Knuth - Morris - Pratt Algorithm

- Algorithm has efficient technique which avoids repeated comparison of characters.
- In KMP, whenever there is a mismatch (after some matches) some of character of text string is already known.

Example:

Text : b b b b b c a b b b

Pattern : b b b b

These characters are already matched.

I] The Prefix function for a pattern

- The prefix function π or LPS for a pattern encapsulates knowledge about how the pattern matches against shift of itself.
- It processes the pattern to find matches of prefixes of the pattern with itself.

⇒ This function finds longest proper prefix which is also suffix.

prefix of Pattern → Subset of a pattern starting from begining

suffix of Pattern → Subset starting from end.

Given String: a b a d a b

(i) Initially put 0 in 0th location

	0	1	2	3	4	5
	a	b	a	d	a	b
P	0					

(ii) String: ab
Prefix: ε, a
Suffix: ε, b

	a	b	a	d	a	b
P	0	0				

(iii) String: aba
Prefix: ε, a, ab
Suffix: ε, a, ba

	a	b	a	d	a	b
P	0	0	1			

(iv) String: abad
Prefix: ε, a, ab, aba
Suffix: ε, d, ad, bad

	a	b	a	d	a	b
P	0	0	1	0		

(v) String: abada
Prefix: ε, a, ab, aba, abad
Suffix: ε, a, da, ada, bada

a	b	a	d	a	b
0	0	1	0	1	

(vi) String: abadab
Prefix: ε, a, ab, aba, abad, abada
Suffix: ε, b, ab, dab, adab, badab

a	b	a	d	a	b
0	0	1	0	1	2

- It is defined as the size of the largest prefix of the P[0... j-1] that is also a suffix of P[1... j]

LPS[j] = Longest Proper Prefix of Pattern P[0... j] which also a Suffix of Pattern P[0... j]

- It also indicates how much of last comparison can be reused, if it fails.
- It enables avoiding backtracking on Text strings

⇒ Algorithm Calculate_LPS (Pat[], m)

```
{
1.  k=0                  // Length of previous longest pattern
2.  p[0] = 0
3.  i = 1                // Pointer on Pattern
4.  while (i<m)
    {
5.      if (Pat[i]==Pat[k])        // If match found
                                      case (1)
        {
6.          k++
7.          p[i] = k
8.          i++
        }
9.      else
        {
10.         if (k!=0)              // Mismatch & k!=0
                                      case (2)
11.             k = P[k-1]
12.         else                   // Mismatch & k=0
                                      case (3)
            {
13.             p[i] = 0
14.             i++
            }
        }
}
```

Example:

Pattern = a a a c a a a a a c a

	0	1	2	3	4	5	6	7	8	9	10
Pat	a	a	a	c	a	a	a	a	a	c	a
p	0	1	2	0	1	2	3	3	3	4	5

⇒ Processing

i : 1 2 3 3 3 4 5 6 7 7 8 8 9 10

K : 0, 1 2 1 0 0 1 2 3 2 3 2 3 4 5

Init

Case 1 1 2 2 3 1 1 1 2 1 2 1 1 1

Example 2:

	a	b	c	d	b	a	e	a	b	f
P	0	0	0	0	0	1	0	1	2	0

Example 3:

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

II] Search Algorithm

- In this LPS is used to decide next character to be matched
- Logic is - Do not match character that are definitely going to match.

→ How to find number of characters to be skipped?

- Initially set pointer i at initial location of Text and set pointer j at initial location of pattern.
- Keep incrementing i and j until Text[i] and Pattern[j] matches.
- When there is mismatch at index j of Pattern it implies
  - (a) Pattern[0 ... j-1] ≡ Text[i-j ... i-1]
  - (b) LPS[j-1] is count of characters of Pattern[0 ... j-1] which are both Prefix and Suffix.

⇒ We do not need to match LPS[j-1] characters with Text[i-j ... i-1] as they will anyway match.

Algorithm

```
KMP_Matcher (Pat[], Text[])
{
1.   m = pat.length
2.   n = Text.length
3.   Calculate_LPS (Pat[], m)
4.     i = 0                      // Pointer on Text
5.     j = 0                      // Pointer on Pattern
6.   While (i<n)
     {
7.     if (Pat[j]==Text[i])       // Case 1: Match
       {
8.         i++
9.         j++
       }
10.    if (j==m)                  // Case 2: Pattern Found
       {
11.        print ("Pattern Found at", i-j)
12.        j = P[j-1]
       }
13.    if (i<n and Pat[j]!=Text[i])
       {
14          if (j != 0)           // Case 3: Mismatch & j != 0
15              j = p[j-1]
16          else                  // Case 4: Mismatch & j = 0
17              i = i+1
       }
     }
}
```

Question 1: Pattern: a a a

Text: a a a c a a a a c a a

Solution:

Length of Pattern $m = 3$

Length of Text $n = 11$

Compute LPS for pattern

	0	1	2
Pat	a	a	a
P	0	1	2

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

i	0	1	2	3	3	3	4	5	6	7	8	8	8	9	10
j	0	1	2	$\cancel{3}$ 2	1	0	0	1	2	$\cancel{3}$ 2	$\cancel{3}$ 2	1	0	0	1
Case	1	1	1 = 2	3	3	4	1	1	1 = 2	1 = 2	3	3	4	1	2

Pattern Found: 0
$3 - 3 = 0$

Pattern Found: 4
$7 - 3 = 4$

Pattern Found: 5
$8 - 3 = 5$