Today's Content

1. Search for Pattern in Text
2. Pattern matching using KMP
3. Space Optimization using KMP

Q: Given P_K & T_N, count occurences of P_K in T_N

How many substrings of $T == P_K$

Ex1:

$$\begin{array}{cccccccc} & & 0 & 1 & 2 & 3 & 4 & 5 \\ T & = & a & a & b & a & c & d \end{array}$$

$T = a\ \boxed{a\ b\ a\ c}\ d$ #ans = 1

$P = a\ b\ a\ c$

Ex2:

$$\begin{array}{ccccccccccccc} & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ T & = & c & a & b & a & d & c & a & b & a & b & a & e \end{array}$$

$T = c\ \boxed{a\ b\ a}\ d\ c\ \boxed{a\ b\ a}\ b\ a\ e$ #ans = 3

$P = a\ b\ a$

Idea1: Take all substrings of len k in T_N & compare with Pattern P.

TC: $O(N-K+1) * O(k)$

No: of substrings of len = k

Time taken to compare 2 strings of len = k

$$\boxed{\begin{array}{lll} \text{if } k==1 & : & O(N-1+1) * O(1) = O(N) \\ \text{if } k==N & : & O(N-N+1) * O(N) = O(N) \\ \text{if } k=N/2 & : & O(N-N/2+1) * O(N/2) = O(N^2) \end{array}}$$

Dry Run:

$$\begin{array}{cccccccc} & & 0 & 1 & 2 & 3 & 4 & 5 \\ T & = & a & a & b & a & c & d \\ P & = & a & b & a & c & & \end{array}$$

Ex: $T[0:3] =$ aaba == P

$T[1:4] =$ abac == P

$T[2:5] =$ bacd == P

#Idea2:

Optimize using lps[]

1. lps[] helps to search prefix which also exists as suffix, & we need to search pattern in Text, so append pattern at start of Text.
2. Now pattern acts like prefix, Now your lps can help you search pattern in Text.
3. While combining Pattern & Text use a separator to separate P & T

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

T = c a b a d c a b a b a e

P = a b a

S = P $ T

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S =	a	b	a	$	c	a	b	a	d	c	a	b	a	b	a	e
lps[] =	0	0	1	0	0	1	2	3	0	0	1	2	3	2	3	0

Final:

```
if ( lps[i] == P.length() ) {
    c++;
}
```

```
                      K          N
int Occurence(string P, string T) {   TC: O(N+K)      SC: O(N+K)
                                           New String + lps

    string S = P + "$" + T;   # (K+1+N)
    vector<int> lps = LpsCreate(S);   # It will return lps[]
    int c=0;                        #String length = N+K+1
    for( int i=0; i< lps.size(); i++){
        if( lps[i] == P.length()){
            c++;
        }
    }
    return c;
}
```

Why deliminator? It separates pattern & Text

	0	1	2	3
T =	a	b	c	d

P = a a a

	0	1	2	3	4	5	6
S = PT	a	a	a	a	b	c	d
lps[7] :	0	1	2	3			

↳ == P.length(); c = c+1

	0	1	2	3	4	5	6	7
S = P@T	a	a	a	@	a	b	c	d
lps[8] :	0	1	2	0	1	0	0	0

#Note: Deliminator acts as separator between Pattern P & Text T, since it acts has a separator make sure, separator is not in both Pattern & Text character.

Above logic is not Ideal?

Purely because of Space complexity above logic is not Ideal.

SC: $O(N+K)$

↳ #N Indicates, we are creating a full copy of Text, which is okay in your programming assignments, but in your real world application it's not very Ideal.

#Here we need to optimize space.

KMP: Used to search for a pattern in Text

P: a b a b c

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

App1: P @ T (@ → separator)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S =	a	b	a	b	c	@	a	b	a	b	a	b	c	a	b	a	b	c
lps[ ]:	0	0	1	2	0	0	1	2	3	4	3	4	5	1	2	3	4	5
Cmp: S[i] compared index							0	1	2	3	4	3	4	5	1	2	3	4

c=c+1   c=c+1

Q To calculate lps values for Text characters, what is needed.

1. lps of previous char
2. Compare Text char to pattern char.
   - if matching incr lps by 1
   - if not matching update $n = lps[n-1]$ & cmp again.

#Con: To calculate lps value of Text.
We need previous lps value & lps[] values of P+@

Optimized Space:

App2:

	0	1	2	3	4	5
P =	a	b	a	b	c	@
lps[6]:	0	0	1	2	0	0

	0	1	2	3	4	5	6	7	8	9	10	11
T =	a	b	a	b	a	b	c	a	b	a	b	c
n=0	1	2	3	4	3	4	5	1	2	3	4	5

↳ if n == p.length() : c=c+1

```
int ocurences(string P, string T){   TC: O(K+N)  SC: O(K)
                     K          N
    P += '@';
    vector<int> lps = lps(P);  # It will return lps[]

    #Calculate lps value for Text;
    int n = 0, c = 0;
    for(int i = 0; i < T.length(); i++){
        # n is representing lps value of previous.
        while( P[n] != T[i] ){
            if(n == 0){
                n = -1;
            }   break;
            n = lps[n-1];
        }

        n = n+1; #update lps.
        if(n == P.length() -1){
        }   c = c+1;
    }

    return c;
}
```

#Note: Above pattern matching with lps[], is considered as KMP.

Given a String S_N, min character to be added at start of string to make entire string palindrome

Ex: S = d c a d a c d len = 2

S = f e d a b c b a d e f len = 3

S = h g a a e a a g h len = 2

S = e a b a d a a d a b a e len = 1

S = a b a b a len = 0

S = c b a b c len = 2

Idea1: Calculate length of longest prefix palindrome = l
Final ans = $N - l$;

Appl: Generate all prefix substrings & check palindrome or not & get max substring length which is palindrome

TC: $O(N) * O(N) = O(N^2)$ SC: $O(1)$

Prefix substring are N To check substring is pal or not

Idea2:

$$\begin{array}{l} \quad\;\; 0\ 1 \ldots \ l-1 \\ s: \boxed{}\ x\ a\ b\ y\ x \end{array}$$

T : s @ rev(s)

$$T : \boxed{0\ 1 \ldots\ l-1}\ x\ a\ b\ y\ x\ @\ x\ y\ b\ a\ x\ \boxed{0\ 1 \ldots\ l-1}$$

lps[] : l

l: Indicates length of longest prefix palindrome

```
int minChar(string s){
    string T = s + "@" + rev(s);
    vector<int> lps = lps(T);
    int l = lps[lps.size()-1];
    return s.length() - l;
}
```

TC: $O(2N + 2N) = O(N)$ — $2N$: Concatenation, $2N$: lps of T

SC: $O(2N + 2N) = O(N)$ — $2N$: T string, $2N$: lps[]