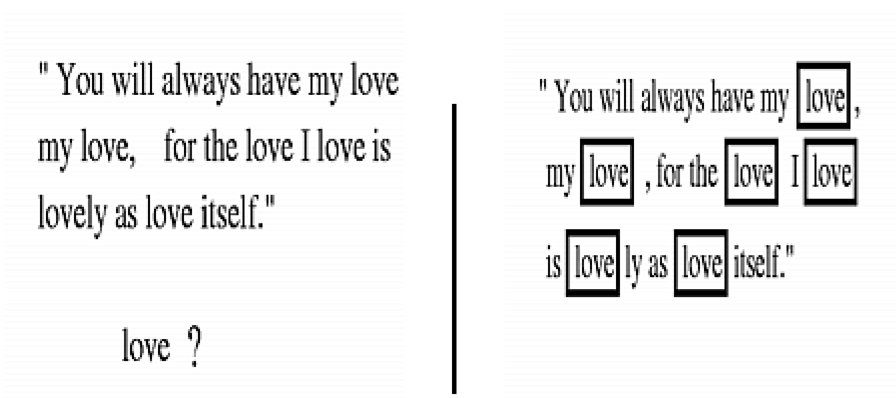
String Matching

Geetha Patil
ID: 312410
Reference: Introduction to Algorithms, by Cormen, Leiserson and Rivest

Introduction:
This paper covers string matching problem and algorithms to solve this problem.
The main algorithms discussed in this paper are Naïve string-matching algorithm, Rabin-Karp Algorithm and Knuth-Morris-Pratt algorithm. Each algorithm is adopted with different method, so different time complexity for each.

Input Description: A text string T of length *n*. A patterns string T of length *m*.

Problem: Find the first (or all) instances of the pattern in the text.



INPUT **OUTPUT**

Related to terms to output

Shift (s): The number of position before the pattern occurrences in text.

Invalid Shift: The position after which partial matching occurs.

Valid Shift: The position after which complete matching occurs.

Naïve string matching algorithm

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

This algorithm works by marking shift positions. After each shift position, it compare all characters of pattern with text and outputs all occurrences if there is exact match.

Example: T = C A R L E T O N U N I V E R S I T Y
P = U N I V E R S I TY

Shift=0
T = C A R L E T O N U N I V E R S I T Y
P = U N I V E R S I TY

Shift=1
T = C A R L E T O N U N I V E R S I T Y
P = U N I V E R S I TY

Shift=8
T = C A R L E T O N U N I V E R S I T Y
P = U N I V E R S I TY

Outputs as pattern found in 9^{th} (s+1) position.

Analysis
This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is **O(n-m+1).**

Rabin-Karp Algorithm

This algorithm performs well in practice and that also generalizes to other algorithms for related problems. Its worst case running time is O (n-m+1) which is same as Naïve string matching algorithm but average case running time is O(n+m).

This algorithm makes use of elementary number-theoretic notions. Each character is converted in to decimal digit and strings of k-consecutive characters are represented as length-k decimal number.

Example : string 31415 is represented as 31,415

Compute decimal value **p** for pattern P[1..m] and $\mathbf{t_s}$ for all sub strings of T[1..n] each of length m .
Comparing p with all ts with p for all occurrences of pattern.

Computing decimal value is by Horner's rule,

For pattern, $p= P[m] * 10^{m-1} + P[m-1] * 10^{m-2} + P[m-2] * 10^{m-3} + \ldots +P[1] * 10^{0}$

Computing p takes O(m).

For pattern, $t_1 = T[s+m] * 10^{m-1} + T[s+m-1] * 10^{m-2} + T[s+m-2] * 10^{m-3} + .. +T[1] * 10^{0}$

The remaining decimal values ts+1,ts+2….can be computed using t_s and all t_{s+1} can be calculated in O(n) constant time by using $t_{s.}$

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

But the only difficulty with this procedure is that p and t_s may be too large to work with conveniently. If P contains m digits (if is too large) then assuming constant time operations on P is unreasonable. Fortunately there is simple cure for this problem as computing p and t_s 's modulo by suitable modulus. The modulus will be chosen as prime number. The computation modulo of p and t_s by modulus takes O(n+m) time.

The fallowing procedure makes these ideas precise.

```
Rabin-Karp-Matcher(T,P,d,q)
n = length(T)
m = length(P)
h = d^(m-1) mod q
p = 0
t0 = 0
for I = 1 to m
      do p = (dp + P[I]) mod q
            t0 = (dt0 + T[I]) mod q
for s = 0 to n-m
      do  if p = =ts
             then if P[1..m] == T[s+1…s+m]
                    then "Pattern occurs with shift " s
             if s< n-m
                 then t_(s+1) = (d(t_s -T[s+1]h) + T[s+m+1]) mod q
```

Analysis:
O(m) is to compute decimal value for pattern and O(n) to compute decimal value for all sub strings each of length m. So, the total complexity is O(n+m).

Knuth-Morris-Pratt algorithm

This is linear time string matching algorithm. This algorithm achieves in O(n+m) running time by avoiding the invalid shifts. Avoiding invalid shift is by knowledge of prefix function, which encapsulates about how pattern matches against itself in shift.

KMP-MATCHER (T,P)

```
1.  n = length (T)
2.  m = length (P)
3.  Π  = COMPUTE-PREFIX-FUNCTION (P)
4.  q = 0
5.  for i = 1 to n
6.      do while q > 0 and  P[q+1] != T[i]
7.           do q = Π [q]
8.         if P[q+1] = T [i]
9.             then q = q+1
10.        if q = m
11.            then  print "Pattern occurs with shift" i-m
12.                  q = Π [q]
```

COMPUTE-PREFIX-FUNCTION (P)

```
1.  m = length(P)
2.  Π [1]  = 0
3.  k = 0
4.  for q = 2 to m
5.      do while k > 0 and  P[k+1] != P[q]
6.           do k = Π [k]
7.         if P[k+1] = P [q]
8.             then k = k+1
9.        Π [q] = k
10. return Π
11.
```

The information that “q” characters of pattern have matched successfully with text determines the immediate invalid shifts.

Example : T = b a c b a b a b a a b c b a b
P = a b a b a c a

The prefix value for each character can be tabulated as fallows.

A	B	A	B	A	C	A
0	0	1	2	3	0	1

When we compare P with T,

b a c b a b a b a a b c b a b

a b a b a c a

We get s = 4 and q (matched characters) =5

Next shift can be, S' = s + (q-k) where k=prefix function for last matched character. In this example s' = 4 + (5-2) = 7 the position could be valid shift. We are avoiding the positions 5^{th} and 6^{th} as invalid by knowing prefix function value.

Analysis

The knuth-Morris-Pratt runs in O(n+m) time. The call of Compute-prefix-function takes O(m) and a similar amortized analysis using the value of q as the potential function, shows the reminder of KMP-algorithm takes O(n) time. So the total taken is O(n+m)

Conclusion

Naïve String Matching	O(n-m+1)m
Rabin-Karp Algorithm	Worst case : O(n –m +1)m Average case : O(n+m)
Knuth-Morris-Pratt Algorithm	O(n+m)