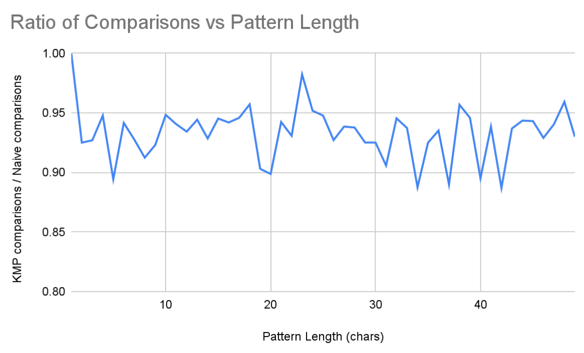
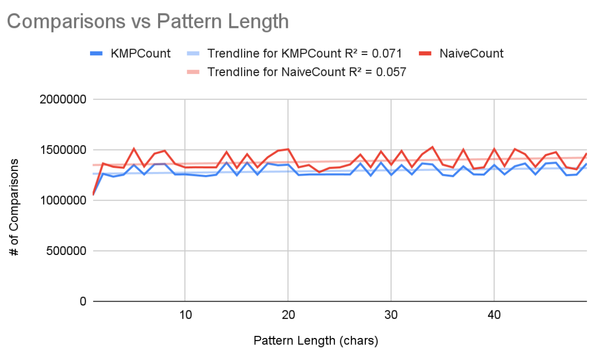
KMP Algorithm Final Report

Summary:

The datasets we used for benchmarking are taken from the human genome sequence. This is a 400 Mb .txt file that we took varying sized datasets from to test how changing the pattern length and data length affects the efficiency of the KMP algorithm.
The graph 1 and 2 show the effect of varying the pattern length. We took pattern lengths from 1 to 50 characters with a fixed dataset length of 100 Mb. The patterns were created randomly by choosing from the letters A, G, T, and C which are the same letters the dataset is made up of.

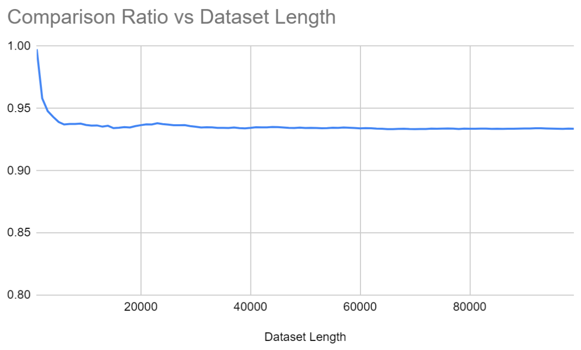
The graph 1 is the ratio of the KMP solution's comparisons to the naive solution's comparisons. On average, the number of comparison KMP algorithm used is 93.7% compared to the naive algorithm (KMP comparisons / naive comparisons). The graph 2 shows two lines. The red line represents the number of comparisons for the naive solution. The blue line is the number of comparisons the KMP algorithm took. The x-axis is the pattern length, and the y-axis is the number of comparisons each algorithm took. The number of comparisons stays about the same as the length of pattern increases, and there's no strong linear relationship since both R^2 values are less than 0.1.

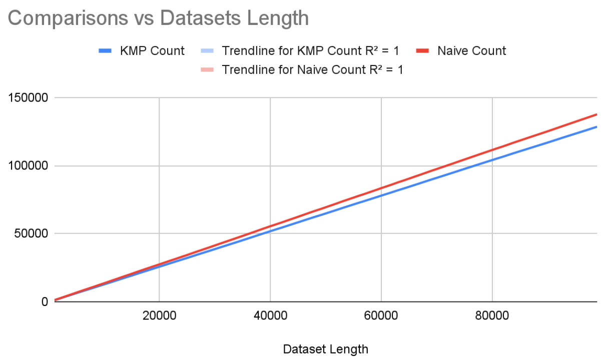




Graph 1, 2 Performance of KMP algorithm vs Pattern length

The graph 3 and 4 show the effects of varying dataset sizes. The dataset size was varied from 1000 to 100,000 characters, each time increased by 1000. The graph 3 is the ratio of comparisons between the naive and KMP algorithms. For small datasets less than 5000 characters, the KMP algorithm is consistently better than naive algorithm but not by a significant amount. For large datasets, the percent improvement plateaus at about 93.5% when compared to the naive solution (KMP comparisons / naive comparisons). The graph 4 has two lines. The red for naive and blue for KMP. The x-axis is the length of the dataset, and y-axis is the number of comparisons each algorithm took. Both naive and KMP algorithm has a strong linear relation between number of comparison and dataset length, with both has the R^2 value of 1.





Graph 3, 4 Performance of KMP Algorithm vs Dataset Length

- The output and correctness of your algorithm – You should go over your entire test suite and explain in words how the total test suite has proven your algorithm is correct. For example, include a manual calculation showing the correct output for a given input and give the name of the test that confirms the algorithm itself is getting the same answer.

To test the accuracy of our implementation for KMP algorithm, we have first implemented the naive algorithm for searching. It found the matching pattern by iterating through the data set and pattern one by one. The naive algorithm was thoroughly testing using tests that check for pattern matches at first, last, and overlapping characters.

For example, the testcase "Naive simple 1" has pattern of "ABC";, and dataset "ABCuhluhu". And we should expect the result to be {0}, as the pattern only showns up at the index 0.

We use the naive algorithm to verify the accuracy and the efficiency of the KMP algorithm. Every test compares the KMP algorithm's result to the naive algorithms to check for accuracy. For example, the test "Test with input" calculates the naiveSearch as the “expected” output and compares this to the KMP algorithm's output.

We also test the components of the KMP algorithm individually. To verify the functionality of the LPS function we manually calculated the expected result. LPS stands for longest proper suffix that is also a prefix. We then compared this to the returned vector. The vector LPS stores the length of the desired prefix of the corresponding substring for the pattern. This would be used latter in the ‘search’ function. We have multiple tests that exhaustively test when the pattern includes overlapping and repetitive characters. We also tested the number of comparisons that the LPS function uses by calculating the expected value on paper before comparing it to the outputted comparison count. This ensured that the algorithm was logically correct.

For example, the testcase "test lps2" has pattern of "AABAACAABAA". In this case, for instance, the value at the index 4 is supposed to be “2”. Because for substring (0, 4), the longest proper suffix “AA” (0, 1) is also the prefix “AA” (3,4) has a length of 2. Similarly, the value at the index 10 is supposed to be 5, since “AABAA”(0, 4) and (6,10) has a length of 5. Therefore, the expected result is {0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}.

We then tested the comparisons for the full algorithm. The test "Test with input" compares the expected number of comparisons to the actual number of comparisons ensuring that our benchmarking is accurate to the expected efficiency.

Finally, we tested the KMP algorithm on large datasets. The algorithm outputs the location of the pattern matched in the text so it is easy enough to verify that the algorithm is pointing to the correct locations. We tested on books, DNA, and excerpts from the website. Testing on larger datasets ensures that the algorithm scales.3 We also randomly generated strings to test to attempt to eliminate bias that can arise from handpicking data.

- Benchmarking of your algorithm and a proposed Big O – Using at least five meaningfully distinct dataset sizes of increasing size, submit a figure with labeled axis that shows your benchmarked runtimes for each input dataset. Using this information, your code base, and your knowledge of CS theory, write a detailed justification for the overall Big O of your method. This justification can be a formal proof providing the bounds of your algorithm or rely on a pseudocode overview of your approach. If your approach ultimately did not match the theoretical Big O of your data structure, discuss why.

The pseudocode for makeLPS function is shown below:

```
function makeLPS(pattern):
```

```
len = 0  // length of the prefix to be found (to be put into lps)
lps = new Vector of size pattern.size()
lps[0] = 0  // lps[0] is always 0
pos = 1  // current position in pattern

while pos < pattern.size():  // while not at the end of the pattern
   KMPcount++
   if pattern[pos] == pattern[len]:  // prefix matches suffix
      len++  // increase length of prefix
      lps[pos] = len
      pos++
   else:
      if len != 0:
         len = lps[len-1]  // revert length to last match
         // position is not updated!!
      else:
         lps[pos] = 0
         pos++

return lps
```

The ‘makeLPS’ function is intended to create a vector that tells us how many characters we could skip in the next iteration when a mismatch happens. The vector named ‘LPS’, has integers $A_0, A_1,\ A_2 \ldots A_{m-1}$, where m is the length of the pattern we want to search for. A_k represent the longest proper suffix that is also the prefix of the substring 0 to k of the pattern.

We compute LPS by setting two pointers:

1. ‘pos’, this points to the location in the LPS we are currently computing.
2. ‘len’, this points to length of the longest proper suffix that is also the prefix of the current substring

If the characters at ‘pos’ and ‘len’ are matching, we increasing both pointer. If the characters aren’t matching, we change ‘len’ to the previously calculated LPS for ‘len’-1, which would be the longest proper suffix that is also the prefix in the current suffix we found. Then the iteration would go on until ‘pos’ reach the end of the pattern.

The time complexity would be calculated by considering the algorithm as two part.

The first part is comparing the ‘pos’ and ‘len’ as ‘pos’ iterating through the pattern. Only one comparison will be done for each character, since ‘pos’ will increment after the comparison. So this would be O(m), where m is the length of the pattern.

The second part is to find ‘len’ when mismatch happens. For the base case, first mistach happens at k_0 in the pattern, where k is the integer and $0 \leq k_0 < m$. It would take $k_0 - j_0$ iteration and comparison to find the j_0, where $0 \leq j_0 \leq k_0$ and j_0 is the position of new ‘len’. This is because ‘len’ would decrement at least 1 for every iteration and stop at 0.

For nth mismatch at k_n, we found new ‘len’ at j_n, where $0 \leq j_n \leq k_n$, then it means we need $k_n - j_n$ iteration to find ‘len. For the next mistach k_{n+1}, ‘len’ equals $j_n + k_{n+1} - k_n$ since ‘len’ will increase the same amount when matches happens. So the maximum amount of iteration requires to found new ‘len’ is $j_n + k_{n+1} - k_n - j_{n+1}$. The total maximum amount of comparision would be $k_n - j_n +$ $j_n + k_{n+1} - j_{n+1} - k_n = k_{n+1} - j_{n+1}$. This means the time complexity is also O(m).

The total time complexity for function ‘makeLPS’ would be O(m).

The pseudocode for function ‘search’ is shown below:

```
function search():
   result = new Vector of size_t
   dataPos = 0
   patternPos = 0

   while dataset.size() - dataPos >= pattern.size() - patternPos:
      KMPcount++

      if pattern[patternPos] == dataset[dataPos]:  // character matches
         dataPos++
         patternPos++

         if patternPos == pattern.size():  // if patternPos is at the end of pattern, then
match found
            result.push_back(dataPos - patternPos)
            patternPos = lps[patternPos - 1]
```

```
        else:
            if patternPos != 0:
                patternPos = lps[patternPos - 1]  // go to the last matching position
            else:
                dataPos = dataPos + 1  // no matches; pattern at the beginning, restart

    return result
```

The ‘search’ function returns the positions of the matching pattern within the dataset. Same as the ‘makeLPS’, the ‘search’ function uses two iterators to points toward the positions in the dataset and the pattern. When the mismatch happens, the iterator find the next positon to compare base on the previously calculated ‘LPS’.

The time complexity analysis is the almost the same by considering the fact ‘makeLPS’ is just the same process but pattern and dataset are the same.

First part is iterating through the dataset, and each position will be compared for one time only as it will increment after the comparison. So time complexity will be O(n), where n is the length of the dataset.

For the second part. When first mismatch happened at k, where mistach happens at k_0 in the pattern, where k is the integer and $0 \le k_0 < m$. It would need $k_0 - j_0$ iteration to find the all the position in the pattern it needs to compared to, where j_0 is the last position it needs to compare $0 \le j_0 \le k_0$.

For the next mistach, there are two possibility. One is that it found a match, the iteration for this case is already considered in the first part Then it would start again by resetting the iterator in pattern as 0. The second one is that it found another mismatch before a full match. In this case, we have the previous mismatch at k_n and last positon to compared at j_n, where $0 \le j_n \le k_n < n$, then it means we need $k_n - j_n$ iteration previously. For the next mistach k_{n+1}, it equals $j_n + k_{n+1} - k_n$ since interator in the pattern will increase the same amount when matches happens. So the maximum amount of iteration requires to found last position to compare in the pattern is $j_n + k_{n+1} - k_n - j_{n+1}$. The total maximum of comparision would be $k_n - j_n + j_n + k_{n+1} - k_n - j_{n+1} = k_{n+1} - j_{n+1}$. The time complexity for the second part is O(n).

In total, the time complexity for ‘makeLPS’ and ‘search’ combined is O(m+n).

As all graphs shown, the KMP algorithm did consistently better than the naive solution. However, the pattern length did not seem to have a strong linear relationship with the number of comparison taken, with both R^2 values less than 0.1. This doesn’t fit with our expectation that the number of comparison should be linear compared with length of the pattern. This is mostly because there are less partial matches as the length of the pattern increase. As the length become to long to have matches, the improvement made by skipping characters based on partial matches is consistent but not significant. If the algorithm could only found partial matches for small proportion at the beginning of the pattern, then rest of the pattern won’t affect much of the result, so the length of the pattern doesn’t have a linear relationship with the number of comparisons.

As graph 3 has shown, the lengt of the dataset has a strong linear relationship with the number of comparisons, with both R^2 values equals to 1. This does fit with our expectation that KMP should have a linear and better performance comparing to the naive algorithm, especially as the size of the dataset increases. Because as the dataset gets bigger, the improvement made by skipping characters based on partial matches become more consistent and significant. The large datasets also make the computation of LPS less significant compared to the number of comparisons for iterating through the dataset.