Design and analysis of Algorithms

(Prof. Parminder Kaur Wadhwa)

String Matching Algorithms

Part A :- includes :-

1. Simple/Naive Algorithm
2. Brute-Force Algorithm
3. Rabin-Karp Algorithm.

(1 and 2 are the same)

Part B :- includes :-

4. Boyer-Moore Algorithm
5. Knuth-Morris-Pratt Algorithm
6. Applications of string matching algorithms.

Date
Page

Design and Analysis of Algorithms

(Prof. Parminder Kaur Wadhwa)

String Matching Algorithms (Part-B)

Topics covered :-

4. Boyer-Moore Algorithm
5. Knuth-Morris-Pratt Algorithm
6. Applications of String Matching algorithms.

4. Boyer-Moore Algorithm

4.1 The Basic Boyer-Moore Algorithm

[involving :-
1. Looking-glass heuristic
2. character-jump heuristic (or Bad-Character Rule)]

4.2. Boyer-Moore Algorithm using Good-Suffix Rule (along with looking-glass heuristic)

4.3. Boyer-Moore Algorithm combining all the three components together :-

1. looking-glass heuristic
2. Bad Character Rule
3. Good-Suffix Rule

4.4. Boyer-Moore Horspool Algorithm

4.1. The Basic Boyer-Moore Pattern Matching algorithm

Boyer-Moore algorithm learn from the past character comparisons and skip the unnecessary comparisons in future.

It works on the following two heuristics:-

1. Looking glass heuristic
2. Character-jump heuristic (Bad-character rule)

1. Looking glass heuristic :-

It says that, when testing a possible placement of P against T, begin the comparison from the end of P and move backward to the front of P.

In other words, we can say that

"Try alignments in 'left-to-right' order,

and

try character-comparisons in 'right-to-left' order."

For example:-

(LEFT) (RIGHT)

T: f o x m e f o r

P: f o r ← Compare x and r first (comparisons will be started from right-to-left)

→ Pattern will move from Left-to-right for alignment

This is opposite to naive or simple or brute force pattern matching algorithm.

Boyer-Moore Algorithm

Left Right

T: f o x m e f o r

P: f o r ← I st comparison (from Right to left)

Naive / Simple / Brute Force Algorithm

(Left) (Right)

T: f o x m e f o r

P: f o r

I st comparison (from left to right)

2. Character-jump heuristic (or Bad-Character Rule)

Let T is the text given
P is the pattern to be matched

T[i] = c , a character at ith index due to which mismatch has occured.

The Bad-character Rule says that:-

Upon mismatch, skip alignments/placements until:-

(a) mismatch becomes a match

Or

(b) pattern P moves past mismatched character

Thus, the character-jump heuristic says that:-

During the testing of a possible placement of P against T, a mismatch of test character T[i] = c with the corresponding pattern character P[j] is handled as follows:-

If c is not contained anywhere in P, then shift P completely past

T[i] (because it cannot match any character in P).

Otherwise, shift P until an occurrence of character C in P gets aligned with T[i], (making the mismatch a match).

The actual search for the pattern, in basic Boyer-Moore Algorithm takes $O(nm)$ time in the worst case,

(where $n \rightarrow$ is the size of the given text T

$m \rightarrow$ is the size of the pattern P to be matched/found)

Example :- of :-

The Basic Boyer - Moore Algorithm for Pattern Matching

T: f o x m e f o r

P: f o r

1. T: f o x m e f o r
   P: f o r — 1st comparison
   ✗ mismatch, 'x' is not required
   So skipping 'x' all together
   (otherwise, it will go on creating mismatches).

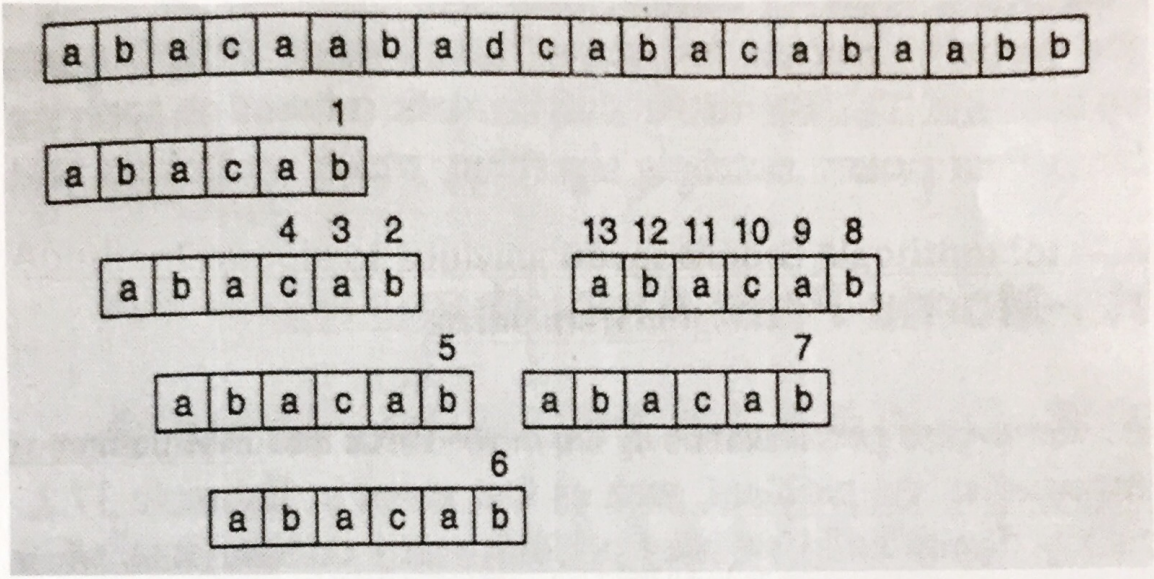
2. T: f o x m e f o r
   P: f o r — 2nd comparison
   ✗ mismatch, but 'f' is required
   So, aligning 'f' of text T with 'f' of pattern P,
   (thus, making the mismatch, a match)

3. T: f o x m e f o r
   P: f o r
   5th ✓ 4th ✓ 3rd ✓ comparisons are successful.
   (MATCH FOUND)

(BASIC)

Another Example of applying Boyer Moore Algorithm for string matching

To find pattern: a b a c a b from the given text



Due to mismatch in **comparison 1**, due to 'a' , both 'a' of text and pattern are aligned.

Comparisons 2nd and 3rd are successful.

4th comparison is a mismatch, due to 'a'. The immediate unchecked 'a' of pattern is aligned with 'a' of pattern p.

5th comparison is a mismatch, due to a. The immediate unchecked 'a' of pattern is aligned with 'a' of pattern p.

6th comparison is a mismatch, due to 'd'. But 'd' is not required. So, cross 'd' otherwise, it will go on creating mismatches.

7th comparison is a mismatch due to a . The immediate unchecked 'a' of pattern is aligned with 'a' of pattern p.

The comparisons **8th, 9th, 10th, 11th , 12th and 13th** are successful and the pattern is matched in given text .

Applying Bad-character rule (Boyer-Moore Algorithm) for DNA pattern Matching :-

DNA is Deoxyribonucleic acid

It is the molecule that carries genetic information for the development and functioning of an organism. It is a molecule that supplies the genetic instructions that tell living creatures how to develop, live and reproduce. DNA can be found inside every cell and is passed down from parents to their offsprings.

DNA is made of two linked strands that wind around each other to resemble a twisted ladder — a shape known as double helix.

Each strand has a backbone made of alternating sugar (deoxyribose) and phosphate groups.

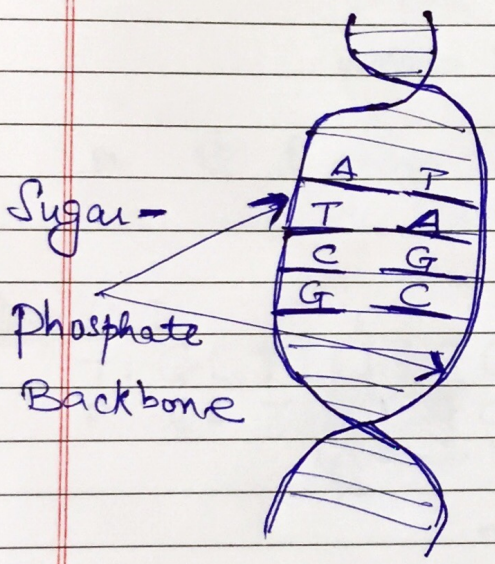
Attached to each sugar is one of four bases :-

1. Adenine (A)
2. Cytosine (C)
3. Guanine (G)
4. Thymine (T)

The two strands are connected by chemical bonds between the bases:-

adenine bonds with thymine (A bonds with T)

cytosine bonds with guanine (C with G)



Deoxyribonucleic Acid (DNA)

The sequence of the bases along DNA's backbone encodes biological information, such as the instructions for making a protein or RNA molecule.

The DNA pattern matching, helps the scientists to find the locations of particular DNA subsequences in a DNA sequence.

Applying Boyer Moore : Bad character Rule for DNA pattern matching

The bad character rule is:- upon mismatch :-
skip alignments until :-

(a) mismatch becomes a match
(align the characters of text T and pattern P that has caused mismatch)

or (b) P moves past mismatched character
(i.e. cross the character altogether)

Step 1:-

T: G C T T C T G C T A C C T T T T G C G C G C
P: C C T T T G C
4 3 2 1
×

4th comparison is mismatch due to 'C'. Align nearest 'C' of Text and 'C' of P.

Step 2:- T: G C T T C T G C T A C C T T T T G C G C G C
P: C C T T T T G C
6 5
×

6th comparison is mismatch due to 'A'. But it is not required. So, cross it altogether.

Step 3: T: G C T T C T G C T A C C T T T T G C G C G C
P: C C T T T T G C
14 13 12 11 10 9 8 7

from 7th to 14th, all comparisons are successful. [match is found]

4.2. **Boyer-Moore Algorithm using good-suffix rule (along with looking glass heuristic)**

The bad-character rule says that upon mismatch we want to shift P such that the mismatch becomes a match

but,

the good-suffix rule (or good-character rule) says that for the characters we did match, we want to shift P such that we don't turn any of those matches into a mismatch at the end of that shift.

Bad-character Rule	Good-character rule (or good-suffix rule)
Upon mismatch, shift P such that mismatch becomes a match	Upon mismatch, shift P such that old matches done should not become new mismatches
i.e. (try to convert a mismatch into a match)	i.e. (try to maintain matches already done)

Boyer-Moore: Good Suffix Rule

Let t = substring matched by inner loop;

Skip until:-

(a) there are no mismatches between P and t.

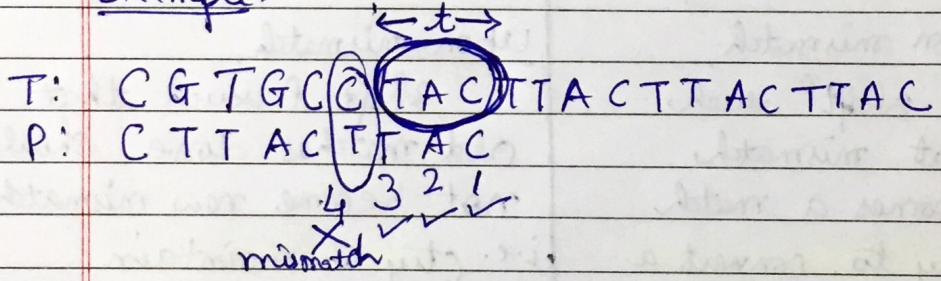
(i.e. Old matches are still maintained to maximum extent).

Or

(b) P moves past t.

(cross altogether)

Example:-



try to maintain matches already done.
Don't make old matches as new mismatches.

Applying Boyer-Moore – Good Suffix Rule for DNA pattern matching

Step 1:

```
          t
T: CGTGCCTACTTACTTACTTACTTAC
P: CTTACTTAC
   4 3 2 1
   X ✓ ✓ ✓
```

Try to look occurence of TAC i.e. 't' already matched in pattern P (on left side) and align them.

Step 2:-

```
             t
T: CGTGCCTACTTACTTACTTACTTAC
P:    CTTACTTAC
      12 11 10 9 8 7 6 5
      X  ✓  ✓  ✓ ✓ ✓ ✓
```

12th comparison has caused mismatch.
But 5th to 11th comparisons are successful.
Try to find occurence of already matched character sequence on left of P.

```
  X
C TACTTAC
CTTACTTAC
```

The sequence "CTTAC" should align with each other to maintain old matches to their maximum level.

Step 3:-

```
T: CGTGCCTACTTACTTACTTACTTAC
P:       CTTACTTAC
         21 20 19 18 17 16 15 14 13
         ✓  ✓  ✓  ✓  ✓  ✓  ✓  ✓  ✓
```

Comparisons from 13th to 21st are successful.
(MATCH FOUND)

4.3 Boyer-Moore Algorithm combining all the three components together :—

1. Looking glass heuristics (compare from Right side)
2. Bad-character Rule
3. Good-Suffix Rule

In practice, both rules :-
bad-character rule and good-suffix rule can be used together.

Whenever, we have a mismatch, we can try both rules. Each rule will tell us some amount that we can shift the pattern P, in other words :-

Some number of alignments/placements that we can skip (i.e. to avoid comparisons which are unnecessary), and
we are simply going to take the maximum of the two.

i.e. Upon mismatch :-

Calculate :- how many alignments are skipped by both rules.
The rule which gives the maximum number of alignments/placements is chosen to apply for that particular mismatch, since it is avoiding more comparisons than other.

Applying Boyer-Moore Algorithm supporting both rules → Bad-character rule, Good-Suffix rule for DNA pattern matching

Step 1 :-

T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG

mismatch

Since, no character is matched, the good suffix rule is not applicable here as there is no matching characters. So, we cannot use the good-suffix rule.

We have to use the bad-character rule. So, 'T' has caused the mismatch.

But 'T' is required. So, aligning 'T' of text with 'T' of pattern P, (making a mismatch as a match)

Step 2 :-

T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
1 2 3 4 5 6

If we align 'T', and make the mismatch as a match, we are skipping 6 alignments (or comparisons).

On the other hand, good-suffix rule, has no matching characters and thus, nothing is there to maintain (and

is not applicable).

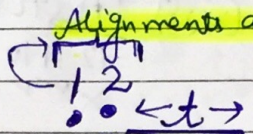
i.e. $bc = 6$, $gs = 0$

(bc ⇒ bad-character
gs ⇒ good suffix)

Here, $bc > gs$.
So, Bad-character rule is applied.

Step 2:-

Alignments are skipped (by gs)



T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
5 4 3 2
X

If bad-character rule is applied, i.e. 'C' which has caused mismatch are aligned (of text T and pattern P)
⇒ then, no alignment (or comparison) is avoided or skipped.

But, if we apply good-suffix rule:-

GCG is already matched. We need to find the occurence of GCG on the left of P (to their maximum level) and GCG another ~~exists~~ occurence exists.

If we align GCG of T and P, then, it skips 2 alignments.

$\therefore$ bc = 0 , gs = 2

gs > bc

$\therefore$ good-suffix rule is chosen to apply for this mismatch.

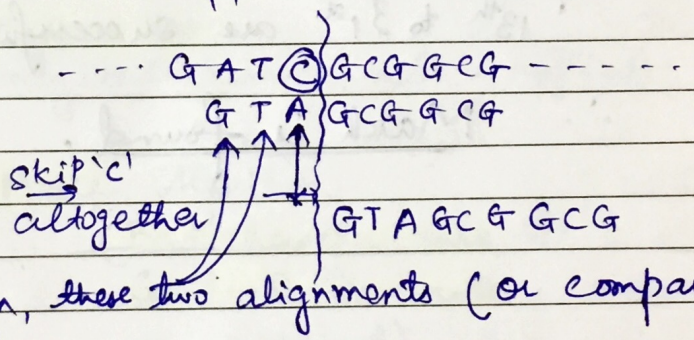
Step 3:-

T: G TT ATA GCTGATC GCG GCG TA GCGG CG AA
p: GTA GCGGCG
12 11 10 9 8 7 6
✗ ✓ ✓ ✓ ✓ ✓

12th comparison is a mismatch, due to 'c'.

If bad-character-rule is applied, then 'c' is not required and we need to skip 'c' altogether.

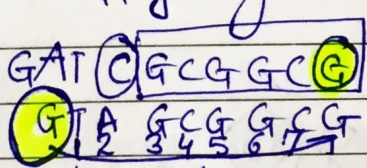
If bc rule is applied then:-



Then, these two alignments (or comparisons) are skipped.

$\therefore$ bc = 2

But if, we apply good-suffix rule, then:-



'G' can be maintained as match. Leading to

skipping of 7 alignments.

$\therefore gs = 7$

Now, $bc = 2$, $gs = 7$

$gs > bc$ $\therefore$ good-suffix-rule is chosen to be applied for this mismatch.

$\therefore$

Step 4 :-

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A

G T A G G G G C G

21 20 19 18 17 16 15 14 13

All comparisons from :-
13^{th} to 21^{st} are successful.

$\therefore$ Match is found.

The worst-case running time of basic Boyer-Moore algorithm [(using bad-character-rule) only] is

$$O(nm)$$

But, the Boyer-Moore algorithm achieves running time

$$= O(n+m)$$

if the good-suffix-rule is also used (in combination with bad-character-rule).

Brute-force Pattern Matching Algo $= O(nm)$

Rabin-Karp " " " $= O(n+m)$

Basic Boyer-Moore " " " $= O(nm)$
(Only bad-character-rule is applied)

Boyer Moore Algorithm $= O(n+m)$
(Bad-character-rule & Good-Suffix-rule are applied)

- Brute force algorithm is good for smaller texts and small patterns.
- Rabin-Karp Algo is good when it is used for
  multiple-pattern matching
  (i.e. finding all occurences of a set of patterns in a fixed text).

- Boyer-Moore Algorithm (using bad-character-rule & good-suffix rule) is good for
  single-pattern matching.

  Boyer-Moore algorithm is extremely fast on large texts (relative to the length of the pattern).

An attempt has been made by the researchers to have a variant of Boyer-Moore algorithm (involving both rules):-
in order to achieve an average time complexity of:-

$O(n)$

which is linear, ($n \rightarrow$ size of text T) and is independent of:-
size of pattern m. This algorithm is Boyer-Moore Horspool Algorithm.

(4.4) Boyer-Moore Horspool Algorithm

In 1980, Nigel Horspool (a retired professor of computer science, of the University of Victoria, Canada) invented a variant of Boyer-Moore Algorithm, which is called as:- (or known as):-

Boyer-Moore Horspool Algorithm.

[Note:- a) Boyer-Moore algorithm was developed by Robert S. Boyer (professor of computer science at University of Texas, US)

b) and J. Strother Moore (Computer Scientist, at University of Texas, US)

They developed Boyer-Moore Algorithm in the year 1977].

The Boyer-Moore Horspool algorithm, was able to give the average time complexity

$$\doteq O(n)$$

which is linear & is independent of the size of the pattern m.

But, its worst case is $O(nm)$ which is same as that of brute-force algorithm.

Boyer Moore Horspool Algorithm	Boyer-Moore Algorithm (using bad-character & good-suffix rules)
Average-case time complexity = $O(n)$	Average case time complexity = $O(n+m)$
Worst case time complexity = $O(nm)$	Worst case time complexity = $O(n+m)$

The explanation of Boyer-Moore Horspool Algorithm

- It includes: the step of:- making a 'Bad Match Table'
- Compare pattern to text, starting from rightmost character in the pattern
- If mismatch, move pattern forward corresponding to value in the Bad Match Table.

For example:- (Applying Boyer-Moore Horspool Algorithm).

T: t r u s t h a r d t o o t h b r u s h e s
P: t o o t h

Constructing Bad Match Table:- (use the formula)

Value = length - index - 1
and → (keep last letter's value = length)
and → (and every other letter = length)

T O O T H Length = 5 (Total characters in P)
Index = 0 1 2 3 4

Step 1

Letter	T	O	H	* (Any other)
Value				

We need to fill this table. There are three different letters in pattern P: "Tooth". These are T, O and H.

Step 2:-

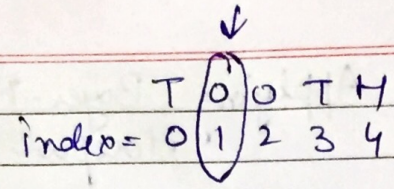
for letter T ⇒ length - index - 1
= 5 - 0 - 1
= 4.

(0 is the index of T)

T O O T H
0 1 2 3 4

Letter	T	O	H	*
Value	4			

Step 3:-



For letter O at index 1

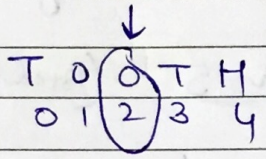
$\Rightarrow$ length - Index - 1

$= 5 - 1 - 1$

$= 3$

letter	T	O	H	*
value	4	3		

Step 4:-



For letter O at index 2

$\Rightarrow$ length - index - 1

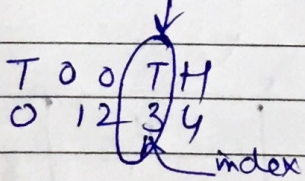
$= 5 - 2 - 1$

$= 2$

Updating the value for 'O' in the Bad-Match-Table as follows:-

letter	T	O	H	*
value	4	2		

Step 5:-



for letter T at index 3 $\Rightarrow$ length - index - 1

$= 5 - 3 - 1$

$= 1$

Updating the value for letter 'T' in the Bad Match Table as follows:-

Letter	T	O	H	*
Value	1	2		

Step 6:-

T O O T H ← Last letter
0 1 2 3 4

Keep the value for the LAST LETTER as length i.e. = 5

Letter	T	O	H	*
Value	1	2	5	

Last letter's value = length

Step 7:-

for "any other letter" i.e. *, keep the value = length = 5.

Letter	T	O	H	*
Value	1	2	5	5

So, this is the Bad Match Table that will be used when, Boyer-Moore Horspool algorithm will be applied.

Proceeding with the algorithm; →

Bad Match Table :-

Letter	T	O	H	*
Value	1	2	5	5

Step 1

T: TRUST HARD TOOTH BRUSHES
P: TOOTH
1st ⇒ comparison
× mismatch

The mismatch is due to 'T'. Look the value for letter 'T' in the Bad Match Table :-
The value = 1 (for letter T)

So, shift the pattern by '1' character.

Step 2 :-

× ✓ ✓

T: TRUST HARD TOOTH BRUSHES
P: TOOTH
4th 3rd 2nd comparison
× ✓ ✓

4th comparison is a mismatch.

for this alignment we, compared "H" in the text first.

TRUST HARD. . .
TOOTH ← This was the first comparison done in this particular alignment

(But, further it led to a mismatch)

So, we will look for the value of letter H in the table, i.e. 5

[Or, you may look * in the table, as 3 does not exist]

So, the pattern will be shifted by 5 characters.

Step 3:-

T: TRUST HARD TOOTH BRUSHES

P: 1 2 3 4 5 TOOTH

5th comparison is a mismatch due to O

Look 'O' in the table. Value for 'O' is 2

So, shift the pattern by 2.

Step 4:-

T: TRUST HARD TOOTH BRUSHES

P: 1 2 TOOTH

6th comparison is a mismatch due to T

Look for value of letter 'T' in the table.

It is 1.

So, shift the pattern by 1.

Step 5:-

T: TRUST HARD TOOTH BRUSHES

P: 1 TOOTH

11 10 9 8 7 ✓ ✓ ✓ ✓ ✓

all comparisons are successful.

(Match is found)

5. Knuth - Morris - Pratt Algorithm

(KMP) algorithm

The KMP algorithm, avoids the waste of information in case a pattern character does not match in the text. Thus, it achieves a running time of O(n+m).

This is the worst case time complexity which is acceptable as,

even in the worst case, any pattern matching algorithm with this time complexity is examining all the characters of the text and all the characters of the pattern at least once.

The main idea of the KMP algorithm is to preprocess the pattern string P so as to compute a failure function f, that indicates the proper shift of P so that, to the largest extent possible, we can reuse previously performed comparisons.

The algorithm for calculating prefix function is as follows:—

Given pattern

Compute-Prefix-function (p)

{ m is length of p

$\pi(1) = 0$ (Assign π of first character as 0)

k = 0

for q ← 2 to m do

{ while (k > 0 and p[k+1] ≠ p[q]) do

k ← π(k); // k when not matching

if (p[k+1] = p[q]) then

{ k = k+1 } // increment k when matching

π[q] = k

return π

}

}

Applying KMP algorithm for given text T and pattern P

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

P = a b a c a b , m = size of pattern = 6

First we need to preprocess the pattern P:-

P ⇒	P[1]	P[2]	P[3]	P[4]	P[5]	P[6]
	a	b	a	c	a	b

Let $\Pi[1] = 0$ (for $q = 1$ $\Pi[q] = \Pi[1]$ ⇒ is set as zero)

and $k = 0$

k	q	P[k+1]	P[q]	new k	Π[q]
0	(1)	----	P[1] = a.		Π[1] = 0 ← (initialized)
	q from 2 to 6 ie. 2 to m (m=6) size of pattern				
0	(2)	P[0+1] = P[1] = a	P[2] = b; k>0? NO; a≠b ∴ Π[2] = k = 0	0	Π[2] = 0
0	(3)	P[0+1] = P[1] = a	P[3] = a; k>0? x; a = a, k = k+1	1	Π[3] = 1
1	(4)	P[1+1] = P[2] = b	P[4] = c; k>0? yes; b≠c; k = Π[k] = Π[1] = 0	0	Π[4] = 0
0	(5)	P[0+1] = P[1] = a	P[5] = a; k>0 x; a = a, k = k+1 = 0+1 = 1	1	Π[5] = 1
1	(6)	P[1+1] = P[2] = b	P[6] = b; k>0 ? yes; but b = b, k = k+1 = 1+1 = 2	2	Π[6] = 2

P[q]

1	2	3	4	5	6
a	b	a	c	a	b
0					

So,

1	2	3	4	5	6
a	b	a	c	a	b
0	0	1	0	1	2

The prefix function for pattern a b a c a b

is computed as follows:-

	1	2	3	4	5	6
P[]	a	b	a	c	a	b
π	0	0	1	0	1	2

In KMP algorithm, the comparison is from Left to Right:-

T: a b a c a a b a c c a b a c a b a a b b

P: a b a c a (b)

1 2 3 4 5 6 ✗

6th comparison is a failure for 'b' in pattern

The value in table, for this 'b' is 2.

So, start comparing from 2nd character in the pattern from this position.

T: a b a c a a b a c c a b a c a b a a b b

P: a (b) a c a b

Already matched (no comparison will be done here)

7th ✗

But 7th comparison is a mismatch.

Value is '0', so, move the string altogether

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

P: a (b) a c a b

8th 9th

8th ✗

9th comparison is a mismatch. Value for this b is zero, so move the pattern altogether.

T: a b a c a a b a c a b a a b b

P: a b a c a b

10th comparison is a mismatch due to a.

value for this a is zero. So move the string altogether.

T: a b a c a a b a c c a b a a b b

P: a b a c a b

11 12 13 14 15 16

Match is found.

6. Applications of String Matching Algorithm

The string matching algorithms provide/play key role in various real world problems or applications.

A few of its important applications are:- as:-

a) Spell checkers
b) Spam Filters
c) Intrusion Detection System
d) Search Engines
e) Plagiarism Detection
f) Bioinformatics
g) Digital forensics
h) Information Retrieval systems.,

etc. ....