String Matching:

In General text editing programs frequently need to find all occurrences of a pattern in the text. Typically the text is a document being edited by the users, and the pattern searched for is a particular word supplied by the user.

Algorithms for such problem known as "string matching" is highly required. These programs are searched the element and edit as per the requirement of users. An efficient algorithms for such problem can greatly aid the responciveness of the text-editing program.

In many applications, string-matching algorithms search particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

Formalization of String-Matching problem.

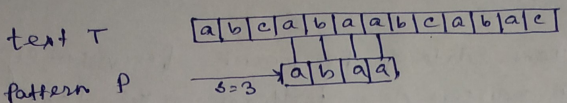
Let us assume that a text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. we may have

$$\Sigma = \{0, 1\}$$

or $$\Sigma = \{a, b, \ldots, z\}.$$

The character arrays P and T are often called strings of character.

Let's discuss with an example.



Refferring to the above figure,

→ the pattern P occurs with shift s in text T, if $0 \leq s \leq n-m$ and $T[s+1..s+m] = P[1..m]$.
(i.e $T[s+j] = P[j]$, $\forall 1 \leq j \leq m$).

→ If P occurs with shift 's' in Text 'T' then we call s a valid shift, otherwise s an invalid shift.

The string matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

In the above fig of string matching problem we want to find all occurrances of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs only once in the text at shift s = 3, which we call a valid shift.

Some Algorithms name with Preprocessing time and Matching time is given below,

Algorithm Name	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$.
Finite-automata	$O(m\|\Sigma\|)$	$\Theta(n)$.

1. Naive String-Matching Algo.

The Naive string-Matching Algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n-m+1$ possible values of s.

Naive String-Matching Algorithm.

```
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 algorithm takes $O((n-m+1)m)$ time for producing the results.

2. Rabin-Karp String Matching Algorithm.

→ The Rabin-Karp string matching algorithm calculate 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 compare the pattern and the M-character subsequence of text.

→ In this way, there is only one comparision per text subsequence, and character matching is only needed when hash values match.

- For expository purpose, let us assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit.
  - → then view a string k consecutive characters as representing a length-k decimal number.
  - → The character string 31415 thus corresponds to the decimal number 31,415.

Given a pattern $P[1..m]$, let p denote its corresponding decimal value. In a similar manner $T[1..n]$, let t_s denote the decimal value of length-m substring $T[s+1..s+m]$

for $s = 0, 1, \ldots, n-m$

Certainly $t_s = p$ if and only if

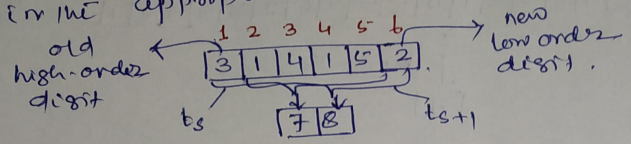
$$T[s+1..s+m] = P[1..m]$$

Thus s is a valid shift if and only if $t_s = p$.

~~Naive algorithm~~

If we could compute p in $\Theta(m)$ time, and all the t_s value in a total of $\Theta(n-m+1)$ time, then we could determine all valid shift s in time $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ by comparing p with each t_s value.

the result by 10 shift the number left by one digit position and adding T[s+m+1] brings in the appropriate low-order digit. For example,



If m = 5, and $t_s = 31415$

then we wish to remove the high-order digit $T[s+1] = 3$ (where s=0), and bring the new low order digit (i.e $T[s+m+1] =$

$\Rightarrow T[0+5+1] =$ ...

$\Rightarrow T[6] = 2$

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

$$= 10(31415 - 10^4(3)) + T[6]$$

$$= 10(1415) + 2$$

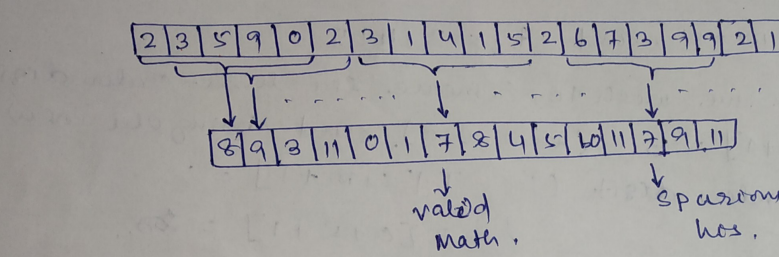
$$= 14152$$

If p and t_s be too large then it is very difficult for calculation. i.e if P contains m characters then we cannot reasonably assume that each arithmatic operation on p takes "constant time". So such problem can be solved by module, mechanism. i.e compute p and t_s values module a suitable modulus q.

For example:

P — 3 1 4 1 5

T — 2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1



For compute P in $\Theta(m)$ time by using Horner's Rule.

i.e $A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\cdots)$

So,

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

Similarly, we compute to form $T[1..m]$ in time $\Theta(m)$.

To compute the remaining value $t_1, t_2, \ldots t_{n-m}$ in $\Theta(n-m)$, we observe that we can compute t_{s+1} from t_s in constant time, since

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

Subtracting $10^{m-1}T[s+1]$ remove the higher-order digit from t_s, multiplying

we can compute p modulo q in $\Theta(m)$ time and all the t_s values modulo q in $\Theta(n-m+1)$ time. If we choose the modulus q as a prime such that 10q just fits within one computer word, then we can perform all the necessary computations with single precision arithmatic.

In general a d-ary (here d=10) alphabet $\{0, 1, \ldots, d-1\}$, we choose q so that dq fits within a computer word and adjust the recurrance equation (i.e t_{s+1} equation) to work modulo q, so that it becomes

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

where $h = d^{m-1} \bmod q$.

```
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.  to = 0
6.  for i = 1 to m
7.      p = (dp + P[i]) mod q
8.      to = (dto + T[i]) mod q
9.  for s = 0 to n-m
10.     if p == ts
11.         if P[1..m] == T[s+1..s+m]
12.             Print "Pattern occurs with shift 's'
13.     if s < n-m
14.         ts+1 = ((d(ts - T[s+1]h) + T[s+m+1]) mod q
```

Here T =

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

P =

1	2	3	4	5
3	1	4	1	5

d = 10 i.e $\{0, 1, \ldots, 9\}$.

q = A prime no. i.e 13.

1. n = T.length = 19
2. m = P.length = 5
3. h = $d^{m-1} \bmod q$ = $10^{5-1} \bmod 13$
   = $10^4 \bmod 13 = 3$.
4. p = 0
5. to = 0.

for step. 6.

i = 1

p = $(dp + P[i]) \bmod q$
= $(10 \times 0 + P[1]) \bmod 13$.
= $(0 + 3) \bmod 13 = 3$

t_0 = $(dt_0 + T[i]) \bmod q$
= $(10 \times 0 + T[1]) \bmod 13$.
= $(0 + 2) \bmod 13 = 2$

so p = 3 and t_0 = 2

i = 2 p = $(dp + P[i]) \bmod q$
= $(10 \times 3 + P[2]) \bmod 13$.
= $(30 + 1) \bmod 13$.
= $31 \bmod 13 = 5$

to = (d to + T[i]) mod q.

= (10 × 2 + T[2]) mod q.

= (20 + 3) mod 13 = 23 mod 13 = 10.

So p = 5 and to = 10.

i = 3.

p = (d p + P[i]) mod q.

= (10 × 5 + 4) mod 13. = 54 mod 13 = 2

to = (d to + T[i]) mod q

= (10 × 10 + 5) mod 13 = 105 mod 13. = 1

So p = 2 and to = 1

i = 4

p = (d p + P[i]) mod q.

= (10 × 2 + 1) mod 13 = 21 mod 13 = 8

to = (d to + P[i]) mod q.

= (10 × 1 + 9) mod 13. = 19 mod 13 = 6

So p = 8 and to = 6.

i = 5

p = (d p + P[i]) mod q

= (10 × 8 + 5) mod 13 = 85 mod 13. = (7)

to = (d to + T[i]) mod 13.

= (10 × 6 + 0) mod 13 = 60 mod 13 = (8).

So p = 7 and to = 8

Hence Similarly execute for the remaining 14. of the Algo. (Lets see one operation on next page.)

9. $s = 0$

if $P = t_s$ False.

i.e $P = t_0$.

13. $s < n - m$

$0 < 14$ True.

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

$\Rightarrow t_{0+1} = ((10(t_0 - T[0+1]3) + T[0+5+1]) \bmod 13$

$= ((10(8 - 2 \cdot 3) + 2) \bmod 13$

$= ((10(8-6)+2)) \bmod 13$.

$= (22 \bmod 13)$.

$= 9$.

So. $\Rightarrow t_1 = t_1 = 9$

Similarly do till same until $s < n - m$.

Rabin-Karp-Matcher takes

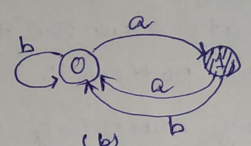
$O(m)$ ← preprocessing time

$O((n-m+1)m)$ ← matching time in worst case.

A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{a, b\}$.

state \ input	a	b
0	1	0
1	0	0

(a)



(b)

(a) A tabular representation of the transaction function δ.

(b) An equivalent state-transition diagram:-
State 1: is the only accepting state. Directed edges represent transitions.

For example:
→ the edge from state 1 to state 0 labeled b indicates that $\delta(1, b) = 0$. This automaton accepts those strings that end in an odd numbers of a's.

Let us define a string matching with finite automata.

T = a b a b a b a c a b a.

P = a b a b a c a

In order to specify the string matching ~~specification~~ automation corresponding to a given pattern $P[1..m]$, we first define an auxillary function σ, called the suffix function corresponding to P.

String matching with finite automata

Many string-matching algorithms build a finite automaton - a simple machine for processing information - that scans the text string T for all occurrences of the pattern P.

These string matching automata are very efficient: they examine each text character exactly once, taking constant time per text character. The matching time used - after preprocessing the pattern to build the automaton - is therefore $\Theta(n)$. The time to build the automaton, however, can be large if Σ is large.

Finite automata

A finite automaton M, illustrated below (*) is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

- Q is a finite set of states, (set of states)
- $q_0 \in Q$ is the start state, (initial state)
- $A \subseteq Q$ is a distinguished set of accepting states. (set of final states)
- Σ is a finite input alphabet. (all i/p alphabets)
- δ is a function from $Q \times \Sigma$ into Q, called the transaction function of M. (transition function)

The function σ maps Σ^* to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x. (Sm.7)

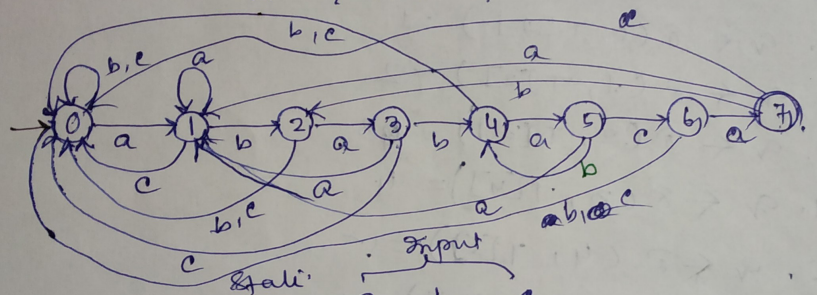
$$\sigma(x) = \max\{k : P_k \sqsupset x\}.$$

T = a b a b a b a c a b a.

P = a b a b a c a.

Total state = 8

$\Sigma = \{a, b, c\}$



State	Input a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

finite Automata or - Matcher (T, δ, m)

1. n = T.length
2. q = 0.
3. for i = 1 to n
4. q = δ(q, T[i])
5. if q == m
6. print "Pattern occurs with shift" i − m.

i = 1 to 11

1. q ← δ(0, T[1]) = 1
2. q ← δ(1, T[2]) = 2
3. q ← δ(2, T[3]) = 3
4. q ← δ(3, T[4]) = 4
5. q ← δ(4, T[5]) = 5
6. q ← δ(5, T[6]) = 4
7. q ← δ(4, T[7]) = 5
8. q ← δ(5, T[8]) = 6.
9. q ← δ(6, T[9]) = 7.

shift = 9 − 7 = 2 position onwards.

Time complexity = O(n).

Solve the following string matching problem with finite automata.

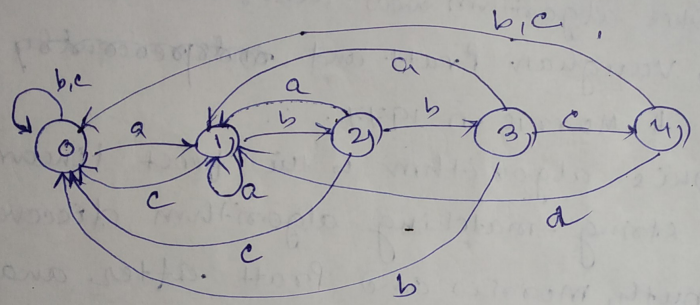
T = a b b a b b c b.

P = a b b c.

Total State = 5 . i.e {0,1,2,3,4}.

$\Sigma = \{a, b, c\}$.

State	i/p a	b	c
0	1	0	0
1	1	2	0
2	1	3	0
3	1	0	4
4	1	0	0



i = 1 to 8.

1. $q = \delta(0, T[1]) = 1$
2. $q = \delta(1, T[2]) = 2$
3. $q = \delta(2, T[3]) = 3$.
4. $q = \delta(3, T[4]) = 1$
5. $q = \delta(1, T[5]) = 2$
6. $q = \delta(2, T[6]) = 3$.
7. $q = \delta(3, T[7]) = 4$

Shift = 7 - 4 = 3rd position onwards.

Knuth-Morris-Pratt algorithm. (KMP Algorithm)

This algorithm avoids computing the transition function δ together, and its matching time is $O(n)$ using an auxiliary function π or LSP (Longest Prefix string), which computed from the pattern in time $O(m)$ and store in an array $\pi[1..m]$.

→ This algorithm was designed by Donald Knuth, Vaughan Pratt and ~~independently by~~ James H. Morris in 1977.

→ This algorithm is the first linear time string-matching algorithm discovered by Knuth, Morris and Pratt after analysis the Naive Algorithm.

→ The basic algorithm, (Naive) does not study the pattern and blindly check every alphabet. By avoiding this waste of information, it achieves a running time $O(m+n)$

→ The implementation of Knuth-Morris-Pratt algorithm is efficient because it minimizes the total number of comparisons of the patter against the input string.

Components of KMP.

→ The prefix function Π.
→ It preprocesses the pattern to find all matches of prefixes of the pattern with the pattern itself.
→ It is defined as the size of the largest Prefix of $P[0..j-1]$ that is also a suffix of $P[1..j]$
→ It also indicates how much of the last comparison can be reused if it fails.
→ It enables avoiding backtracking on the string S.

```
KMP-MATCHER(T, P)
1. n = T.length
2. m = P.length
3. Π = Compute-Prefix-function(P)
4. q = 0
5. for i = 1 to n
6.     while q > 0 and P[q+1] ≠ T[i]
7.         q = Π[q]
8.     if P[q+1] == T[i]
9.         q = q+1
10.    if q == m
11.        Print pattern occurs with shift i-m
12.        q = Π[q]
```

Compute-Prefix-function (P)

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

Let us execute an example with following Text T and pattern P.

T = b a c b a b a b a c a a b

P = a b a b a c a.

First calculate Π (Prefix) for pattern P.

P =

a	b	a	b	a	c	a

Initially m = length[P] = 7

Π[1] = 0

k = 0.

where m ← length of pattern

Π[1] ← Prefix function

k ← initial potential value.

Step-1 $q = 2$ $k = 0$ $\pi[2] = 0$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
π	0	0					

Step-2 $q = 3$, $k = 1$, $\pi[3] = 1$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
π	0	0	1				

Step-3 $q = 4$ $k = 2$, $\pi[4] = 2$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
π	0	0	1	2			

Step-4 $q = 5$ $k = 3$. $\pi[5] = 3$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step-5 $q = 6$ $k = 3 \times 0$ $\pi[6] = 0$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step-6 $q = 7$ $k = 1$ $\pi[7] = 1$

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

now let us execute KMP algorithm for finding wheather pattern 'P' occurs in text 'T'

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

	1	2	3	4	5	6	7
P =	a	b	a	b	a	c	a

Initially $n = T.length = 15$

$m = P.length = 7.$

	1	2	3	4	5	6	7
Π =	0	0	1	2	3	0	1

$q = 0.$

Step 1 $i = 1$ $q = 0$

T = b a c b a b a b a c a a b.

P = a b a b a c a.

P [1] does not match with T[1], hence i incremented by 1

Step 2 $i = 2$ $q = 0.$

T = b a c b a b a b a c a a b

P = a b a b a c a.

Comparing P[1] with T[2] and q is 1

Step 3

$i = 3 \quad q = 1$

T = b a c b a b a b a b a c a a b

P = a b a b a c a

P[1] compare with T[3] Match unsuccessfully

So $q = \Pi[q]$ i.e $q = 0$.

Step-4 $i = 4 \quad q = 0$.

T = b a c b a b a b a b a c a a b,

P = a b a b a c a

Compare P[1] with Ti i.e P[1] with T[4] False.

Hence $q = 0$.

Step-5

$i = 5 \quad q = 0$.

T = b a c b a b a b a b a c a a b

P = a b a b a c a.

Compare P[1] with Ti i.e P[1] with T[5] ⇒ True

So $q = 1$

Step-6 $i = 6 \quad q = 1$

T = b a c b a b a b a b a c a a b.

P = a b a b a c a

Compare P[1] with Ti, i.e P[2] with T[6] → True

So $q = 2$

Step-7 $i = 7 \quad q = 2$

T = b a c b a b a b a b a c a a b

P = a b a b a c a.

Compare P[1] with Ti i.e P[3] with T[7] ⇐ True

So $q = 3$.

Step-8

$i = 8 \quad q = 3$

T = b a c b a b a (b) a b a c a a b,

P = a b a (b) a c a

Compare P_{q+1} with T_i i.e $P_3[4]$ with $T[8]$ True.

So $q = 4$

Step-9 $i = 9 \quad q = 4$

T = b a c b a b a b (a) b a c a a b

P = a b a b (a) c a

Compare P_{q+1} with T_i i.e $P[5]$ with $T[9]$ → True

So $q = 5$

Step-10 $i = 10 \quad q = 5$

T = b a c b a b a b a (b) a c a a b.

P = a b a b a (c) a

Compare P_{q+1} with T_i i.e $P[6]$ with $T[10]$ – false.

hence $q = \Pi[q]$ i.e $q = 3$.

again P_{q+1} compare with T_{10}.

i.e P_4 with T_{10}. – True.

So $q = 4$

Step-11 $i = 11 \quad q = 4$

T = a b a c b a b a b a (b) c a a b,

P = a b a b (a) c a,

Compare P_{q+1} with T_i i.e ($P_3[5]$ with $T[11]$) → True.

$q = 5$

Step-12 $i = 12 \quad q = 5$

T = b a c b a b a b a (c) a a b,

P = a b a b a (c) a,

Compare P_{q+1} with T_i i.e $P[6]$ with $T[12]$ True.

$q = b$

Step-13 $i = 13$ $q = 6$.

T = b a c b a b a b a b a c a a b.

P = a b a b a c a.

Compare P_{q+1} with T i.e P[7] with T[13] → True.

So $q = 7$

Hence $q == m$

So pattern occurs with shift 13 − 7 position or words. i.e. b in position onwards.

and $q = \Pi[q]$ i.e. $q = 1$.

and again search is there any other position where pattern available.

Runtime Analysis.

$O(m)$ ← time required to compute the Prefix function value.

$O(n)$ ← time required to compare the pattern to the text.

Hence total time required to compute the KMP Algorithm is $O(m+n)$.