# String Matching

Nishna Reddy Aleti

Algorithm Design and Analysis

Portland State University

## Table of contents:

Github: https://github.com/nishnareddy1/StringMatching

## 1. Abstract

String Matching algorithms try to find a place where one or several strings are found within a larger string or text. In this paper, theoretical and experimental comparative study is done between three different String-matching algorithms namely Naïve string-matching algorithm, Rabin-Karp algorithm and Knuth-Morris-Pratt algorithm. Also compared the efficiencies of these algorithms by searching speed, pre-processing time, matching time and the key ideas used in these algorithms.

## 2. Introduction

The objective of string matching is to find the location of a specific text pattern within a larger body of text. We can generalize this string pattern matching problem as follows: Given an alphabet A, a text T (an array of n characters in A) and a pattern P (another array of $m \leq n$ characters in A), we say that P occurs with shift s in T (or P occurs beginning at position s + 1 in T) if $0 \leq s \leq n - m$ and $T[s + j] = P[j]$ for $1 \leq j \leq m$. A shift is valid if P occurs with shift s in T and invalid otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T. [1]

T: A B B A A C A A C A

P: A B B

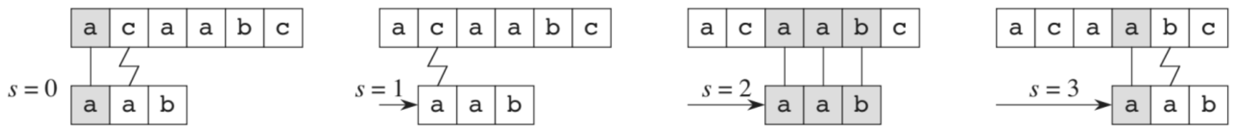
Valid shifts are at index zero and six (t[0] and T[6]).

Text-editing programs frequently need to find all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for string matching can greatly aid the responsiveness of the text-editing program. Internet search engines also use them to find Web pages relevant to queries. This simple problem has a lot of application in the areas of Information Security, Pattern Recognition, Document Matching, Bioinformatics (DNA matching) among others.

## 3. Naïve String Matching algorithm

A simple and inefficient way to see where one string occurs inside another is to check each place it could be, one by one, to see if it's there. So first we see if there's a copy of the pattern in the first character of the text; if not, we look to see if there's a copy of the pattern starting at the second character of the text; if not, we look starting at the third character, and so forth.

Consider pattern, P="CAB" and text, T="CACAB". Initially, P is aligned with T at the first index position. P is then compared with from left-to-right. If a mismatch occurs, slide P to right by 1 position, and start the comparison again. [2]



### 3.1. Algorithm

```
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)
```

### 3.2. Time Complexity Analysis

**Best case:**

The best case occurs when the first character of the pattern is not present in text at all.
text[] = "ABBCACADDEE";
pattern[] = "FAB";

The number of comparisons in best case is O(n), Where n is the length of the text[].

**Worst Case:**

The worst case of Naive Pattern Searching occurs in following scenarios.

1) When all characters of the text and pattern are same.
   text[] = "AAAAAAAAAAAAAAAAAA";
   pattern[] = "AAAAA";
2) Worst case also occurs when only the last character is different.
   text[] = "AAAAAAAAAAAAAAAAAB";
   pattern[] = "AAAAB";

The number of comparisons in the worst case is O(m*(n-m+1)), Where m and n is the length of the pattern[] and text[].

## 4. Rabin-Karp algorithm

This string searching algorithm uses hashing to speed up the search. A hash function is a function which converts every string into a numeric value which is called hash value. The Rabin-Karp string searching algorithm calculates a hash value for the pattern, and for each M-character subsequence of text to be compared. If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence. If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence.

In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

**Pre-processing:**

The preprocessing phase of the Rabin-Karp algorithm consists in computing hash(x). It can be done in constant space and O(m) time because hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say hash(text[s+1 .. s+m]) must be efficiently computable from hash(text[s .. s+m-1]) and text[s+m] i.e., hash(text[s+1 .. s+m]) = rehash(txt[s+m], hash(text[s .. s+m-1]) and rehash must be O(1) operation.

However, there is a problem with this approach. First, because there are so many different strings and so few hash values, some differing strings will have the same hash value. If the hash values match, the pattern and the substring may not match; consequently, the potential match of search pattern and the substring must be confirmed by comparing them; that comparison can take a long time for long substrings. This kind of hits when hash value of substring of text and pattern is the same but that substring of text is not same as pattern string it is called spurious hit or hash collision. A hash function of sufficiently large prime number will provide distinct hashed values and does not have many hash collisions, so the expected search time will be acceptable.

The hash function suggested by Rabin-Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. [2]

Rehashing is done using the following formula:

hash(text[s+1 .. s+m]) = ( d (hash( text[s .. s+m-1]) – text[s]*h) + txt[s + m]) mod q

hash(text[s .. s+m-1]) : Hash value at shift s.

hash(text[s+1 .. s+m]) : Hash value at next shift (or shift s+1)

Where d = Number of characters in the alphabet, q = A prime number, h= $d^{m-1}$ mod q

**4.1. Algorithm**

```
RABIN-KARP-MATCHER(T, P, d, q)
1.  n=T.length
2.  m=P.length
```

```
3.  h = d^{m-1} mod q
4.  p = 0
5.  t_0 = 0
6.  for i = 1 to m                              // preprocessing
7.        p = (dp + P[i]) mod q
8.        t_0 = dt_0 + T [i]) mod q
9.  for s = 0 to n-m                            // matching
10.       if p == t_s
11.             if P [1... m] == T [s +1..s + m]
12.                   print ("Pattern occurs with shift", s)
13.       if s < (n – m)
14.             t_s+1 = (d(t_s - T [s+1]h)+T(s + m + 1])mod q
```

**Working of an algorithm:**

All characters are interpreted as radix-d digits. Line 3 initializes h to the value of the high-order digit position of an m-digit window. Lines 4–8 compute p as the value of P[1....m] mod q and t0 as the value of T[1....m] mod q. The for loop of lines 9–14 iterates through all possible shifts s, maintaining the following invariant: Whenever line 10 is executed, ts = T[s+1…..s+m] mod q. If p = ts in line 10 (a "hit"), then line 11 checks to see whether P[1....m] = T[s+1….s+m] in order to rule out the possibility of a spurious hit. Line 12 prints out any valid shifts that are found. If s <n - m (checked in line 13), then the for loop will execute at least one more time, and so line 14 first executes to ensure that the loop invariant holds when we get back to line 10. Line 14 computes the value of ts+1 mod q from the value of ts mod q in constant time using equation shown in line 14. [2]

**4.2.Time Complexity Analysis**

**Best case:**

The average and best-case running time of the Rabin-Karp algorithm is O(n+m). In many applications, we expect few valid shifts, perhaps some constant c of them. In such applications, the expected matching time of the algorithm is only O((n-m+1)+cm) =O(n+m), plus the time required to process spurious hits.

We can base a heuristic analysis on the assumption that reducing values modulo q acts like a random mapping from Σ* to Zq (where Σ={0,1,2,3,...,9} and d=|Σ| and each character is a digit in radix-d notation). We can then expect that the number of spurious hits is O(n/q), since we can estimate the chance that an arbitrary ts will be equivalent to p, modulo q, as 1/q. Since there are O(n) positions at which the test of line 10 fails and we spend O(m) time for each hit, the expected matching time taken by the Rabin-Karp algorithm is O(n)+O(m(v+n/q)), where v is the number of valid shifts. This running time is O(n) if v=O(1) and we choose $q \geq m$. That is, if the expected number of valid shifts is small (O(1)) and we choose the prime q to be larger than the length of the pattern, then we can expect the Rabin-Karp procedure to use only O(n+m) matching time. Since m ≤ n, this expected matching time is O(n).

**Worst Case:**

RABIN-KARP-MATCHER takes Θ(m) preprocessing time, and its matching time is Θ((n-m+1)m) in the worst case, since like the naive string-matching algorithm the Rabin-Karp algorithm explicitly verifies every valid shift. If P = am and T = an, then verifying takes time Θ((n-m+1)m), since each of the n-m+1 possible shifts is valid. Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of text match with hash value of pattern. For example pattern = “AAA” and text = “AAAAAAA”. So for each hash value it will compare whole substring to pattern in this case. So worst case time complexity of Rabin-Karp algorithm is O(m(n-m+1)).

## 5. Knuth-Morris-Pratt algorithm

The Naive pattern searching algorithm doesn’t work well in cases where we see many matching characters followed by a mismatching character. The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n). The idea behind KMP’s algorithm is whenever we detect a mismatch, we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

**Preprocessing:**

KMP algorithm preprocesses pattern and constructs an auxiliary longest proper prefix(lps) which is also suffix of size same as size of pattern which is used to skip characters while matching. A proper prefix is prefix with whole string not allowed. For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC” and “ABC”.
We search for lps in sub-patterns. More clearly, we focus on sub-strings of patterns that are either prefix and suffix. For each sub-pattern P[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern P[0..i].

**Searching Algorithm:**

Unlike Naive algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

### 5.1.Algorithm

```
COMPUTE-PREFIX-FUNCTION(P)

1.  m= P.length
2.  let lps[1..m] be a new array
3.  lps[1]=0
4.  k =0
5.  for q = 2 to m
6.          while k>0 and P[k + 1] =! P[q]
7.                  k = lps[k]
8.          if P[k+1] = = P[q]
9.                  k = k+1
10.         lps[q] = k
11. return lps
```

```
KMP-MATCHER( T, P )

1.  n=T.length
2.  m=P.length
3.  lps = COMPUTE-PREFIX-FUNCTION(P)
4.  q = 0                                   // number of characters matched
5.  for i = 1 to n                          // scan the text from left to right
6.          while q>0 and P[q + 1] =! T[i]
7.                  q = lps[q]              // next character does not match
8.          if P[q + 1] == T[i]
9.                  q = q + 1               // next character matches
10.         if (q == m) :                   // is all of P matched?
11.                 print("Pattern occurs with shift", i-m)
12.                 q = lps[q]              // look for the next match
```

**Working of compute-prefix-function or preprocessing:**

In the preprocessing part, we calculate values in lps[] (longest prefix suffix values). To do that, we keep track of the length of the longest prefix suffix value (we use k variable for this purpose) for the previous index. We initialize lps[1] and k as 0. Now we use for loop from q=2 to m (m=length of pattern String) to find the longest prefix suffix values. In for loop these comparison will be performed. If P[k+1] and P[q] match, we increment k by 1 and assign the incremented value to lps[q]. If P[q] and P[k+1] do not match and k is not 0, we update k to lps[k].

**Working of searching algorithm:**

We start comparison of P[j] with j = 0 with characters of current window of text. We keep matching characters T[i] and T[j] and keep incrementing i and j while P[j] and T[i] keep matching.
When we see a mismatch:

- We know that characters P[0..j-1] match with T[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
- We also know (from above definition) that lps[j-1] is count of characters of P[0…j-1] that are both proper prefix and suffix.
- From above two points, we can conclude that we do not need to match these lps[j-1] characters with T[i-j…i-1] because we know that these characters will anyway match.

### 5.2.Time Complexity Analysis

Preprocessing time of KMP is $\Theta(m)$ which we can show by the aggregate method of amortized analysis.

- First, line 4 starts k at 0, and the only way that k increases is by the increment operation in line 9, which executes at most once per iteration of the for loop of lines 5–10. Thus, the total increase in k is at most m-1.
- Second, since $k < q$ upon entering the for loop and each iteration of the loop increments q, we always have $k < q$. Therefore, the assignments in lines 3 and 10 ensure that $\pi[q] < q$ for all $q = 1,2,...,m$ which means that each iteration of the while loop decreases k.
- Third, k never becomes negative. Putting these facts together, we see that the total decrease in k from the while loop is bounded from above by the total increase in k over all iterations of the for loop, which is m - 1.

Matching time of KMP is $\Theta(n)$ which we can show by the aggregate analysis. The total number of executions of the while loop of line 6 is $\Theta(n)$. For each iteration for the for loop of line 5, q increases by at most 1, in line 9. This is because $\pi(q) < q$. On the other hand, the while loop decreases q. Since q can never be negative, we must decrease q fewer than n times in total, so the while loop executes at most $n - 1$ times. Thus, the total runtime is $\Theta(n)$ [2].

## 6. Experimental results

### 6.1. Input Datasets and setup

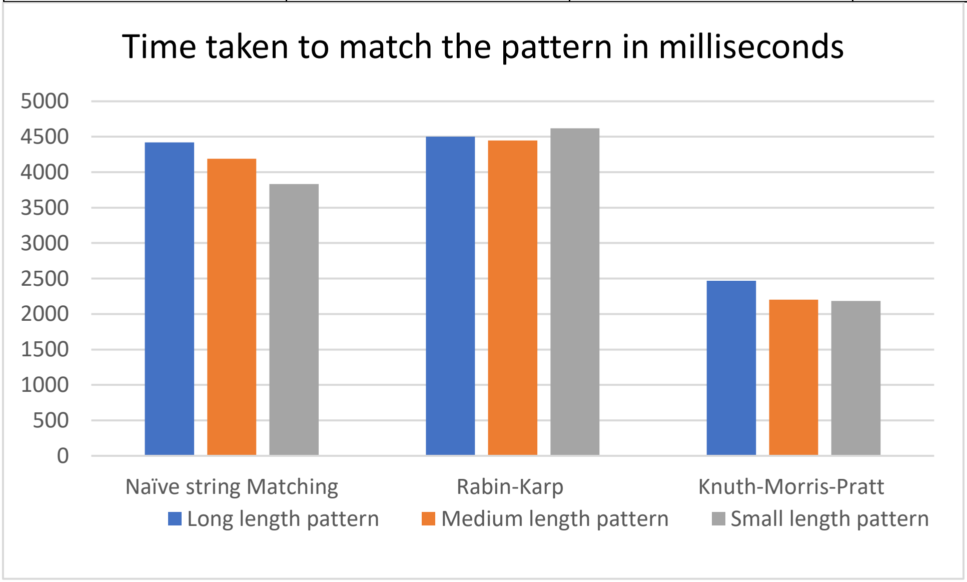
The experimental setup is designed to compare the running time of Naïve, Rabin-Karp, Knuth-Morris-Pratt and Finite Automata algorithms for 2 different large input datasets. One input dataset is a novel by Charles Dickens called ‘A Tale of Two Cities’ [5] which consists of 760,424 characters with spaces. Other dataset is a large file called ‘PCLA clusters’ from Genome project about protein DNA sequence [6] whose size is 90.2MB with 94,590,177 characters with spaces. All the algorithms used the same dataset as input when comparing. The computation is repeated 10 times to obtain an accurate measurable time. It is implemented on Intel i7 processor having 8GB RAM. All the algorithms are implemented in python.

## 6.2. Timing Results

**Test case 1**

For the input 'A Tale of Two Cities', I considered a pattern with 92,000 characters, 9720 characters and 63 characters. A plot of average time taken to match the pattern in milliseconds on y-axis and the algorithms on the x-axis is shown.

| | Brute Force | Rabin karp | Knuth Morris |
|---|---|---|---|
| Average time taken to find the longest length pattern- 92k characters | 4420 ms | 4502 ms | 2470 ms |
| Average time taken to find the medium length pattern- 9k characters | 4192 ms | 4447 ms | 2204 ms |
| Average time taken to find the shortest length pattern- 63 characters | 3834 ms | 4617 ms | 2183 ms |

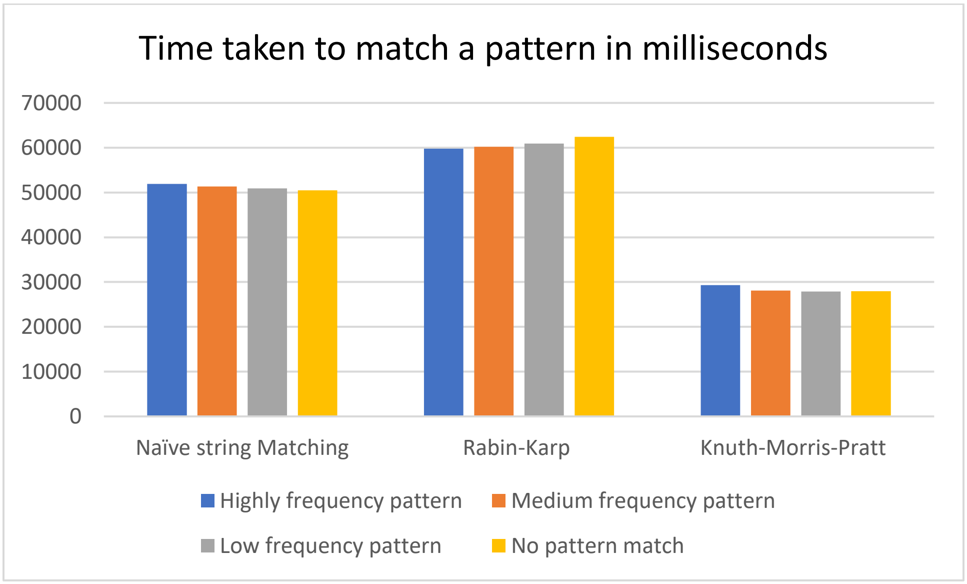


From the above graph we can say that Knuth-Morris-Pratt algorithm works the best among all three algorithms. As the length of the pattern increases the time complexity also increases. For Naïve string-matching algorithm and Knuth-Morris-Pratt algorithm take more time as the length of the string increases. Rabin Karp works worse than Naïve string matching algorithm because of large number of hash collisions.

**Test Case 2:**

For the input 'PCLA clusters', I have considered four different DNA patterns with approximately equal pattern length, one is a highly frequent pattern- 'ribosomal protein' with 2573 matches, medium frequent pattern- '50S ribosomal protein L3' with 572 matches, low frequent pattern- 'ribonucleoside-diphosphate reductase' with 34 matches and 'ribonucleoside-diphosphate reductase L3' with 0 matches. A plot of average time in milliseconds on y-axis and the algorithms on the x-axis is shown.

| | **Naïve String-Matching** | **Rabin Karp** | **Knuth Morris Pratt** |
|---|---|---|---|
| Average time taken to find highly frequency pattern- 2.5k matches | 51919 ms | 59794 ms | 29297 ms |
| Average time taken to find medium frequency pattern- 572 matches | 51347 ms | 60189 ms | 28092 ms |
| Average time taken to find low frequent DNA pattern- 34 matches | 50938 ms | 60892 ms | 27872 ms |
| Average time taken to find pattern with no matches | 50511 ms | 62425 ms | 2735 ms |



From the above graph we can say that Knuth-Morris-Pratt algorithm works the best among all three algorithms. High frequency pattern takes longer time for Naïve string-matching algorithm and Knuth-Morris-Pratt algorithm because for every pattern match the algorithm looks from first to last character of the string, so time complexity increases. Due to large number of hash collisions Rabin-Karp algorithm might have taken more time for high frequency patterns and also because of the same reason Rabin Karp might be working worse than Naïve string-matching algorithm.

## 7. Conclusion

This analysis reviews some typical string-matching algorithms to observe their performance under various conditions and gives an insight into choosing the efficient algorithms. By analyzing these string-matching algorithms, it can be concluded that KMP string matching algorithm is very efficient. KMP decreases the time of searching compared to the Rabin Karp algorithm and Brute Force algorithm. Rabin Karp algorithm gives the same performance or worst performance than brute force as shown in the graphs. It is due to the large number of hash collisions (substring of text having the same hash value as pattern's hash value even though substring is not same as pattern) occurred when running the algorithm. So, Rabin Karp's performance is totally depending on how efficient hash value is generated. In real world KMP algorithm is used in those applications where pattern matching is done in long strings, whose symbols are taken from an alphabet with little cardinality just like test case 2.

## 8. References:

[1]. https://cgi.csc.liv.ac.uk/~michele/TEACHING/COMP309/2005/Lec4.2.4.pdf

[2]. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.

[3]. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

[4]. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

[5]. http://www.gutenberg.org/ebooks/98

[6]. https://ftp.ncbi.nih.gov/genomes/CLUSTERS/