Substring Search

By:

Luke Deratzou,

Aditya Kumar,

Isabel Morales Sirgo,

Pooja Patel,

Garrett Sheehan

Table of Contents

Table of Figures

Introduction: What is Substring Search?

The approach to conducting a substring search given a text of string length N and a pattern string of length M to find an occurrence of the pattern within the text has numerous solutions. One of these solutions is the Knuth-Morris-Pratt (KMP) algorithm. This algorithm was conceived in 1974 by Donald Knuth and Vaughan Pratt and independently by James Morris. The three then published the paper, Fast Pattern Matching in Strings, which discussed the KMP algorithm together in 1977 in the *SIAM Journal of Computing*. [1] This algorithm was produced from the given problem:

> Given a string N, the problem of string-matching deals with finding whether a pattern M occurs in N and if M does occur then returning the position in N where M occurs.

The brute force approach to a substring search would have the order of O(M*N) which is considered a slow running and inefficient algorithm. This approach is slow because the elements of pattern M that had comparisons performed earlier are continually redone. Moreover, when a mismatch is detected for the first time the place of M would only move one position to the right and the search would continue. Overall, the brute force a solution to the substring search involves a large degree of repetition and comparisons. The brute force solution is illustrated below.

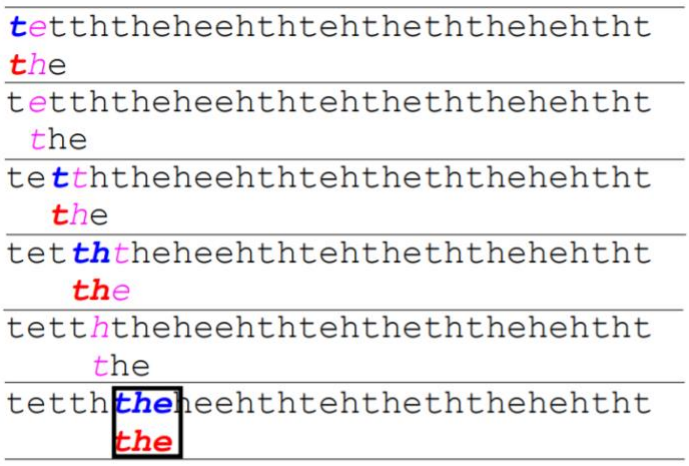

Figure 1: Sample Brute Force Substring Search [2]

Substring searching algorithms have multiple applications including digital libraries, screen scrapers, word processors, web search engines, spam filters, natural language processing, and the DNA alphabet.

Objective 1: Original Publication of KMP

The KMP algorithm originated in a paper published by the SIAM Journal on Computing in 1977. The paper was titled *Fast Pattern Matching in Strings* and authored by Donald E. Knuth, James H. Morris Jr. And Vaughan R. Pratt. The paper describes KMP pattern matching algorithm's development, programming, efficiency, extensions, theoretical considerations, palindromes, historical remarks, and a previous method of pattern matching. [1]

Objective 2: KMP Implementation

The KMP algorithm compares the pattern to the text in left-to-right order but shifts the pattern more efficiently than the brute force algorithm. The KMP algorithm performs at O(M+N). The KMP algorithm is composed of two components, a prefix function and a KMP matcher. The prefix function includes information about how the pattern matches against shifts of itself, this avoids the backtracking within the string N. The KMP matcher functions to find the occurrence of M in N and returns the number of shifts of M after it has occurred. To avoid redundant comparisons the algorithm shifts the pattern by the largest prefix of M [0...j-1] that is a suffix of M [1...j]. The KMP solution is illustrated below.

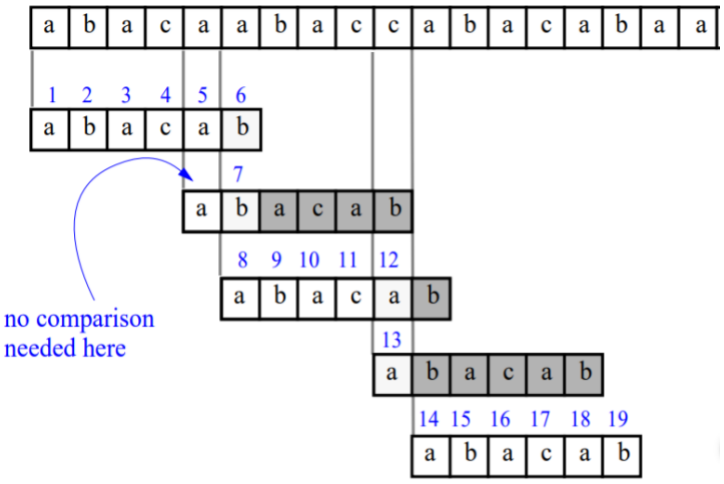

Figure 2: Sample KMP Substring Search [2]

The KMP algorithm running time is proportional to the time needed to read the characters in the text and pattern.

Objective 3: Worst-Case Scenario

In what scenario will the KMP algorithm work its hardest?

First, let us recognize the strengths of the algorithm and then think of a scenario where that strength will be not be helpful. The main strength of the KMP algorithm is its ability to analyze the reoccurring characters in a given pattern and be able to save time when searching text using that knowledge.

Now in what scenario will that knowledge not be helpful?

The first scenario that comes to mind is a large alphabet. A large alphabet would theoretically make KMP less effective since there is a lowered chance of matching a pattern in the text and therefore would force the algorithm to perform more array inspections and comparisons.

Another scenario where KMP would not be helpful is if the text being analyzed contained unique substrings that were the same length as the desired pattern. This is another way to ensure that KMP will struggle to find matches as the desired pattern would be difficult to find.

These scenario sound very similar to the De Bruijn Sequence, a sequence made up of characters that has the following properties:

- Consist entirely of character in an alphabet the size of **k**
- The characters in the alphabet are used to create substrings that are the length of **n**
- Each instance of a substring occurs only once through the entire sequence.

Take this De-Bruijn Sequence for example (k = 2 [0 and 1 are the only characters] ; n = 2):

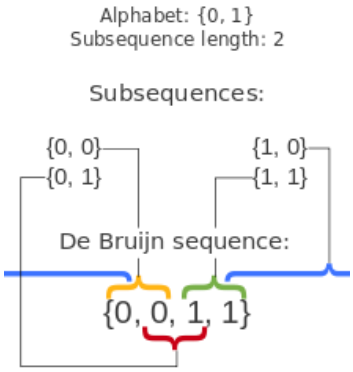

Figure 3: Sample De Bruijn Sequence

Notice how the substrings {0,0} , {0,1} , {1,0} , and {1,0} only occur once throughout the entire sequence.

Imagine using a De Bruijn sequence as the text KMP is analyzing that has a large alphabet and the desired pattern an arbitrary unique substring located in the sequence. KMP will struggle to find any matches to the desired pattern due to the large alphabet and substrings being entirely unique. KMP's behavior would be very similar to a brute-force substring search algorithm with this text and pattern combination. In this case, no amount of prefix and suffix analysis on the pattern will help in a text and pattern string.

Overall, the following properties should make KMP work its hardest:

- The Text is a De Bruijn sequence where the alphabet is large.
- The Pattern an arbitrary unique substring located in the text's De Bruijn Sequence.

As for a sample text and pattern to demonstrate the KMP algorithm working its hardest, I will provide a De Bruijn sequence with a reduced alphabet rather than a large alphabet as the sequence grows very rapidly as you increase the alphabet size. You can easily do it out by hand and observe why KMP is ineffective with this text and pattern set:

Text: A A A B A A C A B B A B C A C B A C C B B B C B C C C A A

Pattern: B A C

Notice that the text is a De Bruijn Sequence where k = 3 {A, B, C} and n = 3 while the pattern an arbitrary unique substring in the sequence.

Objective 4: Evaluate Performance of the Algorithm

NOTE: All relevant data can be found in this spreadsheet: https://wpi0-my.sharepoint.com/:x:/g/personal/lderatzou_wpi_edu/EcZ04bR2TUNNmIE9_f22FOIBEYXCWkuy3koFppGZ7VuEgw?e=jMzJuf

With a time complexity of O(N+M)[2], KMP is a reliable and consistent option when it comes to efficiently performing substring search. The number of comparisons it executes on a typical search is not affected by alphabet, though time is slightly inconsistent.

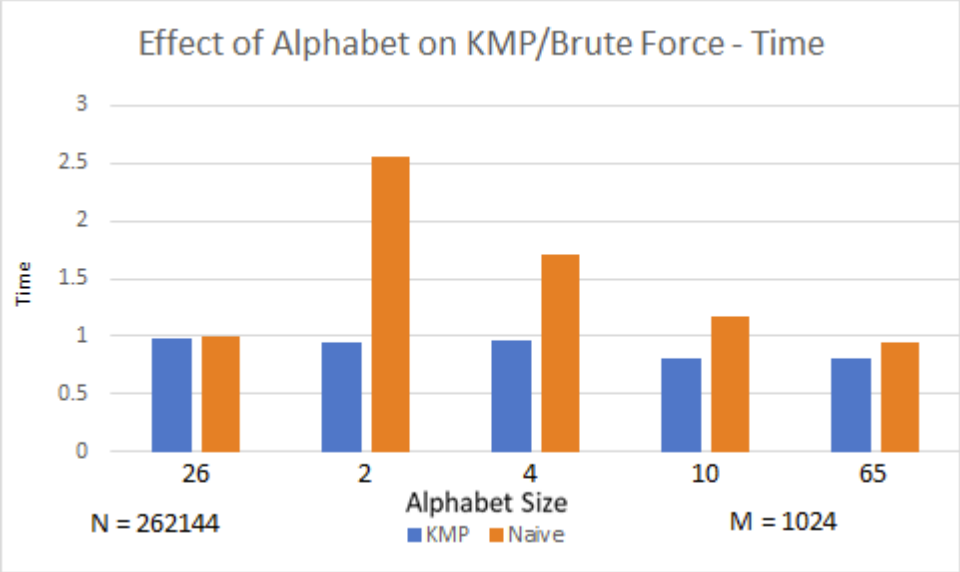

Figure 4: Effect of Alphabet on KMP/Brute Force-Time

KMP's search time increases the greater the substring size is due to a larger substring size causing KMP's strategy of applying the prefix to take longer. As a result, KMP is not at its most optimal when faced with a substring size that is close to the size of the pattern.

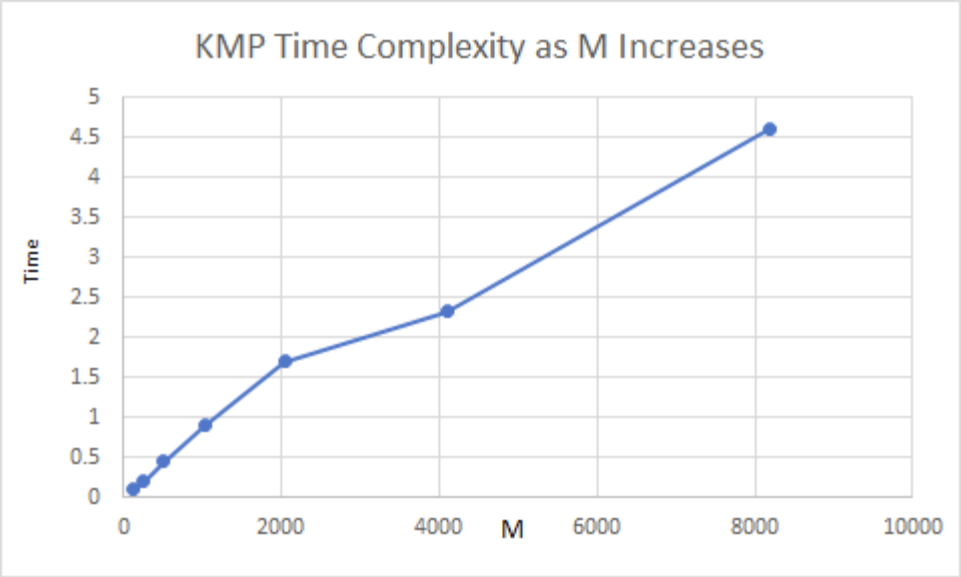

Figure 5: KMP Time Complexity as M Increases

Objective 5: Evaluate Performance of Brute Force Algorithm

With a best-case time complexity of O(M)[2] and a worst-case time complexity of O(MN), [2] performance varies drastically depending on how favorable the pattern is to the brute-force algorithm. Unlike KMP, the brute force method's efficiency is reliant on the size of the alphabet the pattern uses. As seen in the graph below, brute force works well with higher alphabet sizes, but its performance suffers when presented with a low alphabet size.

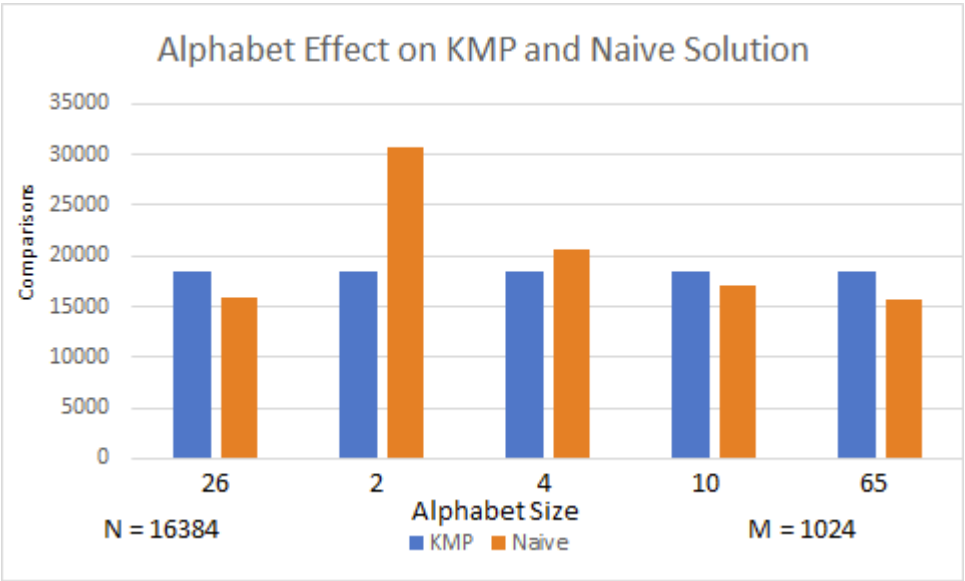

Figure 6: Alphabet Effect on KMP and Naïve Solution

This behavior is most evident when brute-force is provided with a binary alphabet, as binary substring searches take considerably more comparisons than searches with any of the other alphabets it is provided. Another significant performance factor for the brute-force algorithm is how close the substring is to the pattern in size. As seen in the graph below, the brute force algorithm's comparisons decrease as M approaches N, making its performance solid when dealing with a large substring size.

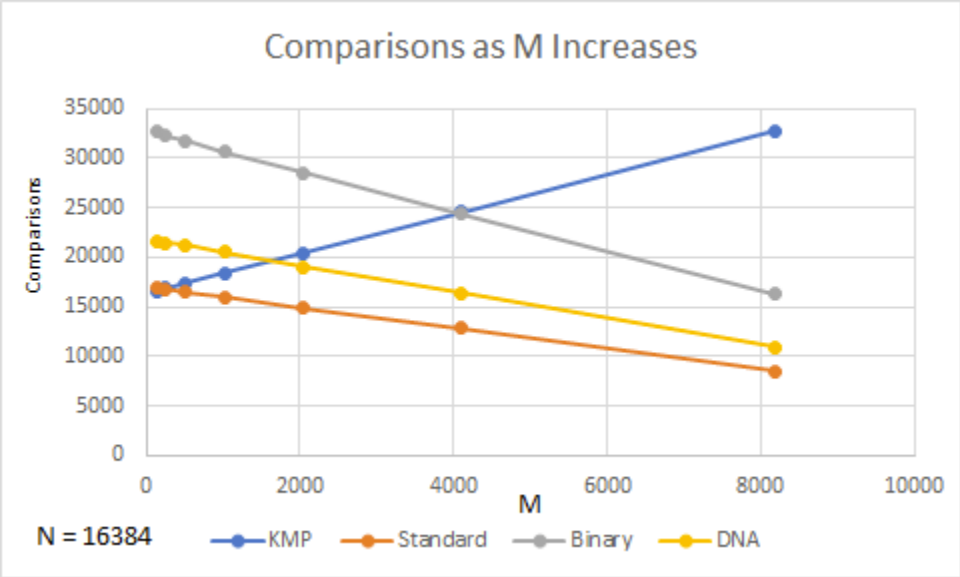

Figure 7: Comparisons as M Increases

On the flip side, the comparisons of brute-force increase considerably when the difference between M and N is higher, and a low alphabet size accentuates this problem.

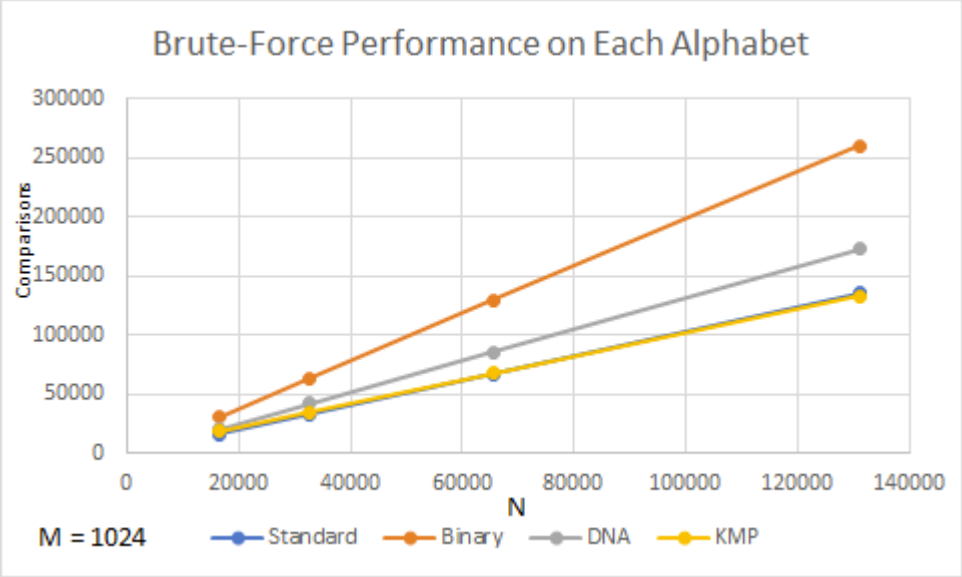

Figure 8: Brute-Force Performance on Each Alphabet Comparisons

Much of these observations on comparisons can be applied to time as well, as the amount of time it takes for brute-force to find the substring increases with the difference in size between N and M and the size of the alphabet.

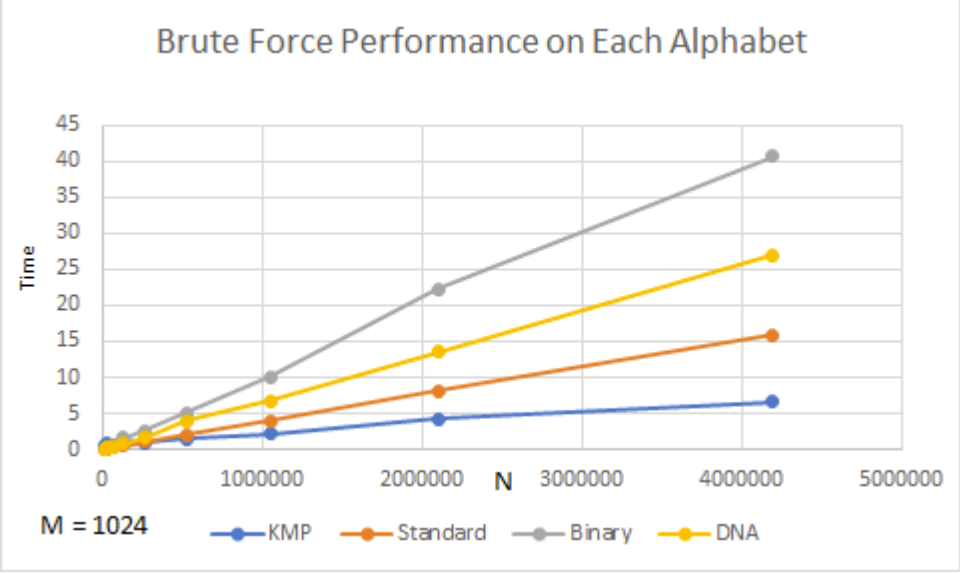

Figure 9: Brute-Force Performance on Each Alphabet Time

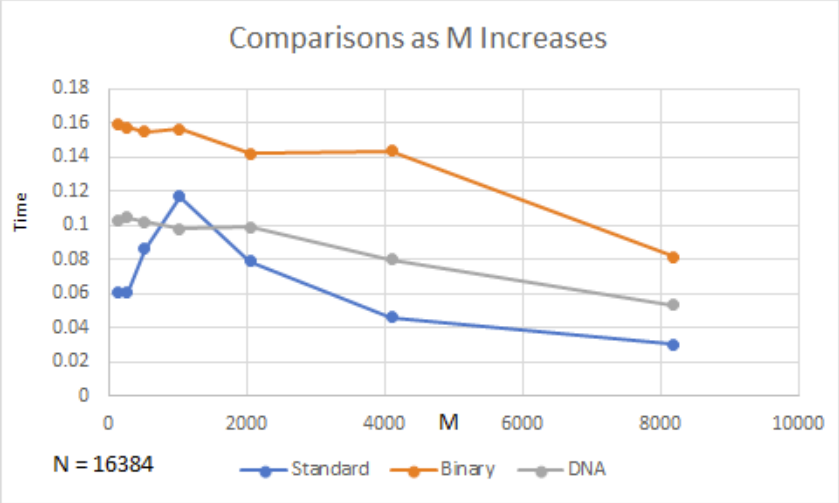

Figure 10: Comparisons as M Increases

Objective 6: When does KMP outperform the brute force algorithm

KMP's ability to outperform the brute force algorithm depends on the size of the alphabet used and the size of the string in comparison to the size of the pattern. A greater alphabet size reduces the amount of repetitions spent looking for a match, with the naïve solution faring far better in the full alphabet than in the DNA alphabet and especially the binary alphabet. As seen in the graph below, the brute-force algorithm with the standard alphabet outperforms KMP at the beginning due to the higher alphabet size working in its favor.

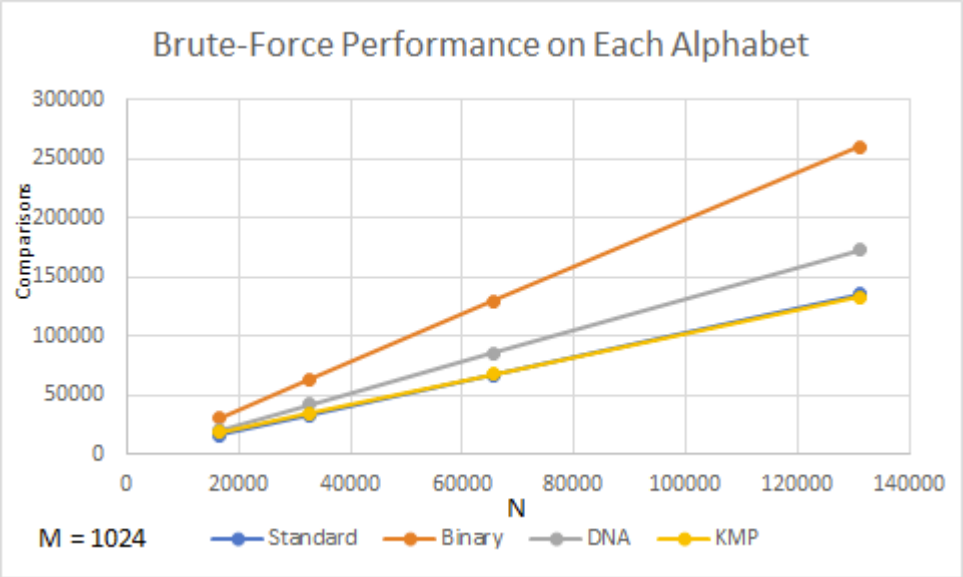

Figure 8: Brute-Force Performance on Each Alphabet Comparisons

Although the brute force method can overtake KMP initially with the help of a higher alphabet size, KMP eventually becomes the clear winner as the pattern size and the difference between the substring size and pattern size increases.

References

[1] Donald E. Knuth, James H. Morris, Jr., Vaughan R. Pratt, Fast Pattern Matching (SIAM Journal on Computing, June 1977), https://pdfs.semanticscholar.org/4479/9559a1067e06b5a6bf052f8f10637707928f.pdf (Accessed: 4/29/2020)

[2] Ananth Grama, Strings and Pattern Matching (Purdue University, n.d.) https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap11.pdf Accessed: 4/29/2020)