String Matching Algorithm

In Computer Science, a string matching algorithm (Sometimes called string-searching algorithm or pattern matching algorithm) are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

Text Array $T[1 \ldots\ldots n]$

Pattern Array $P[1 \ldots\ldots m]$

$$m \leq n$$

Elements of 'P' and 'T' can be $\{0, 1, \ldots 9\}$ or $\{a, b, c, \ldots z\}$

Examples:

T

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

P

a	b	a	a

Applications of String Matching Algorithms

(1) Plagiarism Detection
(2) Bioinformatics and DNA sequencing
(3) Digital Forensics
(4) Spelling checker
(5) Spam filters
(6) Search engines
(7) Intrusion Detection System
(8) Content search in large databases.

String Matching Algorithms

(1) Naive Algorithm
(2) Rabin Karp Algorithm
(3) Knuth Morris Pratt (KMP) Algorithm
(4) String matching with Finite Automata.
(5) Boyer Moore Algorithm

Naive String Matching Algorithm

It is the simplest method for pattern matching.

The naive approach tests all the possible placements of pattern $P[1....m]$ relative to text $T[1...n]$.

We try shifts $s = 0, 1, ... n-m$ successively and for each shift s, compare $T[s+1....s+m]$ to $P[1...m]$.

The naive 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 value of s.

Naive - String - Matcher(T, P)

1. $n \leftarrow$ length [T]
2. $m \leftarrow$ length [P]
3. for $s \leftarrow 0$ to $n-m$
4. do if $P[1....m] = T[s+1.....s+m]$
5. then print "Pattern occurs with shift" s

Analysis

Here, for-loop executes for $n-m+1$ (we need at least m characters at the end) times and in iteration we are doing m comparisons.

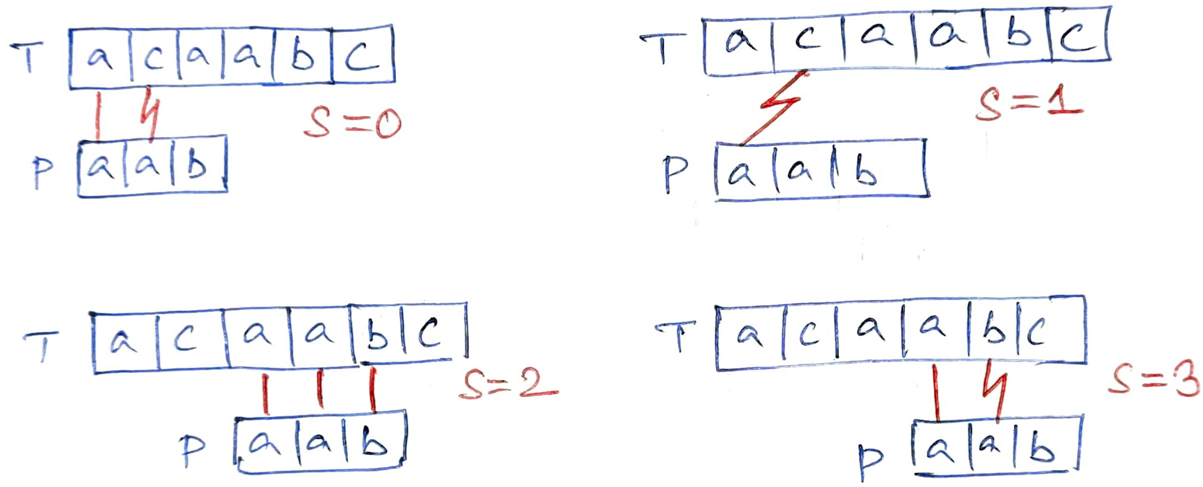
So the total complexity is $O(n-m+1)$.

Example :->

T | a | c | a | a | b | c |

P | a | a | b |

Soln



∴ valid shifts are $S=2$

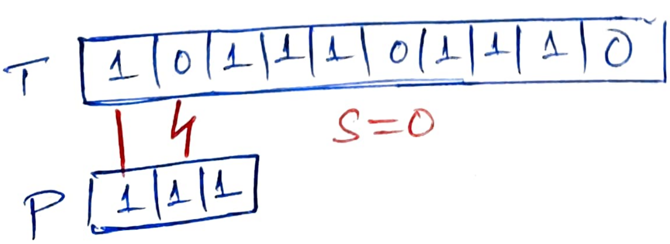
∴ P is found in T at $S=2$.

Example :->

T | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

P | 1 | 1 | 1 |

Soln



T | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

P | 1 | 1 | 1 |

$s = 1$

T | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

P | 1 | 1 | 1 |

$s = 2$

T | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

P | 1 | 1 | 1 |

$s = 3$

T | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

P | 1 | 1 | 1 |

$s = 4$

T | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

P | 1 | 1 | 1 |

$s = 5$

T | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

P | 1 | 1 | 1 |

$s = 6$

T | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

P | 1 | 1 | 1 |

$s = 7$

$\therefore$ Valid shifts are $s=2$ and $s=6$

$\therefore$ P is found in T at $s=2$ and $s=6$.

Rabin-Karp String Matching Algorithm

- Developed by Michael O Rabin and Richard M Karp.
- This algorithm match the pattern in given text using hash Function.

Examples :->

Text (T) = 31234862

Pattern (P) = 234

Hash function used is P mod q, where P is pattern and q is any random prime number.

let q = 13

∴ P mod q = 234 mod 13

= 0
↑
hash code

Note :-> If hash code match, then
- if all characters of 'P' and 'T' matches then it is valid Hit
- if all the characters of 'P' and 'T' does not matches, then it is spurious hit

312 mod 13 = 0 ← Spurious Hit

123 mod 13 = 6

234 mod 13 = 0 ← valid Hit

348 mod 13 = 10

486 mod 13 = 5

862 mod 13 = 4

Q T = 235902314152673992 1

P = 31415

q = 13

Soln P mod q = 31415 mod 13

= 7

23590 mod 13 = 8

35902 mod 13 = 9

59023 mod 13 = 3

90231 mod 13 = 11

02314 mod 13 = 0

23141 mod 13 = 1

31415 mod 13 = 7 ← valid Hit

14152 mod 13 = 8

41526 mod 13 = 4

15267 mod 13 = 5

52673 mod 13 = 10

26739 mod 13 = 11

67399 mod 13 = 7 ← Spurious Hit

73992 mod 13 = 9

39921 mod 13 = 11

Q $T = 31415926535897 93$

$P = 26$

$q = 11$

Soln $p \bmod q = 26 \bmod 11$

$= 4$

$\therefore$ $31 \bmod 11 = 9$

$14 \bmod 11 = 3$

$41 \bmod 11 = 8$

$15 \bmod 11 = 4$ ← Spurious Hit

$59 \bmod 11 = 4$ ← Spurious Hit

$92 \bmod 11 = 4$ ← Spurious Hit

$26 \bmod 11 = 4$ ← Valid Hit

$65 \bmod 11 = 10$

$53 \bmod 11 = 9$

$35 \bmod 11 = 2$

$58 \bmod 11 = 3$

$89 \bmod 11 = 1$

$97 \bmod 11 = 9$

$79 \bmod 11 = 2$

$93 \bmod 11 = 5$

Rabin - Karp (T, P)

n = T. length

m = P. length

h^P = Hash (P[...]) i,e P mod q

h^T = Hash (T[...]) i,e T mod q

for $\overset{\text{shift}}{s}$ = 0 to $n-m$

  if ($h^P == h^T$)

    if $(\overbrace{P[0 \ldots m-1] == T[s+0, \ldots s+m-1]}^{\text{matching all characters of P and T}})$

      Print "Pattern found with shift" s

  if ($s < n-m$)

    h^T = Hash (T(s+1 ... s+m))

Time Complexity

If no spurious hit, then T.C = O(n-m+1)

If there is spurious hit, then T.C= O(mn)

Knuth - Morris - Pratt Algorithm

- KMP is a linear time string matching algorithm.
- KMP uses concept of prefix and suffix for generation of π table.
- Worst case running time of algorithm is $O(m+n)$

Let Pattern : a b c d a b c

∴ Prefix can be a, ab, abc, abcd, ...

Suffix can be c, bc, abc, dabc, ...

Here, abc is the prefix same as suffix.

How to find π table (or) Prefix function (or) Prefix Table

Pattern : a b c d a b c a

Solution :→

1	2	3	4	5	6	7	8
a	b	c	d	a	b	c	a
0	0	0	0	1	2	3	1

Pattern : a a b a a b a a a

Solution :→

1	2	3	4	5	6	7	8	9
a	a	b	a	a	b	a	a	a
0	1	0	1	2	3	4	5	2

Pattern : a b c d a b e a b f

Solution :→

1	2	3	4	5	6	7	8	9	10
a	b	c	d	a	b	e	a	b	f
0	0	0	0	1	2	0	1	2	0

	0	1	2	3	4	5	6	7	8	9	10
Pattern:	a	a	b	a	a	c	a	a	b	a	a
	0	1	0	1	2	0	1	2	3	4	5

	0	1	2	3	4	5	6	7	8	9
Pattern:	a	a	a	c	a	a	a	a	a	c
	0	1	2	0	1	2	3	3	3	4

Initially j is at 0 and i is at 1.

[j==0] if P[j] ≠ P[i], then put 0 and increment i by 1.

if P[j] == P[i], then put j+1 and increment both i and j by 1.

[j≠0] if P[j] ≠ P[i], then check value at (j−1) character and place j there and compare with P[i] again, now if matches, then put j+1 and increment both i and j by 1

else repeat again by placing j at (j−1) character value and compare.

Que

Pattern: a c a c a b a c a c a b a c a c a c

~~λ = 0 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10~~

λ = 0 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11 4

Example :

String : a b a b c a b c a b a b a b d

Pattern : a b a b d

Soln :

	i														
String: (T)	a	b	a	b	c	a	b	c	a	b	a	b	a	b	d
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

	j					
	0	1	2	3	4	5
(P) pattern:		a	b	a	b	d
π		0	0	1	2	0

compare T(i) with P(j+1)

↳ if matches then increment i and j by 1.

↳ if does not matches then move j to the index given under current j position (π value). and again compare P(j+1) with T(i) else ~~and~~ increment i.

	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i
String:	a	b	a	b	c	a	b	c	a	b	a	b	a	b	d
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

	j j	j	j j	j	j	j
	0	1	2	3	4	5
pattern:		a	b	a	b	d
π		0	0	1	2	0

Pattern found in string.

Time Complexity = $O(m+n)$

m → π table, n → Search

Note:->

The implementation of KMP algorithm is efficient because it minimizes the total number of comparisons of the pattern against the input string.

Compute-Prefix-Function (P)

```
m ← length [P]
π[1] ← 0
k ← 0
for q ← 2 to m
    while k > 0 and P[k+1] ≠ P[q]
        k ← π[k]
    if P[k+1] = P[q]
        k ← k+1
    π[q] ← k
return π
```

Example : Pattern : a b a b a c a

$m = \text{length}[P] = 7$

$\pi[1] = 0$

$k = 0$

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

Step 1: $q = 2, k = 0$

$\pi[2] = 0$

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

Step 2: $q = 3, k = 0$ (here $P[1] == P[3]$)
So, $k = 1$

$\pi[3] = 1$

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

Step 3: $q = 4, k = 1$ (here $P[2] \neq P[4]$ is False)
also $P[2] == P[4]$
$\Rightarrow k = 2$

$\therefore \pi[4] = 2$

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

Step 4: $q = 5, k = 2$ ($\underset{(T)}{2 > 0}$ and $\underset{(F)}{P[3] \neq P[5]}$)
but $P[3] == P[5]$
$\Rightarrow k = 3$

$\therefore \pi[5] = 3$

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

Step 5 : $q = 6$, $k = 3$

here, $3 > 0$ and $P[4] \neq P[6]$
(T) (T)

$\Rightarrow k = \pi[3]$

$k = 1$

$\therefore \pi[6] = 1$

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

Step 6 : $q = 7$, $k = 1$

here, $1 > 0$ and $P[2] \neq P[7]$
(T) (T)

$\Rightarrow k = \pi[1]$

$k = 0$

Now, $P[1] == P[7]$ is True

$\therefore k = k + 1$
$= 1$

$\pi[7] = 1$

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

The running time of prefix function is $O(m)$.

KMP-Matcher (T, P)

```
n ← length [T]
m ← length [P]
π ← Compute - Poefix - Function (P)
q ← 0
for i ← 1 to n
        while q > 0 and P[q+1] ≠ T[i]
                q ← π[q]
        if P[q+1] = T [i]
                q ← q + 1
if q = m
        Poenl "Pattern occurs with shift" i-m
        q ← π [q]
```

Q Compute the prefix function π for the pattern ababbabbabbababbabb.

Soln $m = \text{length } [P] = 19$

$\pi[1] = 0$

$K = 0$

Step1: $q = 2, K = 0$

here, $P[1] == P[2]$ (False)

$\therefore \pi[2] = 0$

p:	a	b	a	b	b	a	b	b	a	b	b	a	b	a	b	b	a	b	b
q:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
π:	0	0																	

After Compute-Prefix-Function (P) for q values 2 to 19, we have

P:	a	b	a	b	b	a	b	b	a	b	b	a	b	a	b	b	a	b	b
q:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
π:	0	0	1	2	0	1	2	0	1	2	0	1	2	3	4	5	6	7	8.

Q Apply KMP algorithm.

String: bacbabababacaab

Pattern: ababaca

Soln String: bacbab$\underline{\text{ababaca}}$ab

Pattern: $\underline{\text{ababaca}}$

The total number of shifts that took place for the match are $= i - m$

$= 13 - 7$

$= 6$ shifts.

Note:-> The KMP algorithm never needs to move backwards in the input text T. It makes the algorithm good for processing very large files.