Pattern/String Matching Algorithms

Given a text T[1..n] and a pattern P[1…m],where m≤n, the objective of the problem is to determine whether or not p occurs in T and return the position of P in T.

A pattern p occurs in T with a shift s, if 0≤ s ≤ n-m and T[s+1. . . s+m] = P[1. . .m]. This means T[s+j] = P[j] for all j = 1 to m. A shift s is called a valid shift, if P occurs in T with shift s.

Naive String matching algorithm / Brute Force

This algorithm finds a valid shift that checks P [1 . . . m] = T[s + 1 . . . s + m] for each (n-m+1) possible values of shift s.

```
NaiveStringnMatcher(T,P)
n = length(T)
m =length(P)
for s = 0 to n-m
     j = 1
     While j≤m and  T[s+j] = P[j]
          j = j + 1
     If j = m+1
          Print "Pattern occurs in Text with a shift", s
```

Analysis:

In worst case, Inner while loop takes O(m) times comparing characters in P and T.As there are n-m+1 values for shift for which the characters in T and P matched, the algorithm runs in O((n-m+1)m) time.

Notes: Running time is $O(n^2)$, if $T = a^n$ and $b = a^m$ and $\boldsymbol{m} = \lfloor \boldsymbol{n}/\boldsymbol{2} \rfloor$

Rabin-Karp Algorithm

This algorithm assumes each character as a decimal digit. So a string of k consecutive characters can be viewed as a decimal number of length k.

Given a pattern P[1. . . m] , p denotes its corresponding decimal value and for a text T[1. . .n] , t_s denotes decimal value of m-length substring T[s+1. . . s+m] for s = 0 to n-m. We can say t_s = p if T[s+1. . . s+m] = P[1. . . m]. Thus s is a valid shift.

As the length of the pattern may be large that cannot be fit into a single variable, we can use modulo arithmetic concepts to reduce the size of a number. In this a prime number q can be used to obtain the reduced value of the number.

Example:

T=

3	2	4	3	9	8	1	2	2	1	3	6	1	7	8	3

n = 16

P=

2	2	1	3

m = 4

Solution:

p = 2213

t_0 = 3243

Let take a prime number q =13 and find p % q and t0 % q.

p = 2213 %13 = **3**

t_0 = 3243 %13 = 6
t_1 = 2439 %13 = 8
t_2 = 4398 %13 = 4
t_3 = 3981 %13 = **3** ← Match found but $t_3 \neq$ p .So a **spurious Hit**
t_4 = 9812 %13 = 10
t_5 = 8122 %13 = 10
t_6 = 1221 %13 = 12
t_7 = 2213 %13 = 3 ← **Match found. t_7 = p. So s is a valid shift**
t_8 = 2136 %13 = 4
t_9 = 1361 %13 =
t_{10} = 3617 %13 =
t_{11}= 6178 %13 =
t_{12} = 1783 %13 =

If a match is found at any t_s, then check for t_s = p by applying brute force approach. If they are matched, then P occurs in T with shift ‘s’. If they are not matched (i.e. $t_s \neq$ p) , it is called **spurious hit**.

```
Rabin_Karp_matcher(T,P,d,q)
```

// Given Text T, pattern P, radix of the number d and q is a prime number greater that d.

```
1. n = length(T)
2. m =length(P)
3. h = d^(m-1) mod q
4. p =0
5. t_0 = 0
6. for i = 1 to m
7.        p = (d*p + P[i]) mod q
8.        p = (d*t_0 + T[i]) mod q
9.  for s = 0 to n-m
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-h*T[s+1]) + T[s+1+m] ) mod q
```

Analysis: Processing time in line-6 runs in Θ (m) time to compute p and t_0. To compute remaining values of t (i.e. t_1, t2…), a constant time is required for each t. and altogether for (n-m) shift, Θ (n-m) time is required.

Verifying p and t_s in line-11 requires Θ (m) time in worst case. So in worst case if Θ (n-m+1) spurious hit is occurred, then all verification requires Θ((n-m+1)m) time.

Total running time of the

algorithm is Θ (m) + Θ((n-m+1)m)= Θ((n-m+1)m)

Knuth-Morris-Pratt algorithm for pattern matching

This algorithm was published by Donald Knuth, Vaughan Pratt and James H Morris in 1977. KMP algorithm was the first linear time complexity algorithm for string matching.

The objective of KMP algorithm is to find a pattern (P) in a Text (T).

The algorithm compares characters of P and T from left to right. However, whenever a mismatch occurs, it uses a preprocessed table called "Prefix Table" to skip characters comparison while matching. The prefix table is also known as LPS Table. LPS stands for "Longest proper Prefix which is also Suffix".

Prefix table/LPS Table.

Given a sting S of length n, the prefix function for the string is defined as an array LPS of length n, where LPS[i] is the length of the longest proper prefix of the substring S[1..i] which is also a suffix of this string.

Example

P	a	b	a	b	a	c	a
LPS	0	0	1	2	3	0	1

P	a	b	c	a	b	c	d
LPS	0	0	0	1	2	3	0

```
ComputeLPS(P,m)
1. len = 0
2. LPS[1] = 0
3. i =2
4. while i ≤ m
5.    if p[i] = p[len+1]
6.         len = len+1
7.         LPS[i] = len
8.         i = i +1
9.    else if len != 0
10.             len = LPS[len]
11.        else,LPS[i] = 0
12.                 i = i + 1
13. return LPS
```

Analysis: Construction of LPS table is **preprocessing**. For a pattern of length m, algorithm computes an array of length m. While loop in line number-4 runs in O (m) times.

Matching with LPS Table.

Given a text T[1…n] and pattern P[1…m], compare the characters of P and T for i= 1 to n. When a mismatch occurs, check the value of LPS [i-1] .

If it is 0 , then start comparing the first character of P with next character of T.

If it is not 0, start comparing the characters at the index value equal to the LPS value of previous character of the mismatched character in P with the mismatched character in T.

When there is no mismatch, then continue comparing till the length of P , and output the starting character of T from which there is no mismatch.

```
KMPMatcher(P,T)
1. n = length(T)
2. m = length(P)
3. LPS = computeLPS(P,m)
4. i = 1
5. j = 0
5. while i≤ n
6.   if p[j+1] = T[i]
7.         i = i + 1
8.         j = j + 1
9.         if j = m
10.              print (i-j)
11.              j = LPS [j]
12.  else if j >0
14.              j = LPS [j]
15.        else
16.              i= i +1
```

Analysis:

Algorithm takes O(m) for preprocessing in line-3. Then for matching, each i^{th} character of the text is compared with characters of pattern exactly once except some rare cases. Even in the rare case, the index i never decrements. So the algorithm runs from i= 1 to the complete length of the text which is O(n) times. Total running time = O(m) + O(n) = **O(n)**