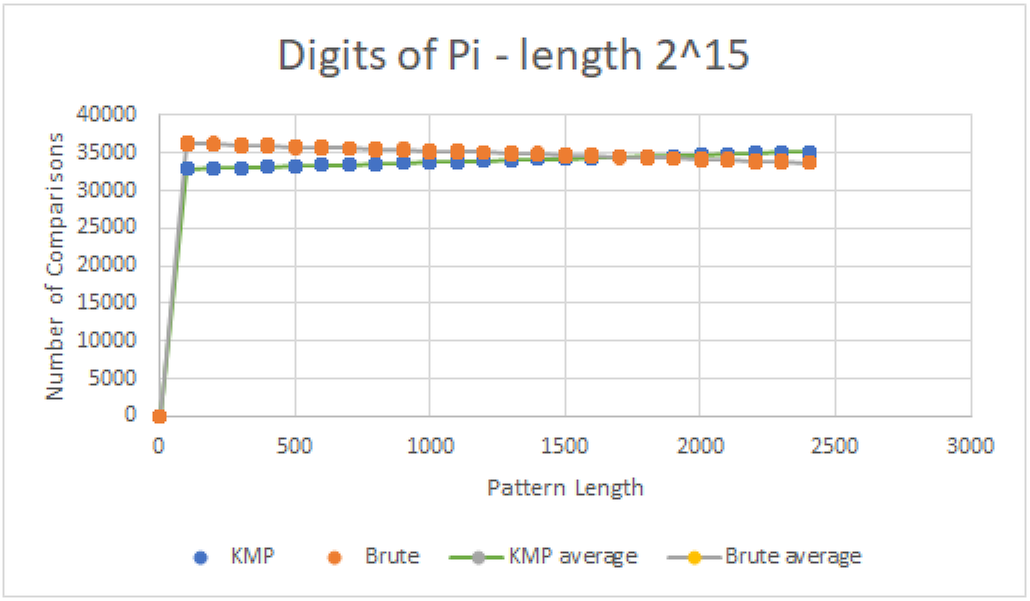


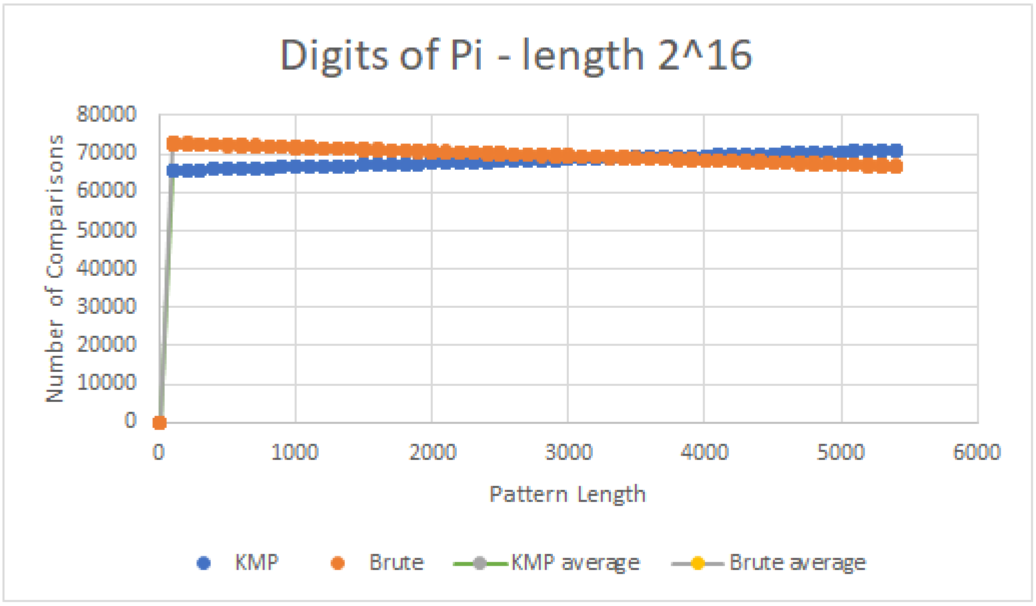


The brute force approach is truly a contender to the KMP algorithm when searching for a pattern in the digits of pi. When searching for a pattern of length 1, this implementation always outperforms the KMP method by 1 comparison; however, it does lose to KMP for a while after that until it makes a heroic comeback later on. For a text length of 2^{14}, we start to see the naive approach have a better performance than KMP when we have a pattern whose length is somewhere around the high 800s. For a text length of 2^{15}, we see that happen as well for a pattern whose length is around a quarter into the 1700s, and similarly for 2^{16}, this happens around mid 3400s. Notice how the number of comparisons where the outperformance by the brute force doubles as the text size doubles as well.

For the three graphs below and in the class "Pi," we chose to increment the pattern lengths by 100 in order to reach a higher pattern length and see where the intersection between KMP and the brute force algorithm occurs.

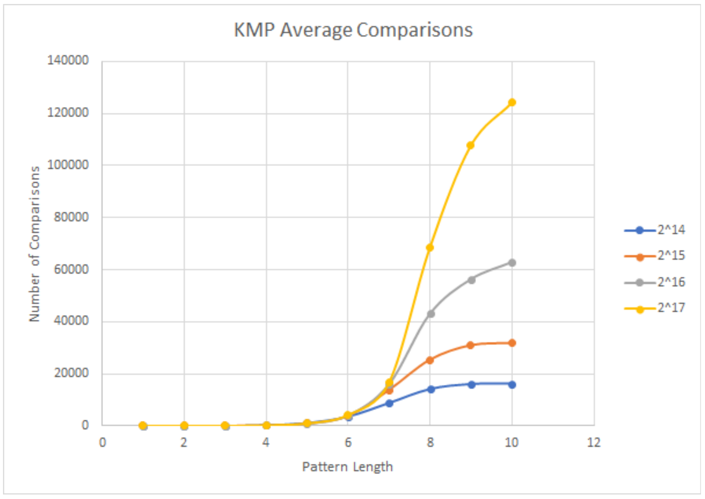

Digits of Pi - length 2^14
Number of Comparisons
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
200
400
600
800
1000
1200
1400
1600
Pattern Length
KMP
Brute
KMP average
Brute average


Digits of Pi - length 2^15
Number of Comparisons
40000
35000
30000
25000
20000
15000
10000
5000
0
0
500
1000
1500
2000
2500
3000
Pattern Length
KMP
Brute
KMP average
Brute average

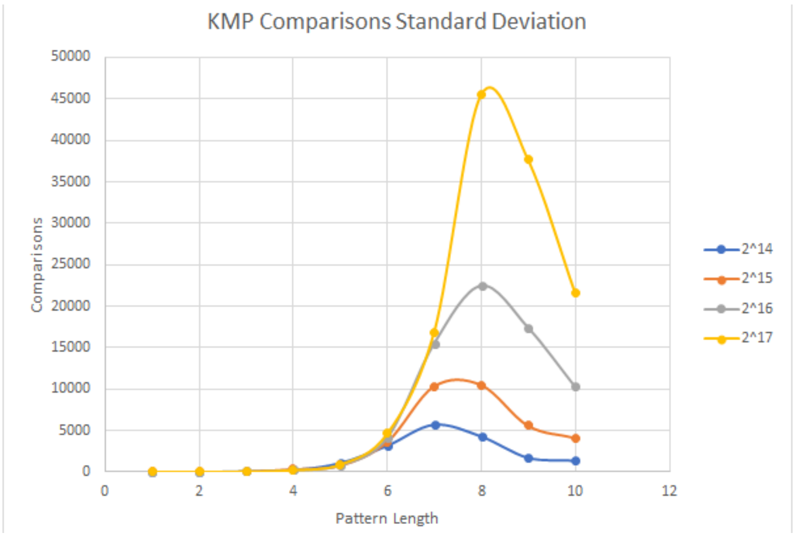


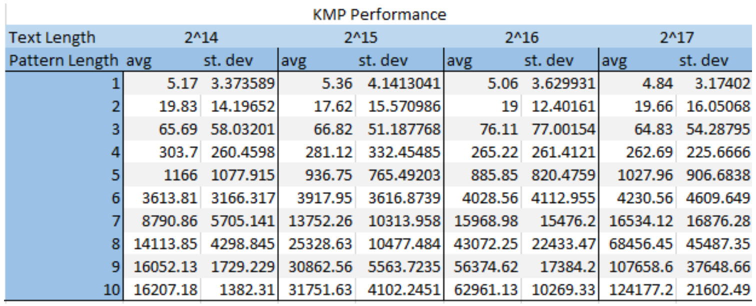
DNA SEQUENCING

The DNA sequencing alphabet is made up of 4 letters, “A”, “C”, “T”, and “G”. Because of its smaller alphabet size, the likelihood of encountering the pattern in the text is higher than that of a larger alphabet. Between the pattern lengths 1 to 6, the text lengths did not affect the number of comparisons, with their averages being around the same. With pattern lengths 7 and more, the greater the text length was, the more comparisons there were.



The number of comparisons had a smaller variance when the pattern was a smaller length and the spread of the comparisons increased as the pattern length increased due to the different likelihoods of finding a particular sequence in the text. For example, finding "A" in a text is more likely because in the randomly generated text, there is a 25% chance of getting an "A", whereas "AA" has a 6.25%. That chance continues to decrease based on the pattern length, having a probability of 0.25^(N) where N is the pattern length. At around pattern length 8, the spread of the number of comparisons is at its highest for all 4 of the different text lengths and means that the probability of the pattern appearing is just enough for it to appear anywhere in the text. The spread begins to decrease after because the likelihood of the pattern appearing is even smaller, therefore was more likely to be found later on in the text.



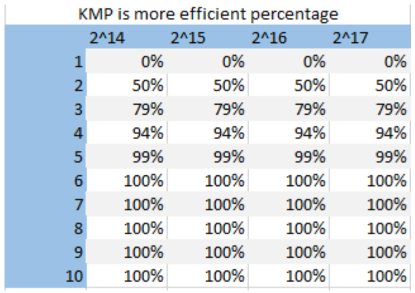


KMP Performance

Text Length	2^14		2^15		2^16		2^17	
Pattern Length	avg	st. dev	avg	st. dev	avg	st. dev	avg	st. dev
1	5.17	3.373589	5.36	4.1413041	5.06	3.629931	4.84	3.17402
2	19.83	14.19652	17.62	15.570986	19	12.40161	19.66	16.05068
3	65.69	58.03201	66.82	51.187768	76.11	77.00154	64.83	54.28795
4	303.7	260.4598	281.12	332.45485	265.22	261.4121	262.69	225.6666
5	1166	1077.915	936.75	765.49203	885.85	820.4759	1027.96	906.6838
6	3613.81	3166.317	3917.95	3616.8739	4028.56	4112.955	4230.56	4609.649
7	8790.86	5705.141	13752.26	10313.958	15968.98	15476.2	16534.12	16876.28
8	14113.85	4298.845	25328.63	10477.484	43072.25	22433.47	68456.45	45487.35
9	16052.13	1729.229	30862.56	5563.7235	56374.62	17384.2	107658.6	37648.66
10	16207.18	1382.31	31751.63	4102.2451	62961.13	10269.33	124177.2	21602.49

For a pattern length of 1, the brute force method always performed better than the KMP by 1 comparison. This makes sense because the KMP method is incrementing by 1 just like the brute force method, but has 1 more comparison when making the DFA array. At a pattern length of 2, the brute force and KMP method varies in which performs better. About half the time brute force will perform better, and the other half, KMP will have less comparisons. Generally if the pattern is found further along in the text, KMP will outperform the brute force method. For pattern lengths of 3 and up, KMP has less comparisons than the brute force method, therefore outperforming it. We chose to only go up to pattern lengths 10 because it was clear that KMP would be more efficient than brute onwards for all the text lengths.

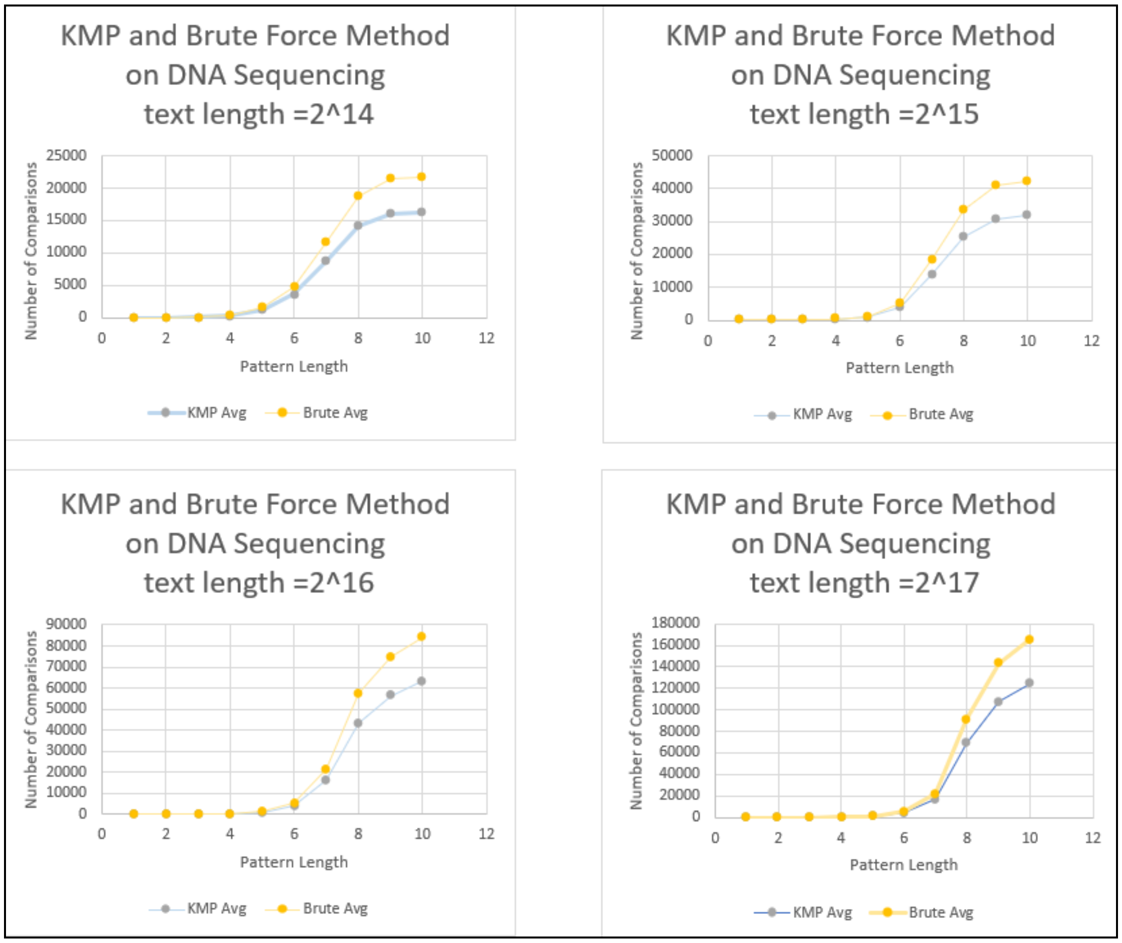
The different text lengths did not affect the performance of either the brute force method or the KMP method because of the small alphabet and the large text size. The data below shows that between each of the different text lengths, that they all have the same percentage, however I believe that the data below is somewhat incorrect because we did not implement a seed change when randomizing our texts, but overall the analysis should generally be about the same.



KMP is more efficient percentage

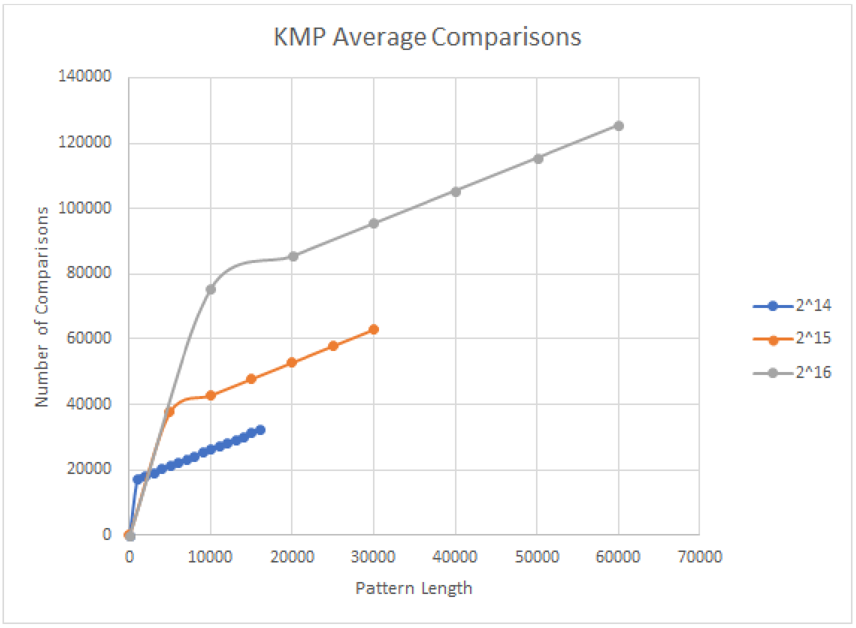
	2^14	2^15	2^16	2^17
1	0%	0%	0%	0%
2	50%	50%	50%	50%
3	79%	79%	79%	79%
4	94%	94%	94%	94%
5	99%	99%	99%	99%
6	100%	100%	100%	100%
7	100%	100%	100%	100%
8	100%	100%	100%	100%
9	100%	100%	100%	100%
10	100%	100%	100%	100%

Between the different text lengths, they follow the same trend where KMP will be more efficient than the brute force method, however the number of comparisons increased as the text lengths increased for both methods. The graphs below show that they all follow a similar shape, however the y-axis's scale differs between them.



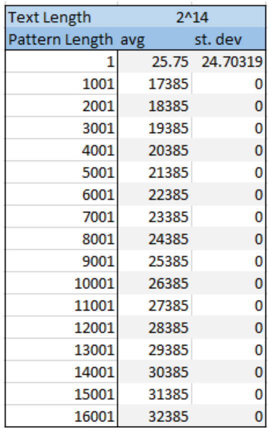
STANDARD "a-z" LOWER CASE LETTERS

The lowercase letters a-z has an alphabet of 26 letters, therefore having the largest alphabet size out of the all the alphabets in this paper. Because of this, we were only able to test up to a text length of 2^{16} due to how long it would take for the program to terminate. The comparisons were significantly greater for the KMP search compared to the DNA sequences and binary because the larger alphabet size meant that the probability of each letter occuring was 1/26 or 3.85%. That means as the pattern length increased, the likelihood of the pattern occurring would be 3.85^n %, where n is the pattern length. Similar to the other alphabets, as the text size increased, so did the number of compares.

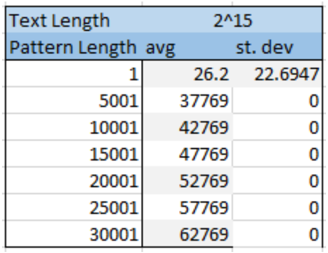


Each of the pattern lengths increment at different rates due to the time it took to run the program, and also to cover the range of each text length. The standard deviation comes out to be 0 for every pattern length except for 1 for either 2 possibilities:

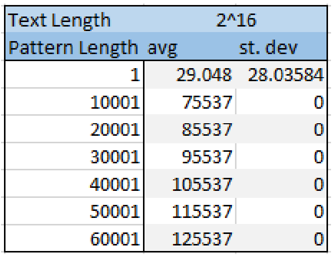
- As seen in the trend in the DNA sequencing, the standard deviation decreases at some point as the pattern lengths get larger. So because these pattern lengths are so large, the patterns are generally found in the same part of the text, so the spread is so small that the standard deviation is approximately 0.
- Because we did not include a seed change in our code, the randomization may not have changed throughout trials, causing the data to be skewed. However, the corresponding brute force data for each pattern and text differ, so we do know that the position changes between trials.



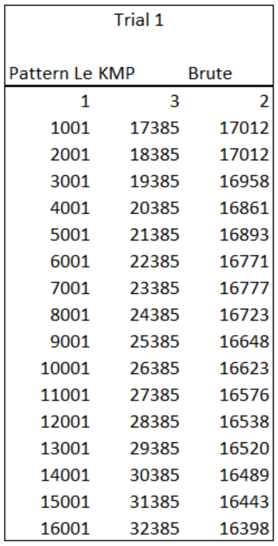
Text Length	2^14	
Pattern Length	avg	st. dev
1	25.75	24.70319
1001	17385	0
2001	18385	0
3001	19385	0
4001	20385	0
5001	21385	0
6001	22385	0
7001	23385	0
8001	24385	0
9001	25385	0
10001	26385	0
11001	27385	0
12001	28385	0
13001	29385	0
14001	30385	0
15001	31385	0
16001	32385	0



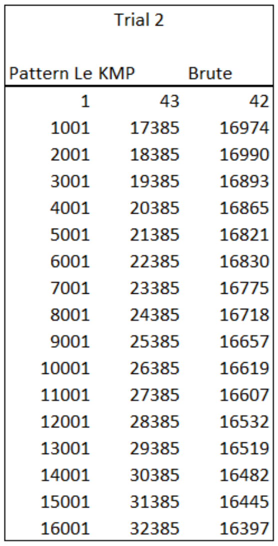
Text Length	2^15	
Pattern Length	avg	st. dev
1	26.2	22.6947
5001	37769	0
10001	42769	0
15001	47769	0
20001	52769	0
25001	57769	0
30001	62769	0



Text Length	2^16	
Pattern Length	avg	st. dev
1	29.048	28.03584
10001	75537	0
20001	85537	0
30001	95537	0
40001	105537	0
50001	115537	0
60001	125537	0

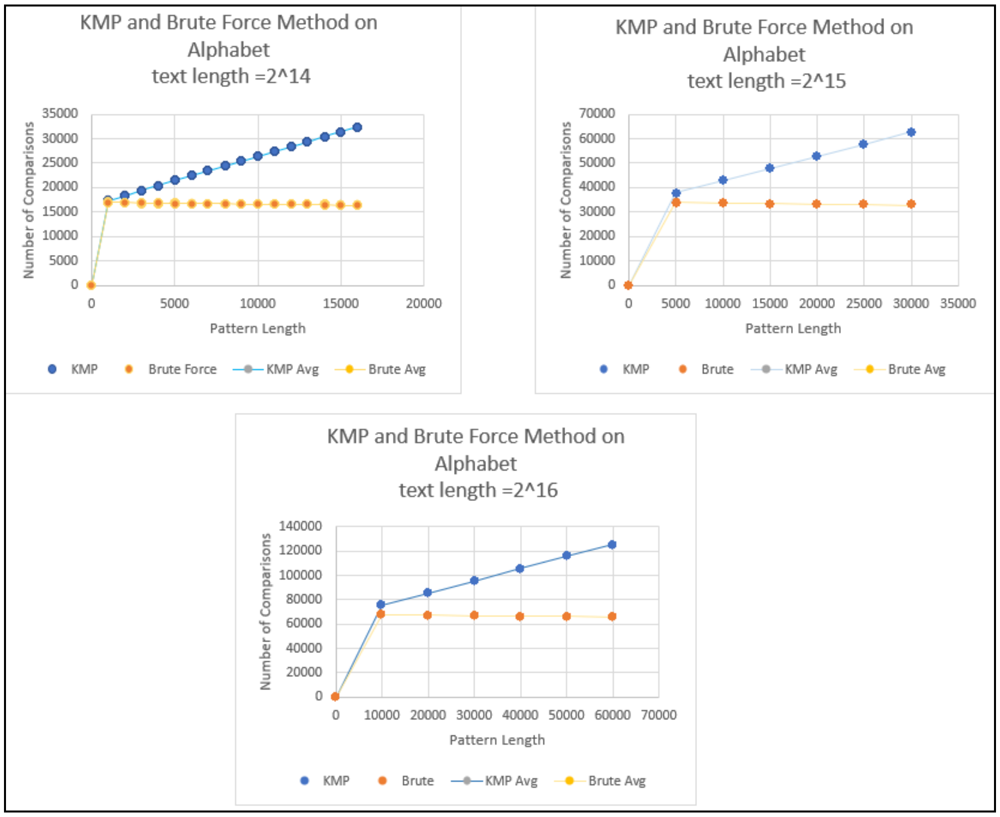


Trial 1		
Pattern Le	KMP	Brute
1	3	2
1001	17385	17012
2001	18385	17012
3001	19385	16958
4001	20385	16861
5001	21385	16893
6001	22385	16771
7001	23385	16777
8001	24385	16723
9001	25385	16648
10001	26385	16623
11001	27385	16576
12001	28385	16538
13001	29385	16520
14001	30385	16489
15001	31385	16443
16001	32385	16398



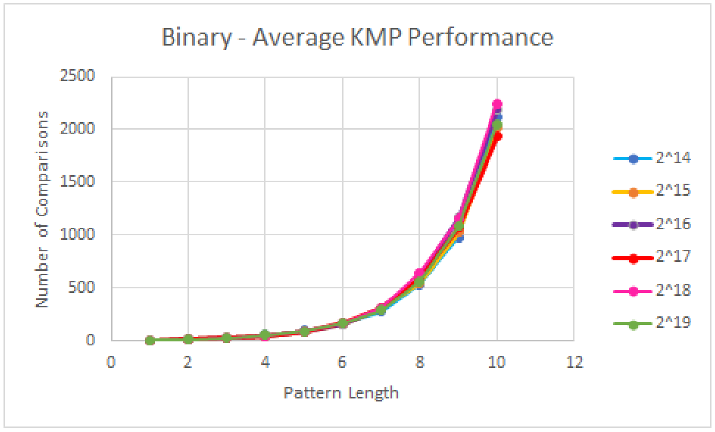
Trial 2		
Pattern Le	KMP	Brute
1	43	42
1001	17385	16974
2001	18385	16990
3001	19385	16893
4001	20385	16865
5001	21385	16821
6001	22385	16830
7001	23385	16775
8001	24385	16718
9001	25385	16657
10001	26385	16619
11001	27385	16607
12001	28385	16532
13001	29385	16519
14001	30385	16482
15001	31385	16445
16001	32385	16397

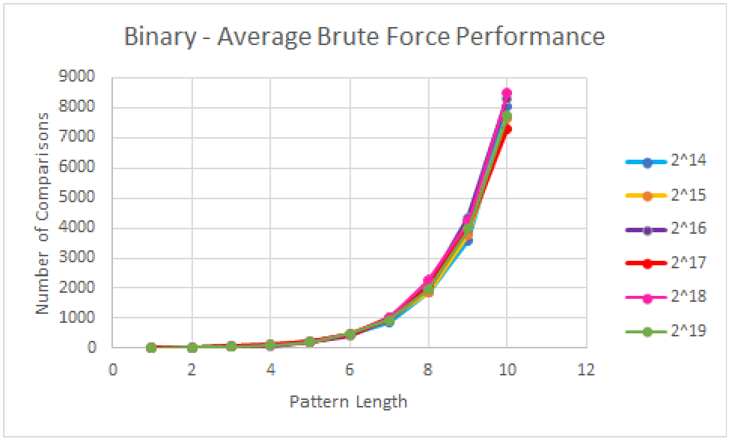
The brute force method performed better than the KMP method at every text length and pattern size. We chose to increment at the pattern lengths different sizes to see if at any point the KMP would perform better than brute force. However, as seen in the graphs below, substring search using brute force was always more efficient, meaning that KMP does not do well for larger alphabet sizes. This makes sense because KMP works best with repetition and because the probability of each letter appearing in the text is so small, the probability of repetition is also very small.



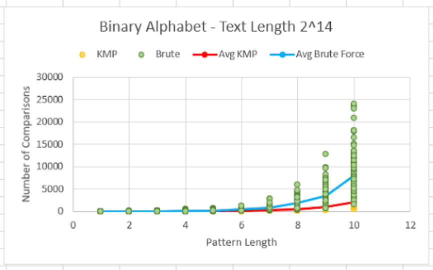
BINARY

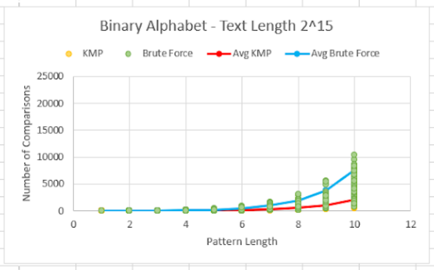
The binary alphabet consists of “0” and “1.” Because the binary number system has such a small alphabet, this increases the chance of a collision. As you can see from the Excel sheets and data, KMP’s performance (in terms of average comparisons) is roughly the same despite text length. This is because the binary alphabet is so small that the chances of parts of a pattern recurring many times is very high. So, even when you increase the text length, it will not affect the average number of comparisons as much as if you were using a larger alphabet. This applies to both the KMP substring search and the brute force methods.

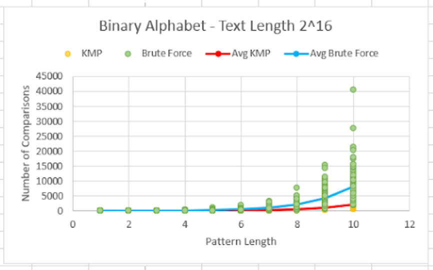


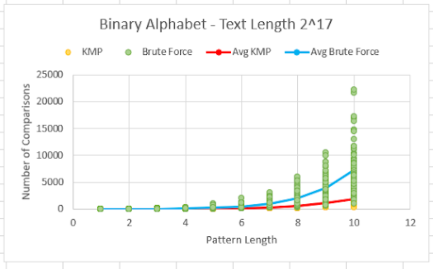


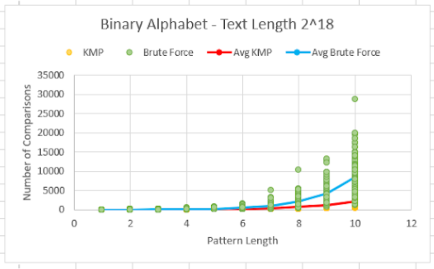
As you can see from the graphs of the averages for both the KMP algorithm and brute force approach, the number of comparisons for each pattern length stays relatively the same across all of the tested text lengths. Please note that since the binary alphabet is so small, we decided to only test up to a pattern length of 10 because the sequences of binary numbers will repeat at a higher frequency than that of a larger alphabet. However, because the alphabet is so small, we were able to test it up to a text length of 2^{19}. We could have tested up through a text length of 2^{20}, but 2^{19} already took close to 3 hours to run.

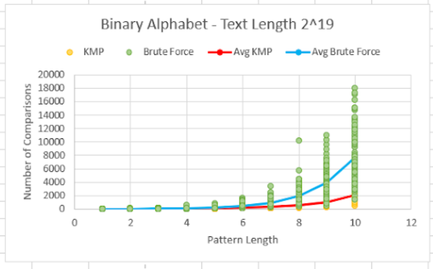












For a pattern length of 1, KMP is only better than brute force by 1. From there, KMP continues to find the given substring in less comparisons than brute force. Please refer to the Binary Excel sheet for trial numbers.

IV. OUTPERFORMANCE BY KMP

When analyzing text strings of size N to determine where KMP outperforms the brute force algorithm, we noticed that the alphabet in which we used did matter. For alphabets of size 2, KMP has a better performance when the pattern length is 2 or greater. For size 4, the outperformance occurs when the pattern length is 3 or greater. For size 26, we were unable to find where the intersection happens since it was too large of a number for our computers to process. Unlike the other 3 alphabets, pi is pi. It is a constant, which means that it isn't randomly generated like the text sequences. With this in mind, it would make sense that its intersection varies between the different text lengths. As discussed earlier, the point where the intersection occurs doubles as the text length doubles. In contrast to the other 3, KMP actually outperforms the brute force algorithm until the intersection (excluding pattern length 1).

V. BIBLIOGRAPHY

(1977) Fast Pattern Matching in Strings. D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. *SIAM J. Comput.*, 6(2), 323–350.

(2011) Algorithms, 4th Edition. R. Sedgewick, and K. Wayne. *Addison-Wesley.*, Section 5.3, 758-769.