Adarsh Bhandari (adarsh4)
Harbin Li (hdli2)

CS225 EC Project Final Report

The algorithm we have chosen is the KMP matching algorithm. We implement and apply this algorithm to keyword searching in text documents. Below, we've chosen a set of data to illustrate the time complexity of our algorithm which will help us aid in relating our implementation's time complexity to the theoretical time complexity of the KMP data structure.

Datasets

The following datasets are curated from long book texts. The datasets consist of primarily alphabet characters as well as punctuation characters. The first dataset is the exception as it contains ten numerical characters. The datasets have been truncated such that the dataset sizes roughly increase by some order of magnitude or easily recognizable factor.

Dataset Name	Dataset size (n characters)
10 Characters	10
HP	100
Percy	1000
Wonderland	10448
LoTR	101148
Hobbit	506516
Full LoTR	2462922

Algorithm Testing

To test our algorithm, we've split up testing into testing each individual component of our data structure. We've written tests for the preprocessing function that handles the LPS array (preprocessLPS), a test for reading the files as a string of desired format (readFileToString), and most importantly our pattern searching function (searchKMP).

The input pattern matching strings are usually of small length in comparison to the data size, which is why when testing the preprocessing function, we've manually found the corresponding LPS arrays. This is done by finding the LPS array given the following rule for lps:

lps[i] = the longest proper prefix of pattern[0..i] which is also a suffix of pattern[0..i]

Therefore, for the pattern "AAAAA", the lps array will be {0, 1, 2, 3, 4} because the possible length for the proper prefix increases for each index. The pattern "ABCDEA" must have "A" as the first character of the prefix, and this is only present at index 5, and the prefix will have length one. Therefore the lps for this pattern is {0, 0, 0, 0, 0, 1}. These specific tests can be checked and calculated by the preprocessLPS function in test cases "PreprocessLPS Test1" and "PreprocessLPS Test2", but more comprehensive tests are also given.

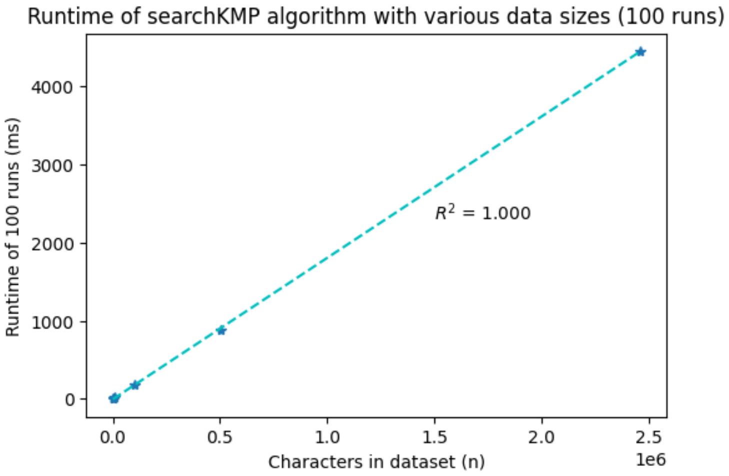
The preprocessLPS function is only a helper function within the larger searchKMP algorithm. The searchKMP algorithm is much harder to do by hand for larger datasets, so instead of manually calculating the result, we've used the standard library std::find to find all indices where our desired pattern string appears. This vector of indices is then checked against the one that is produced by searchKMP algorithm. These tests are labeled with “SearchKMP Test1” and more. For each of our datasets of various magnitudes, we have a test case to calculate how many matches there were and the locations of the matches. These tests exhaust our provided datasets, which allows us to be confident in our result for each of our datasets.

Benchmarking

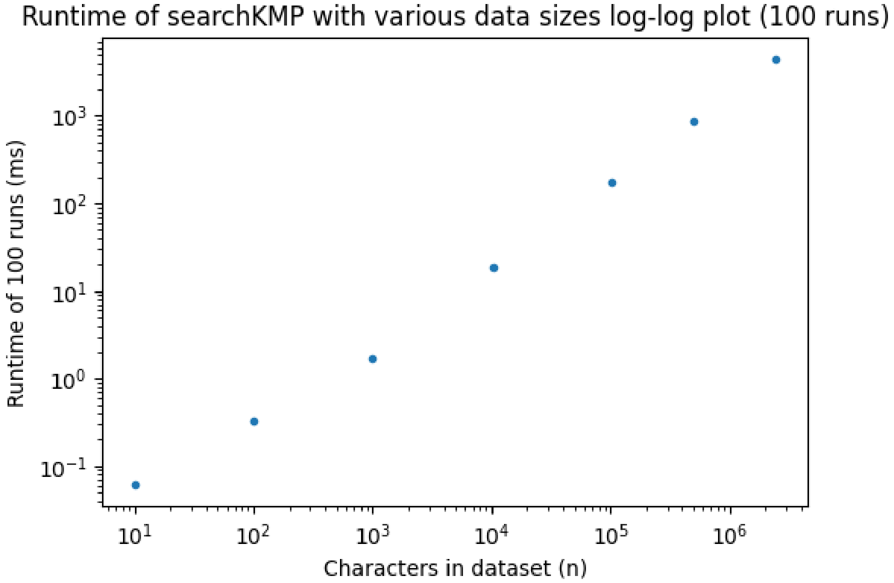
Using the datasets, the runtimes are calculated with an input string of size three (m=3) and the algorithm is run for each dataset. Since m < n for all datasets, the searchKMP algorithm can be tested on all the datasets. The runtime is the sum of running the pattern matching searchKMP algorithm 100 times. Refer to our main.cpp, or follow the README “Running Instructions” to examine our methods and replicate our results for benchmarking the searchKMP algorithm.

Dataset Name	Dataset size (n characters)	Runtime (100 runs; in µs)
10 Characters	10	61
HP	100	335
Percy	1000	1682
Wonderland	10448	18617
LoTR	101148	172807
Hobbit	506516	885401
Full LoTR	2462922	4455458

Taking the dataset size and runtime pairs to produce a graph results in the following. In addition to the pairs, a linear regression of the data pairs has been plotted, with a corresponding R^2 value that is extremely close to 1, which gives us more confidence that our algorithm has a linear time complexity.



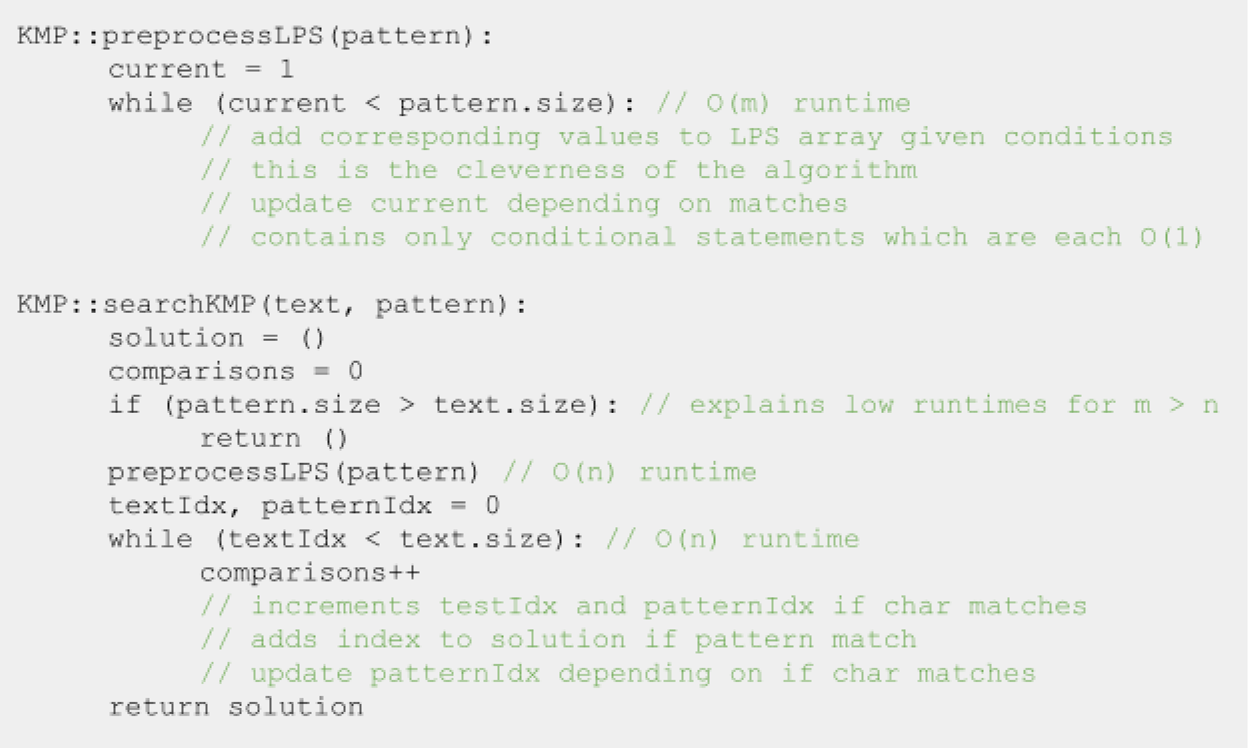
The previous data pairs have also been plotted on a log-log plot to show the linear nature of the algorithm even at various magnitudes of data sizes.



Lastly, we'd like to test whether the runtimes and time complexity of the algorithm was dependent on the length m of the pattern matching string. Times were recorded for an input string of length m=3,000. The ratio of the runtimes for m=3,000 and m=3 in the first three rows are omitted due to m=3,000 > n for those select runs. Therefore, the runtimes for those are trivial to deal with.

Dataset Name	Dataset size (n characters)	Runtime (100 runs; in µs)	RT(m=3,000) / RT(m=3)
10 Characters	10	24	
HP	100	24	
Percy	1000	20	
Wonderland	10448	27042	1.45
LoTR	101148	177210	1.03
Hobbit	506516	884518	0.99
Full LoTR	2462922	4426617	0.99

As shown in the table, the ratio for the runtimes with varying pattern string lengths are close to one for m << n. However, the algorithm seems to stray away from its linearity when pattern string lengths are a large fraction of the dataset size. We'll now run through the algorithm to illustrate our implementation via a simplified skeleton of our implementation using pseudocode, and use this as a means to calculate our actual time complexity.



```
KMP::preprocessLPS(pattern):
      current = 1
      while (current < pattern.size): // O(m) runtime
            // add corresponding values to LPS array given conditions
            // this is the cleverness of the algorithm
            // update current depending on matches
            // contains only conditional statements which are each O(1)

KMP::searchKMP(text, pattern):
      solution = ()
      comparisons = 0
      if (pattern.size > text.size): // explains low runtimes for m > n
            return ()
      preprocessLPS(pattern) // O(n) runtime
      textIdx, patternIdx = 0
      while (textIdx < text.size): // O(n) runtime
            comparisons++
            // increments testIdx and patternIdx if char matches
            // adds index to solution if pattern match
            // update patternIdx depending on if char matches
      return solution
```

As outlined by the pseudocode, the preprocessLPS function has an O(m) runtime to create the LPS vector that is used for the m length pattern matching string. This function is used within the searchKMP algorithm. The searchKMP algorithm consists of this O(m) function, as well as its own while loop that has a runtime of O(n) depending on the n length text size. Therefore, the searchKMP algorithm has a runtime of O(n+m). For most cases, m << n, so the time complexity can be simplified to O(n), but for cases such as using a pattern matching string of length m=3,000 on a dataset of size n=10,000, the time complexity is then affected by the pattern matching string length.

This analysis of the pseudocode aligns with our observations from graphing the runtime data pairs, which concludes that our KMP algorithm does indeed have O(n) time complexity. This reflects the time complexity of the theoretical KMP data structure for the most part, and only shows certain practical limitations when using values of m that are less than but close to n.