String Matching Implementation and Comparison between Randomized and Deterministic Algorithm

Scribe: Qianhang Sun

1 Introduction

String Matching problem is a quite popular topic nowadays which appears in many areas like network searching, text editing and similarity comparison. This problem can be simplified as: Suppose there is a string S and a pattern P, find if P appears in S and return the position index. Sometimes it could return a list of indices if there are multiple matching there. In this paper, I will apply three algorithms which are Naive, KMP and Rabin-Karp to experiment on various conditions to solve string matching problem.

2 Naive Algorithm

Idea. In the example shows that Naive algorithm basically compares pattern with subtext with the same length from left to right until it finds a match. There is nothing to prepare and it is really easy to think of. The time complexity of this algorithm is $O(nm)$ in average case.

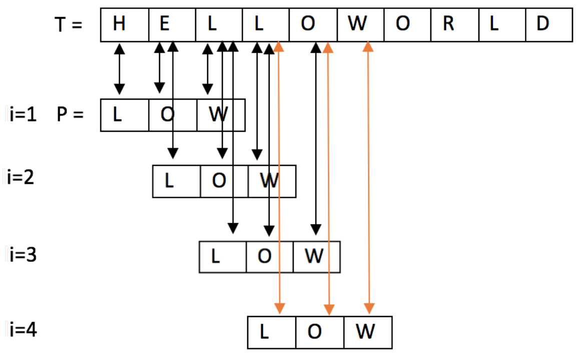


Figure 1: Simple Example of Naive Algorithm

3 Rabin Karp Algorithm

Idea. Since comparing two ”simple” values is much easier than comparing two ”long” strings, if we can change string into corresponding values, then the running time can be reduced and make the whole algorithm become more efficient.

Baby version. According to Rabin&Karp(1987)Efficient randomized pattern-matching algorithms,pp 250-252- Suppose the length of P and T is m and n respectively, $T = \{t_0, t_1, t_2...t_{n-1}\}$, For each character, there exists a ASCII number corresponding to it. So in the following we will use $A(X)$ to represent the ASCII value of X. If we apply a hash function

$$h(T_i) = (A(t_i)k^{m-1} + A(t_{i+1})k^{m-2} + \ldots + A(T_{m+i-1}k^0) mod p_n \quad (1)$$

In this function, k represents the set from which the character of P is generated. For example: $P = "abcde"$, then we can conclude P is generated from lowercase, then $k = 26$. p_n represent a randomly chosen large prime number, it helps shrink the hash value so that it won't overflow the boundary. if we apply this hash function, then the algorithm becomes:

Algorithm 1: BABY VERSION OF RABIN-KARP:

```
1 $x = h(P)$;
2 for each $i = 0, 1, 2, \ldots, n - m + 1$ do
3     Let $T_i = \{t_i, t_{i+1}, t_{i+2}, \ldots, t_{m-1}\}, y_i = h(T_i)$;
4     if $y_i == x$ and $T_i == P$ then
5         return i
6     end
7 end
```

Time Complexity. By applying this function, Line 1 takes $O(m)$; the hash function would take $O(m)$ and since it is in the for loop, overall, it would take $O(mn)$ time complexity. It is even worse comparing with Naive algorithm. To reduce time complexity, we need to think of ways to make hash function consume less time.

Rolling Hash. Luckily, Rabin-Karp really found a much easier way to calculate hash function. Let us begin with a simple example: We have simplified T and P with only numbers, and hash value is the corresponding decimal value.

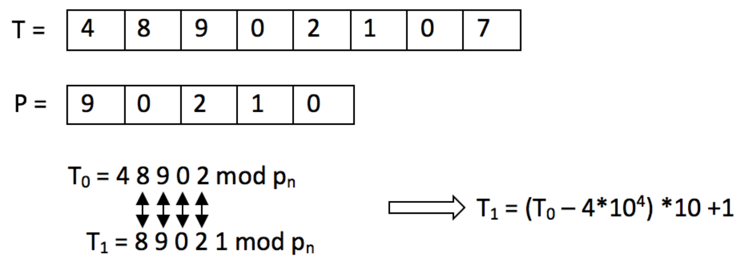


Figure 2: Simple Example of Rolling Hash

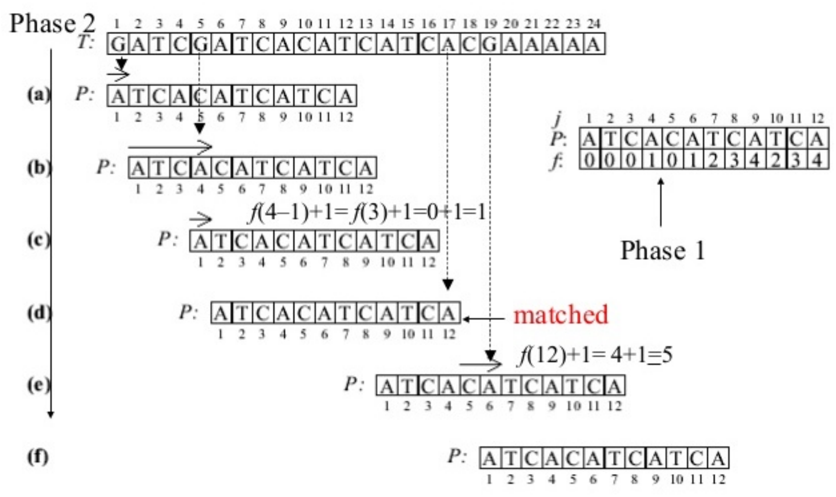
In figure 1, we can do the same calculation to produce T_2, T_3, $T_4 \ldots, T_{n-m+1}$

$$h(T_{i+1}) = (h(T_i) - k^m * \text{first digit of} T_i) * k + \text{next digit after } T_i \quad (2)$$

Then, it will take O(1) to calculate $h(T_{i+1})$ with the help of $h(T_i)$. Overall, the time complexity of Rabin-Karp with Rolling hash is $O(n + m)$. Now we go back to p_n, p_n is chosen randomly from a large prime number. It is chosen relatively large so that there will be less collision in the hash table and helps the algorithm consume less time.

4 KMP Algorithm

Idea. According to Wikipedia, the algorithm was conceived in 1970 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. This was the first linear-time algorithm for string matching. In brief, it firstly create a table like phase 1 in the figure 3, the corresponding value means how many times a certain character has appeared in the previous pattern. With the help of the table, there are two pointers shifting on the pattern and text respectively. The pointer on the text never goes back. Shifting the pattern back and forth and finally find a matching pattern is the main idea of this algorithm. For more detail, see an simple example in the figure 3.



Phase 2
T: GATCGATCACATCATCACGAAAAA
(a)
(b)
(c)
(d)
(e)
(f)
P: ATCACATCATCA
f(4–1)+1=f(3)+1=0+1=1
matched
f(12)+1=4+1=5
j 1 2 3 4 5 6 7 8 9 10 11 12
P: A T C A C A T C A T C A
f: 0 0 0 1 0 1 2 3 4 2 3 4
Phase 1

Figure 3: An example of KMP algorithm. Adapted from Slideshare, Retrieved Dec 9th, 2018, from https://www.slideshare.net/thinkphp/string-kmp, copyright 2014 by thinkphp.

5 Comparison

In the table 1 below, it shows the running time of these three algorithms. It is easy to see, KMP performs the best in theory with time complexity of $O(n)$ overall. Naive performs the worst of these three algorithm. The time complexity of Rabin-Karp which wanders between $O(m)$ and $O(nm)$ basiclly depends on how many collisions in hash table. However, will the experiment result performs the same as the complexity in theory? Let us see in the next section.

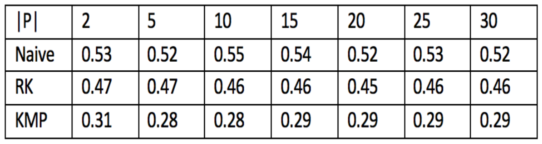
Algorithm	Preprocessing	Running time
Naive	0	O(nm)
KMP	O(m)	O(n)
Rabin-Karp (rolling hash)	O(m)	O(n)~O(nm)

Table 1: Time complexity of These three algorithms

6 Experiment

Pattern length experiment. Suppose the length of text is fixed, we want to know if the length of pattern will influence the running time.

With the help of Random Password Generator, we generate 100 texts, each with 10000 characters arbitrarily chosen from lowercase, uppercase and numbers. Then 100 patterns with length from 2 to 30 are generated with the same generator. After trying 100 times, the running time of these three algorithms are shown in table 2 and figure 3.



|P|	2	5	10	15	20	25	30
Naive	0.53	0.52	0.55	0.54	0.52	0.53	0.52
RK	0.47	0.47	0.46	0.46	0.45	0.46	0.46
KMP	0.31	0.28	0.28	0.29	0.29	0.29	0.29

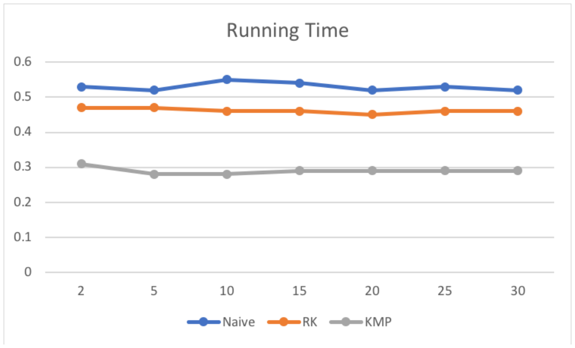
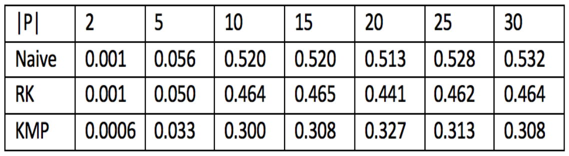


Figure 4: Running time with growing length of text

Analysis. According to Gou, Marc, 2014, Algorithms for String Matching, It seems that KMP is the best and Naive is the worst in the time complexity anlysis. However, it does not follow the idea that as P increases, the running time should increase as well. Was the experiment wrong? Then we found that in the program, we want every pattern to be found, so if only finding the first matching pattern, we wonder if the result will be different.

After modifying code, the result is shown as the following table and figure.



|P|	2	5	10	15	20	25	30
Naive	0.001	0.056	0.520	0.520	0.513	0.528	0.532
RK	0.001	0.050	0.464	0.465	0.441	0.462	0.464
KMP	0.0006	0.033	0.300	0.308	0.327	0.313	0.308

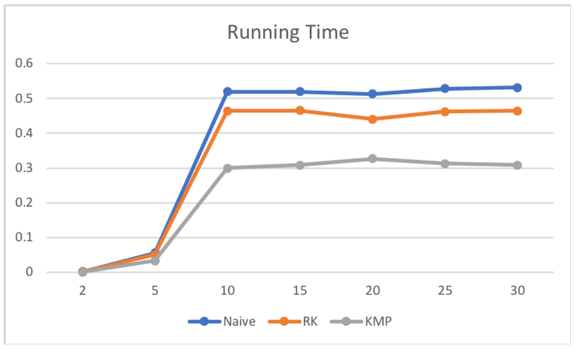


Figure 5: Running time with growing length of text (First match)

Text length experiment. Suppose the length of pattern is fixed, we wonder if the length of text will influence the running time.

This time, we generate 100 patterns with 5 character and text length from 10000 to 400000. The result is shown in the table 3 and also the corresponding figure 4. It follows the theoretical running time.

|T|	10000	50000	100000	150000	200000	250000	300000	350000	400000
Naive	0.52	2.63	5.79	8.13	10.23	13.13	15.47	18.05	20.7
RK	0.47	2.33	4.93	7.00	8.87	11.15	13.56	15.45	17.84
KMP	0.252	1.47	3.06	4.53	5.66	7.16	8.51	9.87	11.22

Table 2: Running time with growing length of text

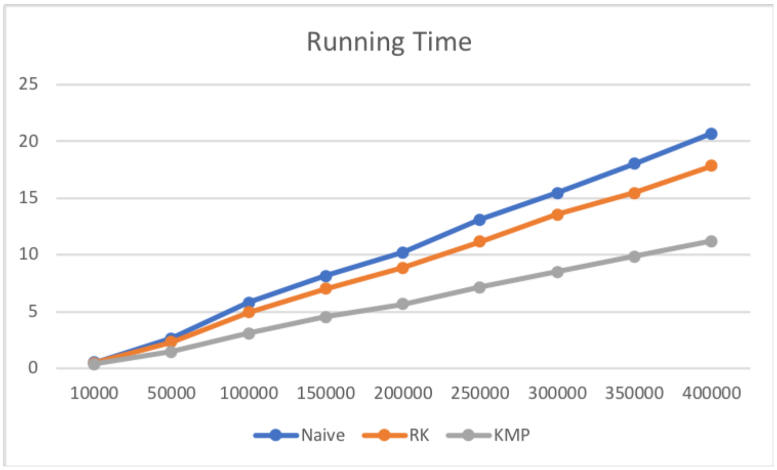


Figure 6: Running time with growing length of text

7 Conclusion

So far we have implemented and compared three different algorithms of string matching. Rabin-Karp algorithm is a Las Vegas randomized algorithm with an average performance, which means it performs good under the easy thought. KMP algorithm perfoms the best but it is really hard to think of and to implement. If the text is not so long and so as the pattern, maybe the best way is to use Naive algorithm.

There are still some more works need to be done: The most efficient algorithm in the worst case; comparing with other string matching algorithm, if their performances are still good enough.

8 Reference

[1]Karp Richard M. & Rabin Michael O.(1987)Efficient randomized pattern-matching algorithms.IBM J. RES. DEVELOP. VOL. 31 NO. 2

[2]Wikipedia contributors. (2018, November 25). Knuth–Morris–Pratt algorithm. In Wikipedia, The Free Encyclopedia. Retrieved December 7, 2018, from
https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt$_a$lgorithm

[3]G, Marc. (July 2014). Algorithms for String matching. Retrieved from
http : //www.student.montefiore.ulg.ac.be/ s091678/files/OHJ2906$_P$*roject.pdf*