Implementation and Computational Analysis of String Searching Algorithms

Raechel Griffin David Perrone

Table of Contents

Abstract

A string search algorithm is a method of searching for a pattern in a larger text. This is useful today because they can be used to filter through vast amounts of data to find where a certain topic is mentioned, or if it is mentioned at all. The purpose of this project is to implement and compare the performance of the Rabin-Karp string search algorithm with Boyer-Moore. A brute force algorithm is used as a baseline because it is the most basic form of a string search.

m – length of pattern

n – length of text

i -- current position in text

Window of text – substring of the text from position i to i + m

To better understand the Rabin-Karp and Boyer-Moore algorithms, it is important to begin with a description of the brute force algorithm. A brute force search algorithm checks every character in a text between 0 and n-m "whether an occurrence of the pattern starts there or not" (Charras & Lecroq, 2004). If the first character of the pattern matches the first character in the window of text, it compares the next character of each until a mismatch occurs or it reaches the end of the window and determines the strings are equal. Brute force search has a worst-case runtime of O(nm). The biggest flaws that Boyer-Moore and Rabin-Karp address are that brute force always slides one position to the right and that it can run into collisions by comparing each individual letter of a string.

The Rabin-Karp algorithm attempts to address the latter of those two flaws by checking if the contents of a window of text look like the pattern rather than checking each position of the text for the pattern (Charras & Lecroq, 2004). It does this by using hash values to represent the strings. The hash values are less likely to cause collisions, but they still need to be manually checked if they appear to be equal. The best case runtime for Rabin-Karp is O(n + m) but still has a worst-case runtime of O(nm).

The Boyer-Moore algorithm, considered as the most efficient string-matching algorithm in usual applications (Charras & Lecroq, 2004) improves upon the brute force algorithm by "performing comparisons from right to left...and using two precomputed functions to shift the window to the right"(Charras and Lecroq, 2004). To briefly summarize, take a pattern of length m and a text of length n, compare pattern[m-1] to text[i + m-1] in the text. In case of a match, the window of text slides one character to the left and pattern[m-2] is compared to text[i+m-2]. This is repeated until the first character of the pattern is reached or until a mismatch occurs. In case of a mismatch, the window of text is shifted x characters to the right. That value x is

computed from the aforementioned functions, referred to as the “good-suffix shift” and the “bad-character shift”(Charras and Lecroq, 2004).

Methods

The Rabin-Karp search algorithm involves a pre-processing phase where the initial hash values of the pattern and text window are calculated. The basic idea is to compute a hash of pattern[0…m-1] and of text[0…m-1] (Sedgewick & Wayne, 2011). If the hash values match, it is still necessary to manually check if the strings are equal. According to Lecroc and Charras, the hash function should be efficiently computable, highly discriminating for string, and hash(text[i + 1…i + 1 + m]) must be easily computable from hash(text[i…i+m]. The key to this last criterion is to "update a rolling hash function in constant time" (Sedgewick & Wayne, 2011). There is no standard way to implement this hash function and many variations of it can be found online. In fact, in Rabin and Karp's paper where they discuss the algorithm, they "select the fingerprint function at random from a family of easy-to-compute functions."

The Boyer-Moore algorithm employs two different shift functions to determine how far the text window should slide. Boyer-Moore has time complexity $\Theta(m)$ preprocessing + $O(mn)$ matching in the worst case and $\Theta(m)$ preprocessing + $\Omega(n/m)$ matching for the best case. The more simple of the two functions is the "bad character shift." The idea behind this function is that if pattern[i] is compared to text[i], and there is a mismatch, the text window can shift to the right until the rightmost occurrence of text[i] in pattern.

In order to properly compare the performance of the three algorithms described above, two tools were written (classes in C++). One tool was a timer that was used to measure the runtime (in ms) of each algorithm. The other, called CompTool was used to count the number of comparisons and collisions made during the tests.

The timer class uses steady_clock found in the std::chrono library to measure the runtime. It is implemented as a class in C++, called timer. The two main variables used in the class are steady_clock time_points: m_startTimePoint and m_endTimePoint. They are used to represent a fixed point in time, much like a stopwatch. A function called Start() sets m_startTimePoint equal to the current point in time. Later on, a function called Stop() sets m_endTimePoint to the current point in time. The duration is then set equal to m_endTimePoint – m_startTimePoint. The default duration value is nanoseconds, which are then casted into milliseconds to make the measurements easier to comprehend.

The comparison tool is also a class in C++, called CompTool. It is used to count the outer comparison, inner comparisons, total comparisons, false positives, and number of patterns found. Both tools have functions called WriteHeader and WriteCSV. The header function writes the names of the variables at the top of the csv file. The WriteCSV function then appends the stored benchmarking data onto the files. The function is specifically using the std::ios::app parameter so that each benchmarking tool can be used several times and write to the same file without overriding any of the previous data.

Some preprocessing was required before the search algorithms could be used. The strings that were used in the tests were collected as books from Project Gutenberg. If we wanted to search a book for a specific pattern, we needed to be careful to ensure that the entire book was treated as one long string rather than as hundreds of lines. In the case of Rabin-Karp, a large part of the trick is to keep a rolling hash from one character to the next and only calculate a full hash value once at the beginning of the search. This could not be done if we read the lines straight from the file and searched them one by one. It would result in us calling the original hash function at the start of every line. To avoid this, we implemented two functions: concatStr and Strip.

The concatStr function was used to read line by line in the file and append each line onto one string. The result was one incredibly long string, rather than several hundred individual strings. The strip function was based on the function in python that removes any leading any trailing whitespace from a line. Not all of the books from Project Gutenberg were formatted the same way. As a result, the following two lines would result in one poorly formatted line:

“ this is leading whitespace”

“this is trailing whitespace “

“ this is leading whitespacethis is trailing whitespace “

The above string has several spaces before and after its body, which would result in extra comparisons being made. It also lacks a space in between each line. The strip function would remove any leading and trailing white space and then a single space would be appended onto each line before finally being appended to the string. This is the result:

“this is leading whitespace this is trailing whitespace “

Implementation

For the Rabin Karp algorithm, the equation used to compute the hash in this report is as follows:

```
h = (int)text[0];
for (int i = 1; i < keyLen; i++) {
    h = (h * b + (text[i])) % q;
}
h = h % q;
```

In this equation, h is the accumulator variable for the hash of a string. Variable b is the base value, which is used to prevent collisions by factoring in the order of the letters.

The function shown above avoids using the power function to increase performance, but the basic idea is as follows:

$$1 + 2 + 3 == 3 + 2 + 1, \text{ but}$$

$$1 * 2^2 + 2 * 2^1 + 3 * 2^0 != 3 * 2^2 + 2 * 2^1 + 1 * 2^0$$

Once again the values for b and q are not uniform amongst Rabin-Karp implementations. 256 is used as the value for b in this implementation because characters take up 2^8 bits of memory and b is usually set equal to the size of the alphabet being used. q is any large prime number that is used to prevent the hash value from becoming too large. q needs to be sufficiently large such that it does not result in collisions but if it is too large, it could slow down the program or even result in overflow. (Sedgewick & Wayne, 2011) Some implementations of Rabin-Karp even went so far as to use a function to randomly compute the value of q. This implementation used a value of 293, which was kept consistent for benchmarking purposes.

Rolling the hash:

```
subHash = ((subHash + q - ((int)m_string[i] * power_val) % q) * b + (int)m_string[i + keyLen]) % q;
```

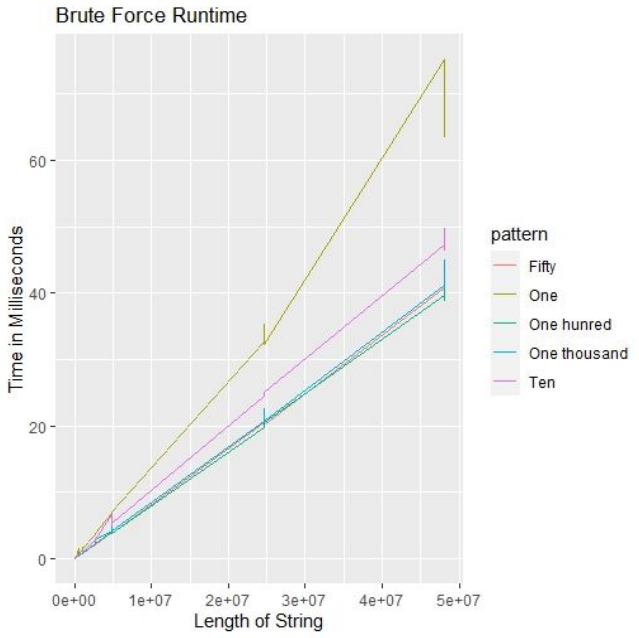
In the first half of the equation, the <value of the first character> * 2^{m-1} is subtracted from the hash and the value of q is added on, to prevent underflow, or negative numbers. The entire hash is then multiplied by b to shift the numbers one decimal place to the left. The hash value of the next character is then added on and the whole hash value is modded by q.

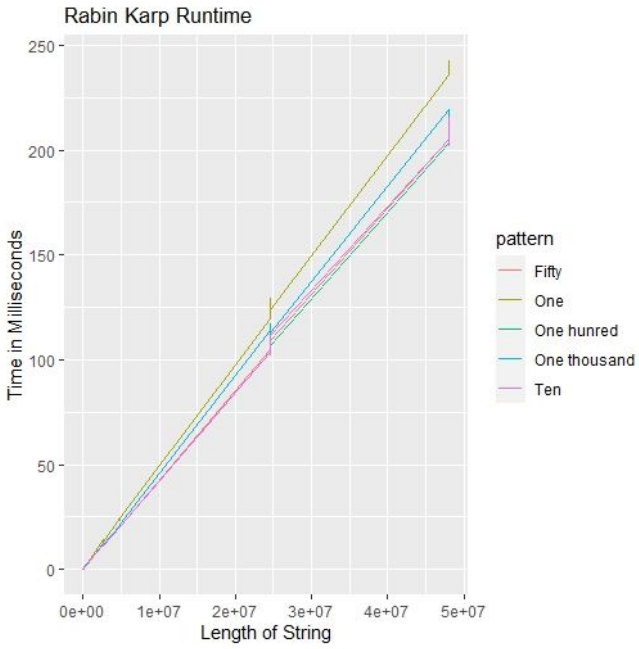
In a simplified example where our current string value is 123 and the next value is 4, using base 10:

$$123 - 10^2 = 23; \qquad 23 * 10 = 230; \qquad 230 + 4 = 234$$

Rabin-Karp string search has a best case runtime of O(n+m) and a worst case runtime of O(nm) (Charras & Lecroq, 2004). The best case is if there are no collisions and the worst case being if every comparison results in a collision. These are the same exact runtimes as brute-force search, however the number of collisions is drastically reduced by using the hash values.

Each algorithm took more time to search as the length of the string increased.The following graphs[1] show the run time in milliseconds versus the string size factored for pattern length for brute force and Rabin-Karp.



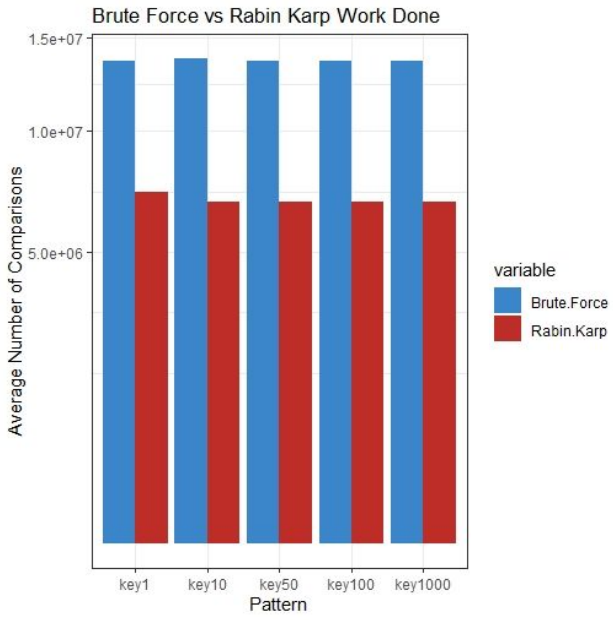


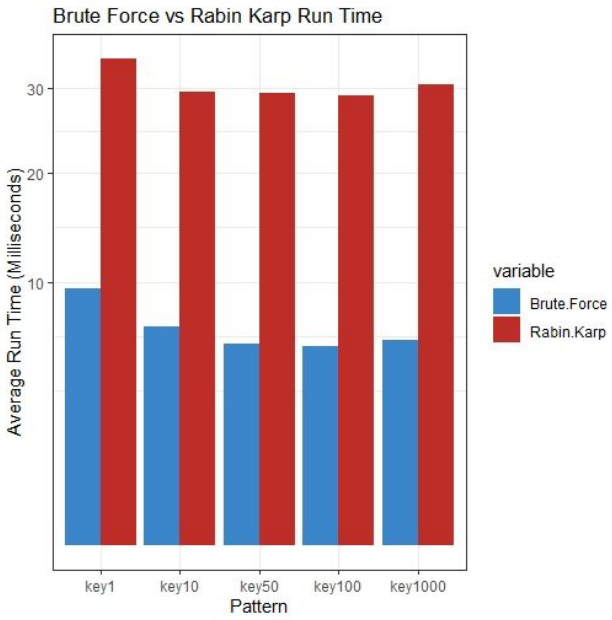
In theory, computing the initial hash of the text and pattern costs O(m), however our hash function uses three operators: multiplication, addition, and modulus, plus one more modulus outside of the loop. So, in practice the computational cost is Big-Theta(3m +1) * 2 (once for text and once for pattern). The rolling hash function, in theory is supposed to be constant time, the first digit is subtracted, and the next digit is added on: O(2). However, in order to compute the hash function, we use 8 operators: Big-Theta(8). To summarize, the average and worst case runtime of Rabin-Karp (in theory) is O(n+m) and O(nm), respectively. However, in practice it is: Big-Theta((3m+1)* 2 + 8n +m) on average and Big-Theta((3m+1)*2 + 8nm) in the worst-case.

The brute-force algorithm will perform more comparisons, but its computational cost is limited to the if statements in the while loops. So each loop uses only one operator and its computational cost is the same as its theoretical: Average Theta(n+m), worst case : Theta(nm).

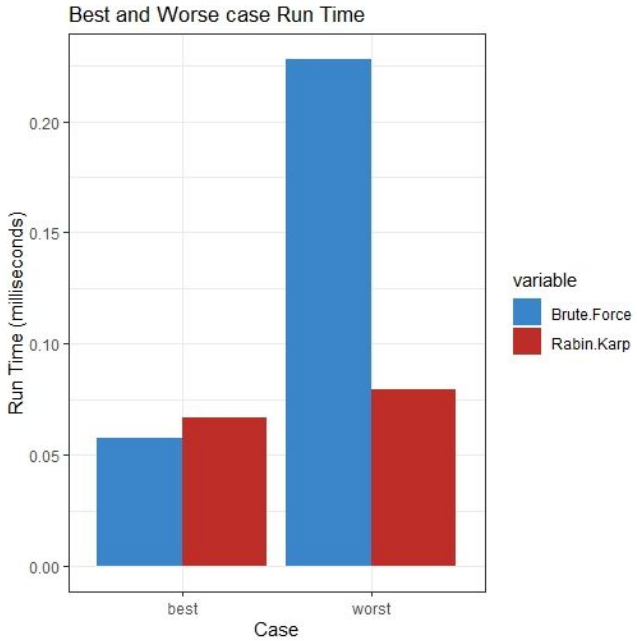
[1] All graphs have a transformed y-axis using a continuous square root transformation. This was necessary as the numerical difference in data was staggering. This transform inflates smaller numbers but stabilises bigger ones.

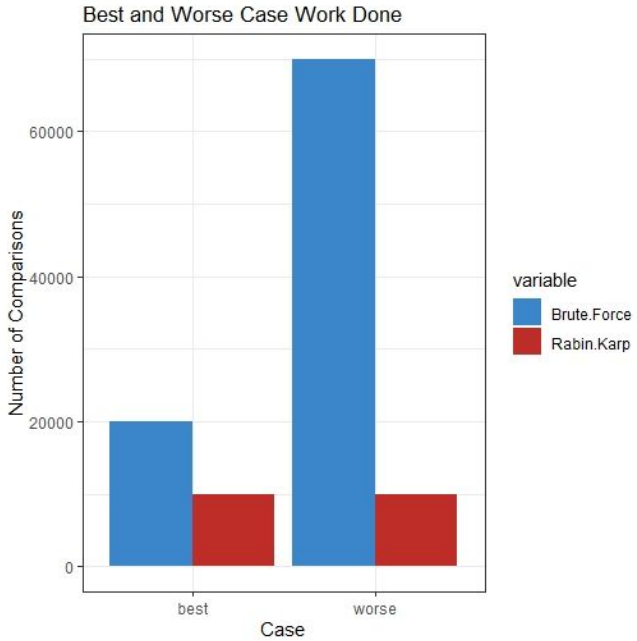
As a result, Rabin-Karp only outperforms brute force in time when brute force is in its worst case scenario of having multiple false positives. The following graphs depict the number of comparisons and average runtime of brute force compared to Rabin Karp for each of the patterns.





On average Rabin Karp makes 49.43% less comparisons than brute force. Even in brute force's best case Rabin Karp makes 50.01% less comparisons than brute force. In brute force's worse case scenario, Rabin Karp excels further making 85.71% less comparisons.





In brute force's best case Rabin Karp only runs 13% slower than brute force. But in Brute forces worst case scenario, Rabin Karp runs 67.46% faster demonstrating that the collisons brute force runs into in its worst case are significantly reduced by using a more efficient Rabin Karp algorithm.

On average, it is better to use brute force search than it is to use Rabin-Karp. According to research by Dr. Lovis and Baud on several string search algorithms within the context of medical text, where "many words have similar suffixes or prefixes" (Lovis, Baud 2000). They implemented the Rabin-Karp algorithm, also noting that "several different ways to perform the hashing function have been published" and came to the conclusion that "the cost of computing the hashing function outweighs the advantage of performing fewer symbol comparisons, at least for common medical language" (Lovis, Baud 2000). In addition to this, another research paper found "it has repeating in checking of characters and slow pattern shifting, which required more consumed searching time in long pattern use"(AbdulRazzaq A., A., Rashid A., A. N., Hasan A., A., & Abu-Hashem, A. M. (2013)).

As mentioned above,the Boyer-Moore algorithm employs two different shift functions to determine how far the text window should slide. This shift is determined by a prec-computed map as follows:

```
for(int i = 0; i < key.length(); i++){
    badCharacterTable[key[i]] = key.length() - 1- i;
}

badCharacterTable[key[key.length()-1]] = 1;
```

It is essentially a map with numbers starting from m and decreasing by 1 each time. If the mismatched character from the text does not appear in the pattern, the window can be safely shifted to the right by m. If a character appears more than once, the rightmost occurrence is used.

Here is an excerpt from Boyer and Moore's paper, "A Fast String Searching Algorithm":

```
Pat:        AT-THAT
Text:       WHICH-FINALLY-HALTS.--AT-THAT-POINT
                                                    Bad char Table:
Pat:                AT-THAT                             A : 1
Text:       WHICH-FINA-AT-THAT-POINT                    T : 0
                                                        - : 4
Pat:                    AT-THAT                         H : 2
Text:       WHICH-FINA-AT-THAT-POINT                Other : 7
```

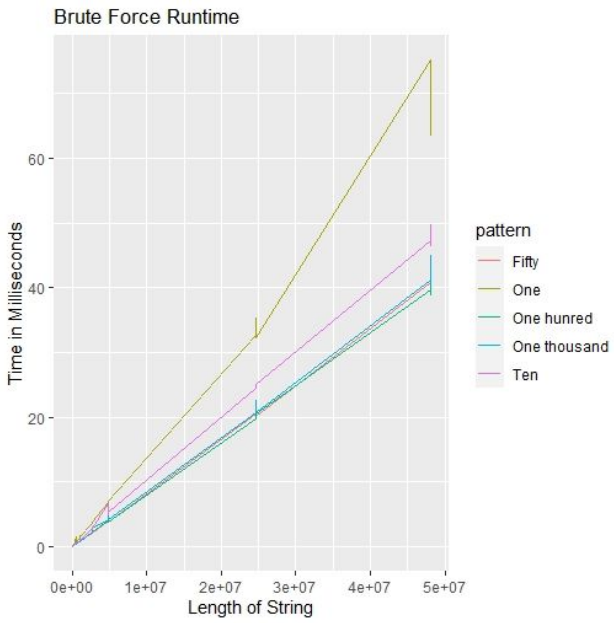
The good character table(s) comes into play when a potential match leads to a mismatch. The first part of the Good Suffix Rule is to find "the biggest index j, such that j < m and prefix[i...j] contains suffix P[i...m] but not suffix P[i-1...m]."(Krishnan, 2015). Put simply, find the starting index of the longest (suffix) of pattern that matches a prefix of pattern. Here is the code:

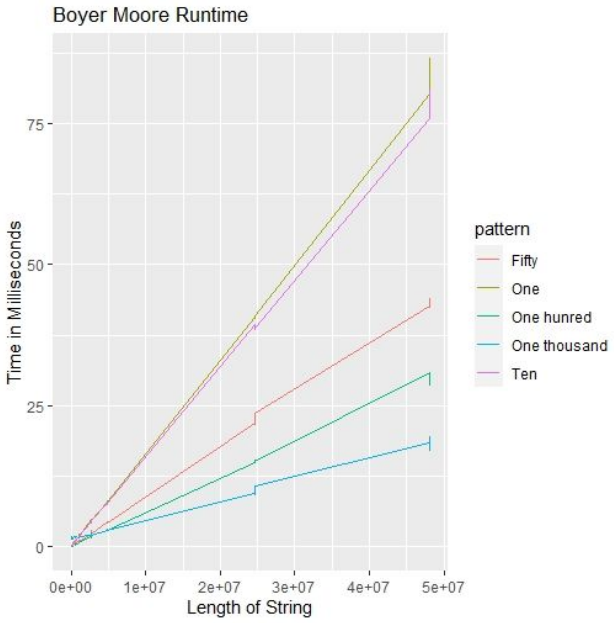
```
// for every substring starting at back of key
for(int i = 1; i < key.length(); i++){
    //find the last occurence of substring before its matching suffix
    //make suffix and prefix substrings
    suffix = key.substr(key.length() - i, key.length());
    prefix = key.substr(0, key.length() - 1);
    //store value as shift
    shift = prefix.rfind(suffix);
    //while suffix found and out of bounds and suffix bad keep looking
    while(shift != std::string::npos && (shift - 1 < 0 || key[shift - 1] == key[key.length() - i - 1])){
        prefix = key.substr(0, shift + suffix.length() - 1);
        shift = prefix.rfind(suffix);
    }

    //if we didnt find one
    if(shift == std::string::npos){
        shift = 0;
    }
    //if we found one
    else {
        shift = shift + suffix.length() - 1;
    }
    //push into array for corresponding i (go where the mismatch is)
    goodArrayL[key.length() - i] = shift;
}
```

Part 1 can lead to some indexes in the table not having any shift value because they do not meet the specifications for it. Part 2 fills in the holes left in the table. The rule is as follows: "l(i) is the longest suffix of P[i...m] that is also a prefix of P]. In the case where part1 has a shift value of 0, this table is used instead. Here is the code:

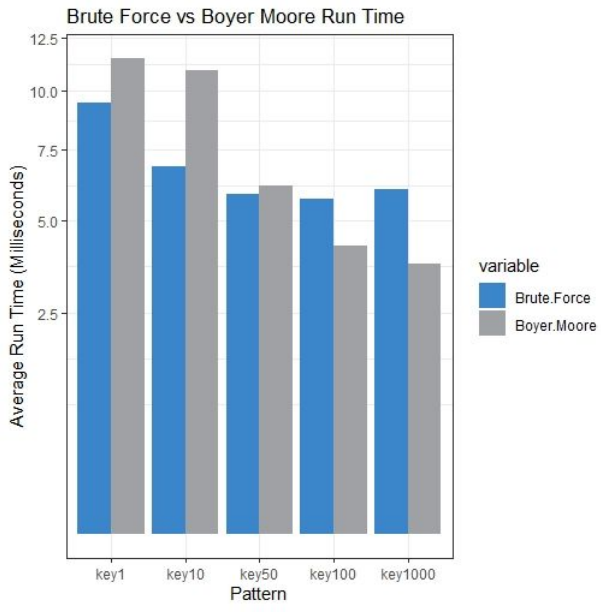
```
//for every substring starting at back of key
for(int i = 1; i < key.length(); i++){
    //find the last occurence of substring before its matching suffix
    //make suffix and prefix substrings
    suffix = key.substr(key.length() - i, key.length());
    prefix = key.substr(0, i);

    //store value as shift
    //length of longest suffix that is prefix if one exists otherwise H[i] = 0
    if(suffix == prefix){
        shift = suffix.length();
    }
    //no match found use previous longest suffix
    else{
        //for suffix length one longest found is 0
        if(i == 1){
            shift = 0;
        }
        //else longest suffix is to the right
        else{
            shift = goodArrayH[key.length() - i + 1];
        }
    }
    //push into array for corresponding i
    goodArrayH[key.length() - i] = shift;
}

//for the 0 case ie a match
goodArrayH[0] = goodArrayH[1];
```

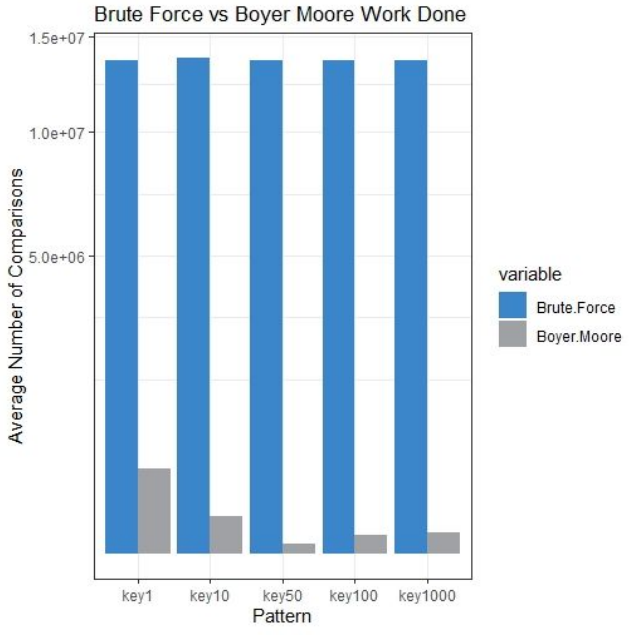
Similarly to brute force and Rabin Karp, Boyer-Moore's run time positively correlates to string length. The longer the string the longer each algorithm took. The following graphs show the run time in milliseconds versus the string size factored for pattern length for brute force and Boyer-Moore.





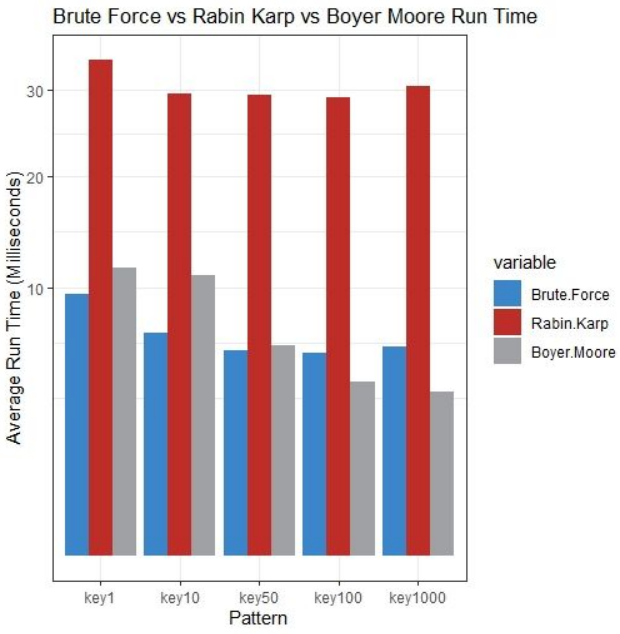
On average Boyer-Moore runs 6.95% longer than brute force. But for longer patterns like 100 and 1000 characters, Boyer-Moore runs 63% faster than brute force. For work done, on average Boyer-Moore makes 99.22% less comparions than brute force!

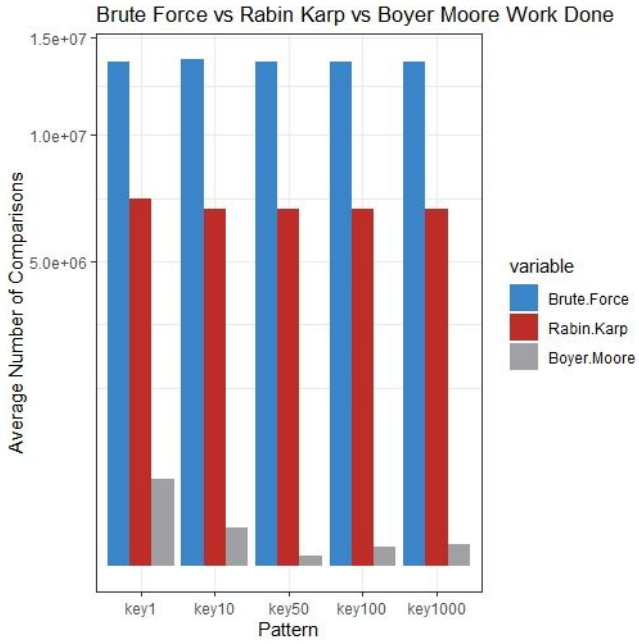




Conclusion

This project implemented and analyzed three searching algorithms: Brute Force, Rabin Karp, and Boyer-Moore. String searching is an immense part of computer science. It is used all the time for web browsers like google and in biological sciences for bioinformatics and gene sequencing. After running computational analyses on all three algorithms, Brute force is the most efficient algorithm minimizing the work done by a naïve brute force algorithm by nearly 100%. The computational cost of which doesn't greatly worsen the run times especially in its best cases. Rabin Karp does less work than brute force, but the computational costs of the hash function worsen runtimes. A visualization of how all three algorithms compare in regards to run time and work done are included below.





Contributions

A. Rabin Karp
    1. Coded solution simultaneously
    2. Both spent individual time working on optimization of original solution
    3. David edited code to work with timer class and reading in large data
B. Boyer Moore
    1. Raechel coded solution
    2. David optimized and edited code to work with timer/comp class and reading in large data
C. Brute Force
    1. Raechel wrote a brute force algorithm for comparison purposes
D. Runtime and Comps
    1. David implemented a timer class, comparison class, file parser class, and ran the benchmarks to collect the data
E. Graphs and plots
    1. Raechel used R software to get statistical data and make all the plots
F. Project report
    1. Both contributed. David contributed most of the sections. Raechel wrote sections on graphs and statistical analysis
G. Powerpoint
    1. Raechel made powerpoint

Hardware/Software specs for benchmarking:

Processor: Intel i7-4790k @ 4.00GHz

RAM: 16 GB

Operating System: Windows 10, Linux subsystem (WSL 1)

Compile command: g++ -O3 -std=c++17 ex.cpp main.cpp -o ex

References

AbdulRazzaq A., A., Rashid A., A. N., Hasan A., A., & Abu-Hashem, A. M. (2013). The exact string matching algorithms efficiency review. *Vol 04*, pp 576-589. AWERProcedia Information Technology & Computer Science. Retrieved from www.awer-center.org/pitcs

Charras, C., & Lecroq, T. (2004). *Handbook of Exact String-Matching Algorithms.* France: King's College Publications.

Diwate, R., & Alaspurkar, S. (2013). *Study of Different Algorithms for Pattern Matching.* International Journal of Advanced Research in Computer Science and Software Engineering.

Sedgewick, R., & Wayne, K. (2011). *Algorithms, 4th Edition.* Princeton University. Retrieved from https://algs4.cs.princeton.edu/home/

Christian Lovis, MD, Robert H. Baud, PhD, Fast Exact String Pattern-matching Algorithms Adapted to the Characteristics of the Medical Language, *Journal of the American Medical Informatics Association*, Volume 7, Issue 4, July 2000, Pages 378–391, https://doi.org/10.1136/jamia.2000.0070378

Krishnan, Narayanan Chatapuram. "Pattern Matching." Indian Institute of Technology, 2015. Website. <https://cse.iitrpr.ac.in/ckn/courses/f2015/csl201/w12.pdf>.

Appendix A

R source Code

```
#RK String Length vs RunTime factor Pattern
> library(ggplot2)
> data<-read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\rk_timer.csv", header=T)
> time <- data$duration
> length <- data$file_name
> ggplot(data, aes(x = length, y = time, colour = pattern)) + geom_line()
> rk <- ggplot(data, aes(x = length, y = time, colour = pattern)) + geom_line()
> print(rk + ggtitle("Rabin Karp Runtime"))
> print(rk + ggtitle("Rabin Karp Runtime") +labs(y = "Time in Milliseconds",
+ x = "Length of String"))
>

#BF String Length vs RunTime factor Pattern
data <- read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\bf_timer.csv", header=T)
> time <- data$duration
> length <- data$file_name
> bf <- ggplot(data, aes(x = length, y = time, colour = pattern)) + geom_line()
> print(bf + ggtitle("Rabin Karp Runtime") +labs(y = "Time in Milliseconds",
+ x = "Length of String"))
> print(bf + ggtitle("Brute Force Runtime") +labs(y = "Time in Milliseconds",
+ x = "Length of String"))
>

#BF vs RK: Average Runtime vs Pattern
>
data<-read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\final_data\\timer_results\\timer_table_rk.csv", header=T)
> data<-melt(data)
Using Pattern as id variables
> data$Pattern <- as.character(data$Pattern)
> data$Pattern <- factor(data$Pattern, levels=unique(data$Pattern))
> rk <- ggplot(data, aes(x=Pattern,y=value,fill=variable))+geom_bar(stat="identity",
position="dodge")+
+ theme_bw()+ylab("Average Run Time (Milliseconds)") +xlab("Pattern") +ggtitle("Brute Force vs Rabin Karp Run Time")
```

```
> rk + scale_fill_manual(values = c("#3B85CA","#BF2D29")) +
scale_y_continuous(trans='sqrt')


#BF vs RK: Average Comps vs Pattern
> data <- read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\comps_table.csv",
header=T)
> data<-melt(data)
Using Pattern as id variables
> ggplot(data, aes(x=Pattern,y=value,fill=variable))+geom_bar(stat="identity",
position="dodge")+
+ theme_bw()+ylab("Average Number of Comparisons") +xlab("Pattern")
>

#RK BF Runtime Factor Cases
> data <- read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\timer_cases.csv",
header=T)
> data<-melt(data)
Using Case as id variables
> ggplot(data, aes(x=Case,y=value,fill=variable))+geom_bar(stat="identity", position="dodge")+
+ theme_bw()+ylab("Time in Milliseconds") +xlab("Case") + ggtitle("Best and Worse case
Runtimes")
>

#RK BF Comps Factor Cases
> data <- read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\comp_cases.csv",
header=T)
> data<-melt(data)
Using Case as id variables
> ggplot(data, aes(x=Case,y=value,fill=variable))+geom_bar(stat="identity", position="dodge")+
+ theme_bw()+ylab("Number of Comparisons") +xlab("Case") + ggtitle("Best and Worse case
Comparisons")


#BM String Length vs RunTime factor Pattern
> library(ggplot2)
>
data<-read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\\final_data\\bm_timer.csv",
header=T)
> time <- data$duration
```

```
> length <- data$str_len
> ggplot(data, aes(x = length, y = time, colour = pattern)) + geom_line()
> bm <- ggplot(data, aes(x = length, y = time, colour = pattern)) + geom_line()
> print(bm + ggtitle("Boyer Moore Runtime") +labs(y = "Time in Milliseconds",
+ x = "Length of String"))
>

#BM vs BF: Average Comps vs Pattern
> data <-
read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\final_data\\comps_table.csv",
header=T)
> data<-melt(data)
Using Pattern as id variables
> ggplot(data, aes(x=Pattern,y=value,fill=variable))+geom_bar(stat="identity",
position="dodge")+
+ theme_bw()+ylab("Average Number of Comparisons") +xlab("Pattern")
>


#BF vs BM: Average Runtime vs Pattern
>data<-read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\final_data\\timer_results\\t
imer_table_bm.csv", header=T)
> data<-melt(data)
Using Pattern as id variables
> data$Pattern <- as.character(data$Pattern)
 b <- ggplot(data, aes(x=Pattern,y=value,fill=variable))+geom_bar(stat="identity",
position="dodge")+
+ theme_bw()+ylab("Average Run Time (Milliseconds)") +xlab("Pattern") +ggtitle("Brute Force
vs Boyer Moore Run Time")
> bm + scale_fill_manual(values = c("#3B85CA","#A0A1A4")) +
scale_y_continuous(trans='sqrt')


#All Time
>data<-read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\final_data\\timer_results\\t
imer_table_all.csv", header=T)
> data<-melt(data)
Using Pattern as id variables
data$Pattern <- as.character(data$Pattern)
> data$Pattern <- factor(data$Pattern, levels=unique(data$Pattern))
```

```
> all <- ggplot(data, aes(x=Pattern,y=value,fill=variable))+geom_bar(stat="identity",
position="dodge")+
+ theme_bw()+ylab("Average Run Time (Milliseconds)") +xlab("Pattern") +ggtitle("Brute Force
vs Rabin Karp vs Boyer Moore Run Time")
> all + scale_fill_manual(values = c("#3B85CA","#BF2D29", "#A0A1A4")) +
scale_y_continuous(trans='sqrt')


#All Comps
>
data<-read.csv("C:\\Users\\raegr\\OneDrive\\Documents\\R\\csc212\\final_data\\compresults\\co
mp_table_all.csv", header=T)
> data <- melt(data)
Using Pattern as id variables
> all <- ggplot(data, aes(x=Pattern,y=value,fill=variable))+geom_bar(stat="identity",
position="dodge")+
+ theme_bw()+ylab("Average Number of Comparisons") +xlab("Pattern") +ggtitle("Brute force
vs Boyer Moore vs Rabin Karp Work Done")
> all + scale_fill_manual(values = c("#3B85CA", "#BF2D29" ,"#A0A1A4")) +
scale_y_continuous(trans='log2')
>
```