String-matching Algorithms

1. The Naive String-Matching Algorithm
   The most method of string-matching uses two loops: the outer loop iterating over text string T and the inner loop iterating over pattern string P to find all possible matches of T[s + 1…..s+m] = P[1…m] for n-m+1 possible values of s. Every time a mismatch is found, the window T[s + 1…..s+m] is shifted by one character and compared again with P.

   <u>The Pseudocode for Naïve String-Matching Algorithm:</u>

```
NAIVE-STRING-MATCHER($T, P$)
1  $n = T.length$
2  $m = P.length$
3  for $s = 0$ to $n - m$
4      if $P[1..m] == T[s+1..s+m]$
5          print "Pattern occurs with shift" $s$
```

   This approach has a running time of O(nm).

2. Knuth-Morris-Pratt (KMP) Algorithm
   KMP is a linear-time string-matching algorithm with a running time of O(n) where 'n' is the length of the input text string. It implements a function COMPUTE-PREFIX-FUNCTION(P) that takes input pattern 'P' and computes the corresponding integer array known as the prefix array obtained by comparing the pattern to itself and recording whether a substring is the prefix of the overall pattern. For example, the pattern 'abcabcnab' would have the following prefix array:

index	1	2	3	4	5	6	7	8	9
P	a	b	c	a	b	c	n	a	b
prefix	0	0	0	1	2	3	0	1	2

   A) <u>The pseudocode for COMPUTE-PREFIX-FUNCTION(P):</u>

```
COMPUTE-PREFIX-FUNCTION($P$)
1  $m = P.length$
2  let $\pi[1..m]$ be a new array
3  $\pi[1] = 0$
4  $k = 0$
5  for $q = 2$ to $m$
6      while $k > 0$ and $P[k+1] \neq P[q]$
7          $k = \pi[k]$
8      if $P[k+1] == P[q]$
9          $k = k + 1$
10     $\pi[q] = k$
11 return $\pi$
```

COMPUTE-PREFIX-FUNCTION(P) has a running time of O(m) where ‘m’ is the length of the pattern string. The outer loop executes a total of ‘m’ times and the while loop inside executes only if k > 0 (i.e. a substring in the pattern string contains the prefix) and there is a mismatch found during the computation. As the execution of the while loop is contingent upon k > 0 and the total increase in k is at most m-1(from lines 8 and 9), the while loop also executes a total of m-1 times at most. During each execution of the while loop, the value of k decreases but has a lower bound of 0. Therefore, the total number of times the while loop runs is bounded above by the number of times the for loop executes with is equal to m – 1.

B) <u>Pseudocode for KMP-MATCHER (T, P):</u>

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                    // number of characters matched
for i = 1 to n                           // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                         // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                        // next character matches
    if q == m                            // is all of P matched?
        print “Pattern occurs with shift” i − m
        q = π[q]                         // look for the next match
```

Having the prefix function (π) for the given pattern allows us to avoid unnecessary shifts that cost more time in the naive string-matching algorithm. For example, if 5 characters of a substring in T have matched with 5 characters in the pattern P but the 6th character is a mismatch, instead of starting over by checking if the 2nd character in the substring matches with the 1st character in the pattern, we can use the information in the prefix array and the fact that the last 5 characters in T were a match to directly compare the 6th character with P(π[5]).

Similar to the running time analysis of COMPUTE-PREFIX-FUNCTION(P), the outer loop in KMP-MATCHER (T, P) executes ‘n’ times and the while loop also executes at most ‘n’ times in total. This is because the while loop’s execution is contingent upon q > 0 and q only gets incremented once per for loop. To put it simply, the while loop represents the scenario when backtracking is required after a mismatch of characters is found in a potential match substring in T. Therefore, the while loop decreases the value of q but q cannot be negative. As the total decrease in q is bounded from above by the total increase in q over the for loop (which is at most n), the while loop executes at most ‘n’ number of times and **the running time of KMP-MATCHER (T, P) is O(n).**

3. Rabin-Karp Algorithm

The algorithm proposed by Rabin and Karp for string matching has a worst-case running time of O((n – m + 1)m) but is known to perform better in average cases (O(n + m)). It uses the fact that if two numbers modulo a third number is equal, the two numbers are equivalent. Therefore, before characters are compared, each character is given a digit representation and the pattern P[1…m] and text substring T[s+1…s+m] are converted to a numeric values p and t_s using Horner's rule:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1]) \cdots))$$

P can be computed in O(m) running time. In addition, computing t_{s+1} from t_s can be done in constant time using the formula:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Therefore, computing all possible values from t_1 to t_{n-m} can be done in O(n – m + 1) running time. This means that by comparing p with t_s, all valid shifts can be found in O(m)+O(n – m + 1) = O(n) amount of time. However, this is only possible if the alphabet ∑ = {0, 1, 2, ……9} which is not the case in most real-world string-matching scenarios. To address this issue instead of directly comparing p with t_s, we can compare the values of p and t_s modulo a suitable value q. For a d-nary alphabet, we can choose q such that dq fits within the max value of the variable type being used. This changes the second equation above to:

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

where, h = d^{m-1} (mod q)

It is important to note that t_s equivalent p (mod q) does not imply t_s = p. However, if t_s is not equivalent to p (mod q) we can be sure that $t_s \neq$ p and move on to checking t_{s+1}. In this way, the Rabin-Karp algorithm uses the test t_s equivalent p (mod q) as a fast way to rule out invalid shifts.

The pseudocode for RABIN-KARP-MATCHER (T, P, d, q):

```
RABIN-KARP-MATCHER($T, P, d, q$)
$n = T.length$
$m = P.length$
$h = d^{m-1} \bmod q$
$p = 0$
$t_0 = 0$
for $i = 1$ to $m$                    // preprocessing
    $p = (dp + P[i]) \bmod q$
    $t_0 = (dt_0 + T[i]) \bmod q$
for $s = 0$ to $n - m$                // matching
    if $p == t_s$
        if $P[1..m] == T[s+1..s+m]$
            print "Pattern occurs with shift" $s$
    if $s < n - m$
        $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$
```

The preprocessing step to calculate the hash values for Pattern P and the first window t_s takes O(m) amount of time. If $t_s \equiv p \pmod{q}$, that part of the text string T is compared to the pattern P to see if it is a match. If not, t_{s+1} is computed. This takes $O((n - m + 1)\,m)$ amount of time. However, on average there are often less number of valid shifts in which case the running time can be expressed as $O((n - m + 1) + cm) = O(n + m)$.

Experiment

In this experiment, all 3 string-matching algorithms mentioned above were tested:

1)	Naive string-matching algorithm	O(nm)
2)	Rabin-Karp matcher	O(m(n-m+1))
3)	Knuth-Morris-Pratt matcher	O(n)

where, n = input string size
m = pattern string size

Experimental Data

Pattern Strings

The experimental data used in this experiment can be categorized into two types: pattern string containing repeating prefixes and pattern string without repeating prefixes. The length of the pattern string used was 50.

1) Regular Pattern: `sdjhfncuhiuexlshgimxajijdfimijonknlmciojimosmihtsb`
2) Repeating prefix: `abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabdeabcde`

Text Strings

A custom string generator was used to generate 14 text string data files – 7 containing the regular pattern and 7 containing the pattern with repeated prefixes. The length of the text string ranged from 10^4 to 10^{10}. The text string was generated in such a way that each text string of length 10^n had the first half of the pattern string randomly placed in 10^n /100 index positions and the complete pattern placed at the very end of the text string. For example, a text string of length 10000 contains potential yet false matches at 100 random index positions. This was done to simulate average to worst-case scenarios for the algorithms.

Results and Discussion

Each algorithm was run 5 times per dataset and an average was taken:

Table 1: Dataset Type 1 - Text String with Regular Pattern

	Running Time [s]		
Text Length	Naive String Matcher	Rabin-Karp	Knuth-Morris-Pratt
10^4	0.0000838	0.00018	0.0000718
10^5	0.0005322	0.0013796	0.000457
10^6	0.0049844	0.0134308	0.0044534
10^7	0.0493714	0.1325804	0.0428154
10^8	0.503074	1.3446374	0.4436312
10^9	5.1161968	13.500118	4.3826766
10^{10}	7.0858948	19.0353528	5.5794682

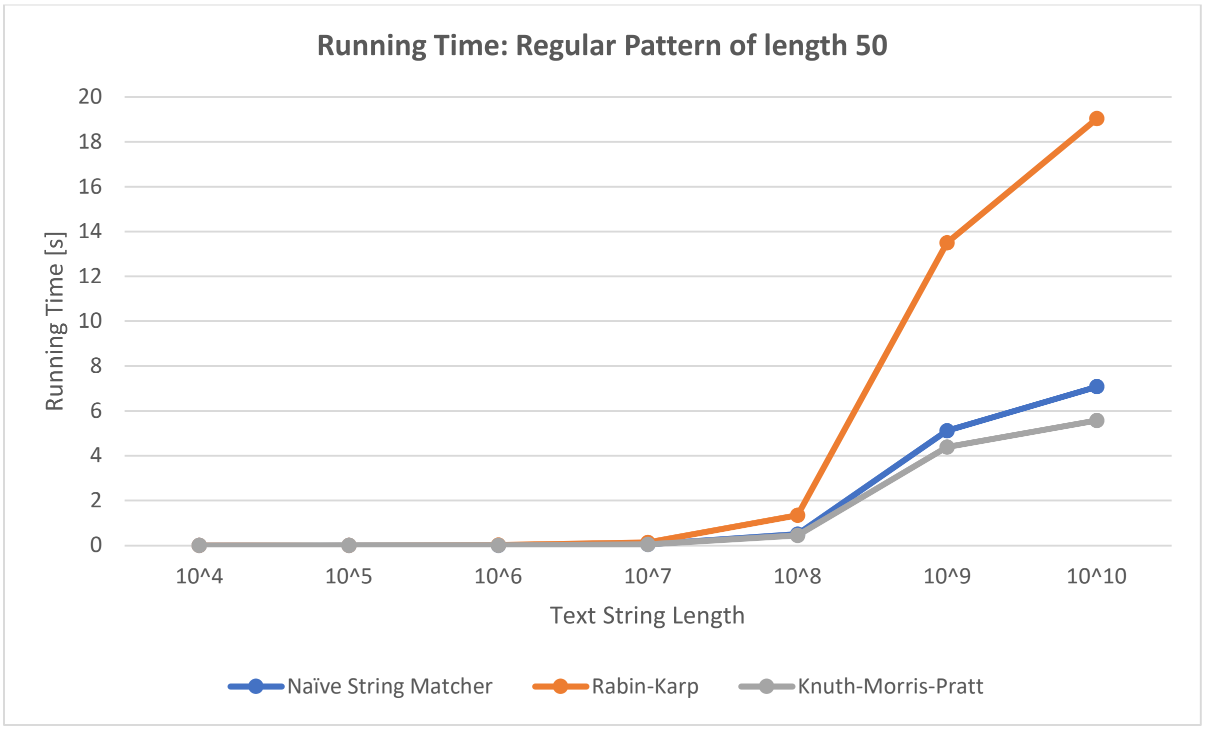

Figure 1: Running time of 3 different string-matching algorithms to find a regular pattern without repeating prefixes.

The line graph above shows that the KMP matcher had the best performance overall followed closely by the Naive string-matching algorithm. Rabin-Karp was found to have the worst performance for text strings of all lengths. The Naive string matcher was found to perform almost as well as the KMP matcher for text strings of size 10^4 to 10^8. This can be explained by the fact that the length of the pattern string(m) was significantly small in comparison to the text strings. Although the worst-case running time for the Naïve matcher is O(nm), if m is significantly smaller than n, $O(nm) \approx O(n)$.

In the case of Rabin-Karp, using an alphabet of size 256 and a large value for q(INT_MAX) in the implementation may have resulted in a longer search time by making the hashing part of the algorithm extremely costly.

Table 2: Dataset Type 2 - Text String with Pattern containing Repeated Prefixes

	Running Time [s]		
Text Length	Naïve String Matcher	Rabin-Karp	Knuth-Morris-Pratt
10^4	0.0002586	0.0001726	0.0000874
10^5	0.000671	0.0013786	0.0004738
10^6	0.00645	0.0133636	0.004544
10^7	0.0645946	0.1338966	0.0446092
10^8	0.6510336	1.3510188	0.4534262
10^9	6.5051192	13.566869	4.6160292
10^{10}	9.1814228	18.651952	6.3608282

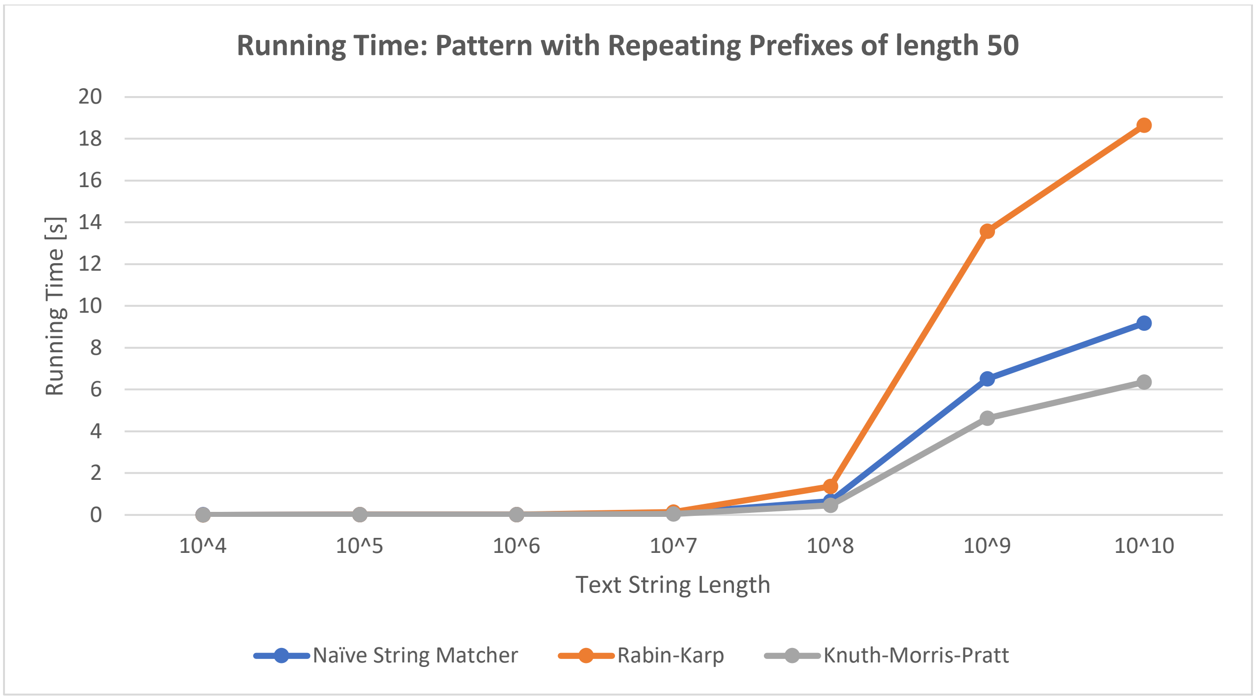

Figure 2: Running time of 3 different string-matching algorithms to find a pattern containing repeated prefixes.

For datasets containing the pattern with repeated prefixes, KMP matcher had the best performance followed by the Naive string matcher and finally Rabin-Karp matcher. In comparison to the datasets with the regular pattern, the performance gap between KMP matcher and the Naive matcher was found to be greater while searching for a pattern with repeated prefixes. Results are shown in the table and line graph below:

Table 3: Time in seconds by which KMP matcher beats Naive matcher

	Time taken by KMP [s] – Time taken by Naïve matcher [s]	
Text String Length	Regular Pattern	Repeating prefixes
10^4	0.000012	0.0001712
10^5	0.0000752	0.0001972
10^6	0.000531	0.001906
10^7	0.006556	0.0199854
10^8	0.0594428	0.1976074
10^9	0.7335202	1.88909
10^10	1.5064266	2.8205946

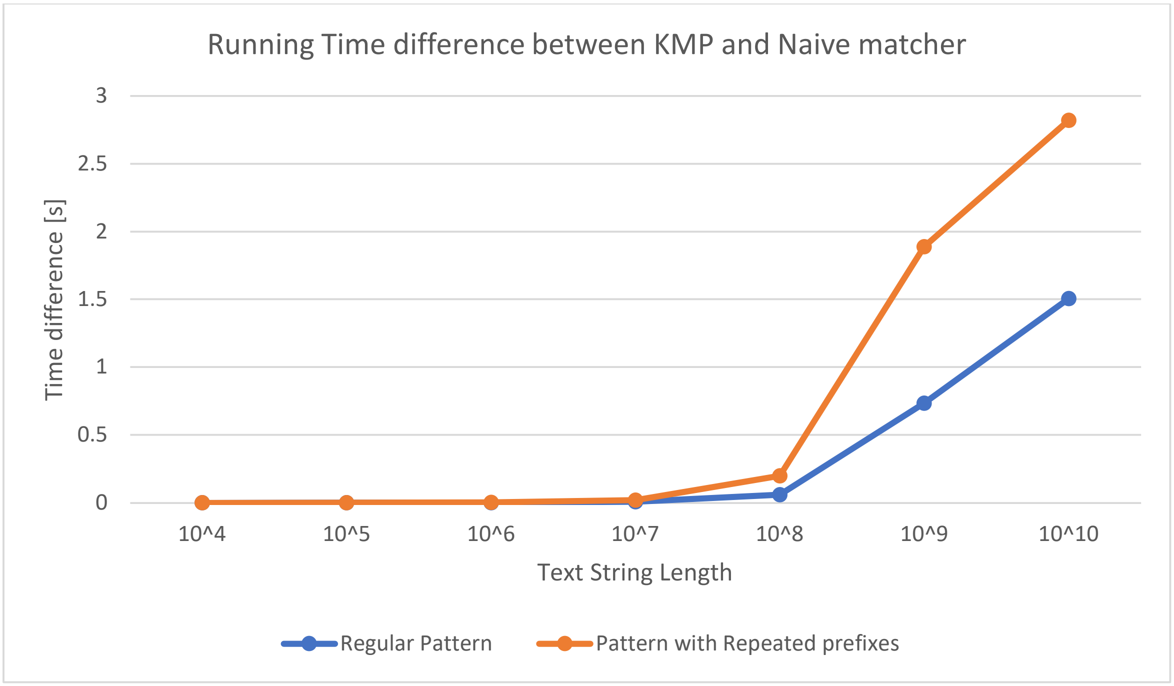

Figure 3: Line graph showing time in seconds by which KMP beats the naïve string-matcher for regular pattern vs pattern with repeating prefix.

The KMP matcher was able to outperform the naïve string matcher, especially for text strings of length greater than 10^8. This is because the KMP matcher uses a prefix array for the pattern string to avoid unnecessary backtracking in both the text and the pattern string Therefore, searching for a pattern with repeating prefixes gives the KMP matcher a significant advantage. Figure no. 3 shows that for a text string of length 10^{10}, the KMP matcher outperformed the Naive string matcher by 1.5 seconds while searching for a regular pattern vs 3 seconds while searching for a pattern with repeating prefixes. The Rabin-Karp matcher continued to have the longest search time for text strings of all lengths.

Conclusion

The performance of string-matching algorithms was found to greatly depend on the type of data for each algorithm. For example, text string data that does not contain any potential yet false candidates for a match can significantly reduce the running time for the naive matcher; however, for such data, the running time for Rabin-Karp does not significantly improve as the hash value of each window of size ‘m’ will still have to be checked. For most of the experimental data, the naive matcher performed almost as well as the KMP matcher as the size of the pattern string remained constant. KMP matcher was found to outperform two other implementations for each dataset and perform even better while searching for patterns with repeating prefixes. The experiment also showed that the Rabin-Karp algorithm was not able to perform as well as the other two implementations for any of the datasets. A possible reason behind Rabin-Karp’s poor performance could be the use of a large alphabet (d = 256) as well as a large value for q (INT_MAX)..