Pattern Matching

Agenda

1. Longest Prefix
2. Boring Substring
3. Pattern Matching :
   KMP Algorithm

Hello Everyone
Very Special Good Evening
to all of you 😊😊😊
We will start session
from 9:06 PM

Longest Prefix

Given an array of Strings. Find the longest string which is prefix of all strings in the array.

Array : ["axdc" , "axbc" , "axdb"] , ans: ax

prefix → substring which is starting from index '0'

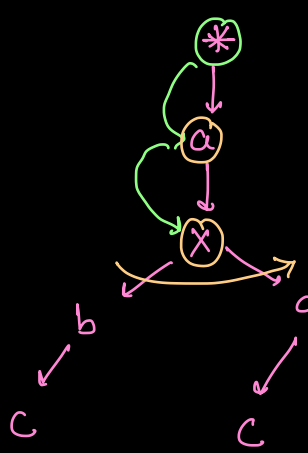
What is the Longest Common Prefix for the given array of Strings? A = ["aabc", "aaabc", "aabc"]

ans= "aa"

Longest Prefix using Trie:

Array : ["axdc" , "axbc" , "axdb"]

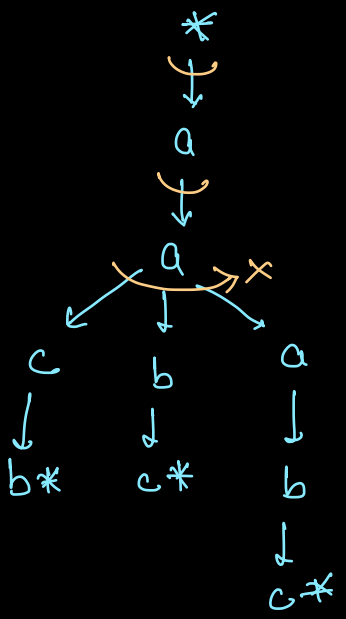
"ax"



Common prefix = "ax"

End of word not there
have single child.

eg: [aabc aaabc aacb]



"aa" → longest common prefix

T.C: O(n*l)
S.C: O(n*l)

n → no. of string
l → l → avg. length of string.

Bruteforce Approach:

0 1
a x c c
a x b c → stop, No further common character available
a x c b
a x a b

ax

T.C: O(n*l) } 9+ is better than trie
S.C: O(1)

ans:=

C++ → Normal string →
Java → * make a StringBuilder
* after preparing ans convert into string.

Boring Substring

Given a string S, check whether it is possible to re-arrange the characters of the string S such that there will be no boring substring in S.

Boring Substring : length = 2 and consecutive alphabets

↳ ab , cd , xy , yz, yx, dc, cb, za , ba

S= "abc"

abc
acb
bac
bca
cab
cba

false

S= "abcd"

abcd
cadb → true
bdac

cadb

ca → x
ad → x
db → x

aa → x
ac → x
nm → ✓ Boring substring.
tr → x

eg → ① "aabccd"

ccaadb } true
bdaacc

② "aabccb"

a, b, c

a _ b _ c a

↳ false

Idea: Construct all permutation of given string and check if they are having boring substring or not.

T.C: O(N! * N)

S.C: O(1)

Idea2:

s = "aabccdb" bbd aacc ans → true

a b c d e f g h i j k . . . . . w x y z

a c e g i k . . . . w y → odd ASCII
97 99 101 103 121

b d f h j l . . . . x z → even ASCII
98 100 102 122

a → 97
b → 98
c → 99
d → 100
:
z → 122

s = aabccdb

odd → aacc
even → bdb

cac a a bb
acc
cca

cacadbb
accadbb
ccaadbb

ans: true.

s = aabccb

aacc
bb

ans: false

first character	second character	
a →	b, b	x
a →	b, b	x
c →	b, b	x
c →	b, b	x

N
N/2 N/2

Itr = N/2 × N/2

T.C: O(n²)

Idea2:

① only unique character:

max possible character in one end

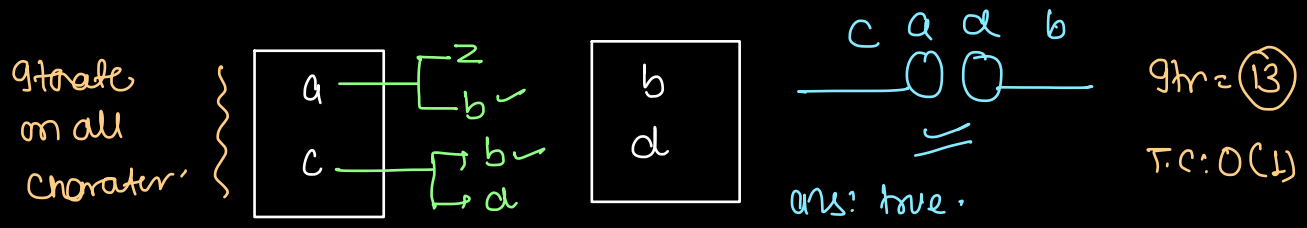
a c e g i . . y (13)

b, d, f, h, . . z (13)

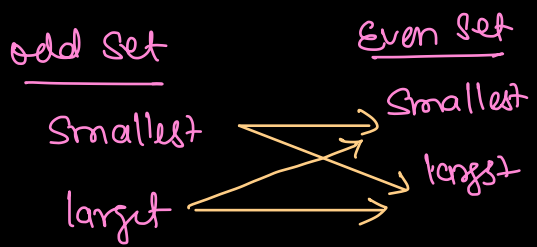
str = 13×13 → constant

T.C: O(1)

② Hashmap / HashSet to check for all odd ASCII character, if anything other than (ch-1) or (ch+1) is present.



③ S = a a b c c d b



T.C: O(1)

a a a c c e g g
S L

b b d f f
S L

a → b ✗
a → f ✓
g → b ✓
g → f ✗

true.

S = "ccebbaf"

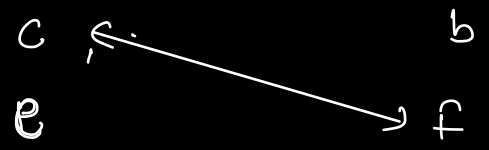
smallest odd ascii → c

largest odd ascii → e

smallest even ascii → b

largest even ascii → f

T.C: O(n)
S.C: O(1)



true ;

KMP : Knuth-Morris-Pratt algorithm

* Prefix and Suffix of string
* LPS of string
* LPS array of String
* Optimisation for LPS
* Need of LPS

⇒ Given a string of length N :

What is prefix strings : Substring which starting from index 0

What is suffix string : substring ending at index (n-1)

S: a b a b
0 1 2 3

Prefix	Suffix
a	b
ab	ab
aba	bab
abab	abab

S: a b a
0 1 2

Prefix	Suffix
a	a
ab	ba
aba	aba

LPS of String: length of longest prefix which is also a suffix is LPS of String.

NOTE: excluding complete string.

S = a b c a b

(excluding complete string)

prefix	suffix
a	b
ab	ab
abc	cab
abca	bcab

LPS(S) = 2

S = a b c d e

(excluding complete string)

Prefix	Suffix
a	e
ab	de
abc	cde
abcd	bcde

Lps(S) = 0

S = a a a a

Prefix	Suffix
a	a
aa	aa
aaa	aaa

Lps(S) = 3

S = a b c d a b c

Prefix	Suffix
a	c
ab	bc
abc	abc
abcd	dabc
abcda	cdabc
abcdab	bcdabc

Lps(S) = 3

Calculate LPS of this string and Time Complexity of that Approach ?

S = a b c a b

length	Prefix	Suffix		Iteration Required
1.	a	b	⟶	1
2.	ab	ab	⟶	2
3.	abc	cab	⟶	3
⋮	⋮	⋮		
n-1	abca	bcab	⟶	n-1

lps = ∅ 2

total iter = 1+2+3+ . . . + (n-1)

$= \frac{n(n-1)}{2}$

To find LPS of a string

T.C : $O(n^2)$

LPS - array

length of longest prefix which is also suffix.

String ⇒ abc

lps[i] → LPS value for substring from 0 to i

lps(a) ⟶ LPS[0]

lps(ab) ⟶ LPS[1]

lps(abc) ⟶ LPS[2]

Break

" 10:20 - 10:30 "

Problem: Given a string S, return LPS[]

LPS[i] → LPS value of substring from 0 to i

S = "a a b a a b c"
0 1 2 3 4 5 6

LPS[] → | 0 | 1 | 0 | 1 | 2 | 3 | 0 |

gtr.

LPS[0] → Substring(0,0) = a , ans = 0 → 1^2

LPS[1] → Substring(0,1) = aa , ans = 1 → 2^2

LPS[2] → Substring(0,2) = aab , ans = 0 → 3^2

LPS[3] → Substring(0,3) = aaba , ans = 1 → 4^2

LPS[4] → Substring(0,4) = aabaa , ans = 2 → 5^2

LPS[5] → Substring(0,5) = aabaab , ans = 3 → 6^2

LPS[6] → Substring(0,6) = aabaabc , ans = 0 → 7^2

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

LPS[] in which for substring(0,i) using $O(n^3)$ to calculate LPS

Total gtr = $1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$

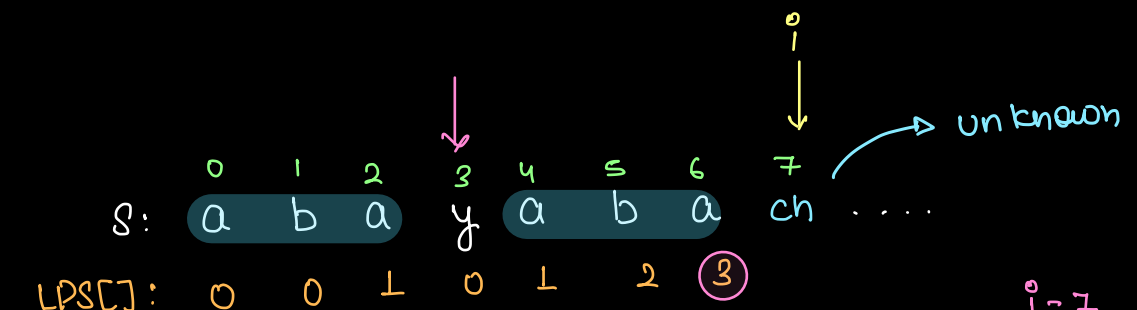
$\sum n^2 = \frac{n(n+1)(2n+1)}{6}$

T.C : $O(n^3)$

↓

$O(n)$ → using optimised lps array technique

Understanding with Examples:



lps(abayaba) = 3

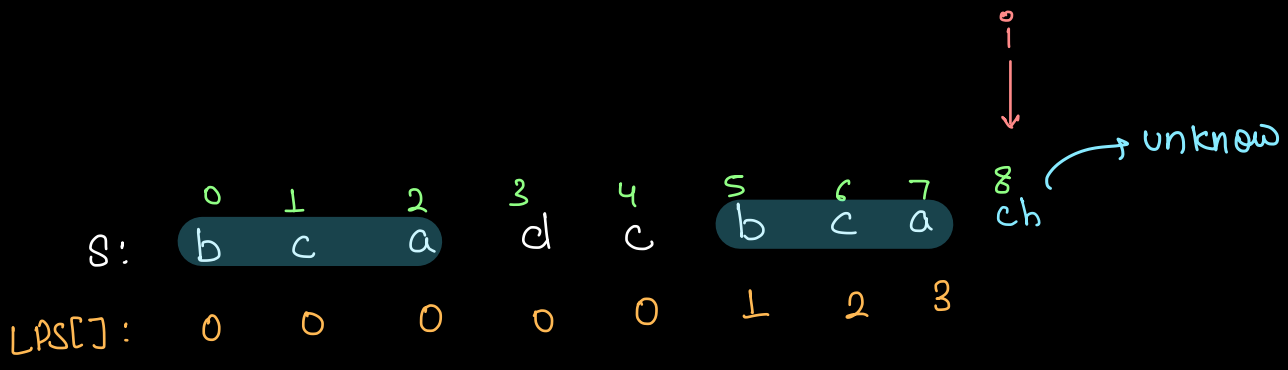
```
if( s[x] == s[i]) {
    lps[i] = x+1;
}
```

i=7

x = lps[i-1]

x = lps[6]

x = 3



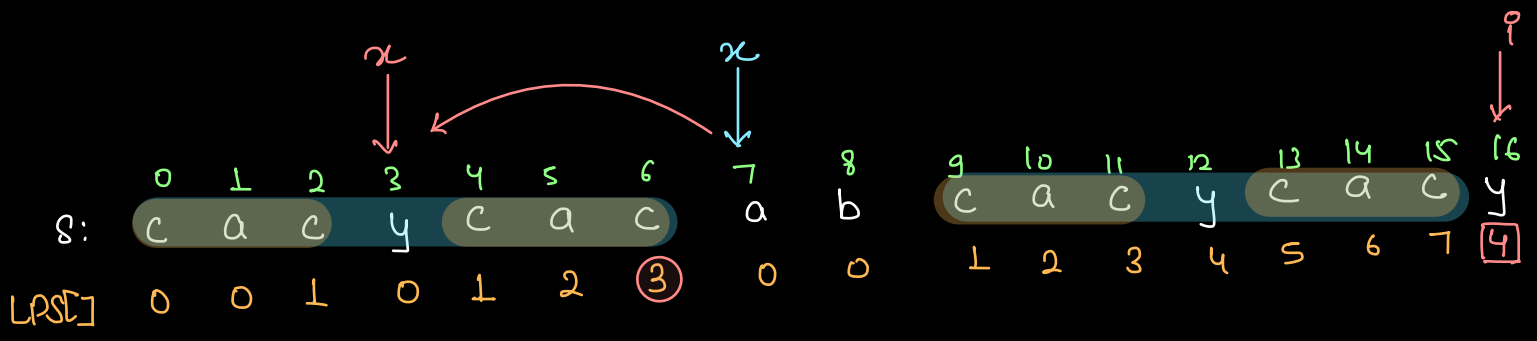

```
if( s[x] == s[i]){
    lps[i] = x+1;
}
```

i=8

x = lps[i-1];

x = lps[7]

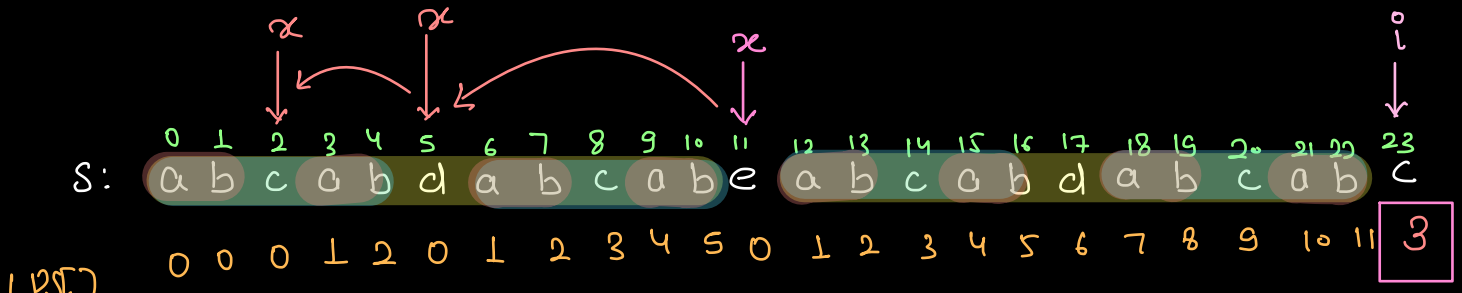
x = 3



$i=16, \quad x = lps[i-1]$

$= lps[15] = 7$

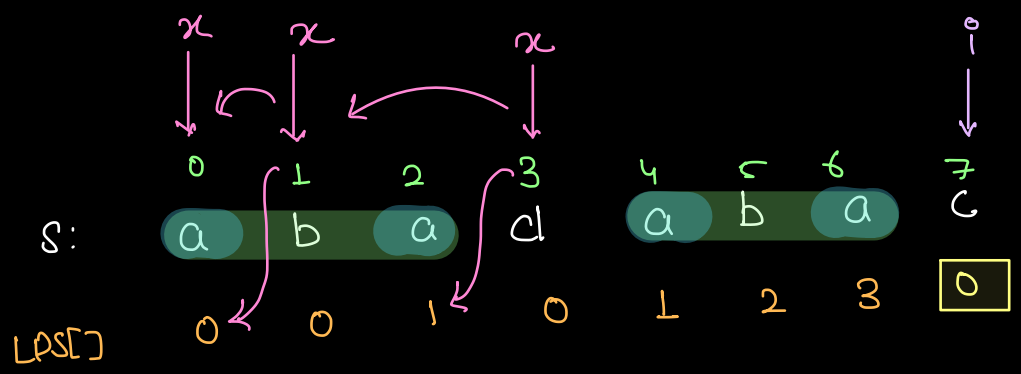
x	S[x] == S[i]	Action
7	S[16] == S[7] y == a	→ Not same, $x = lps[x-1]$ $= lps[6]$ $x = 3$
3	S[16] == S[3] y == y	→ Yes, they are same $lps[i] = x+1$



$i=23, \quad x = lps[i-1]$

$= lps[22] = 11$

x	S[i] == S[x] →	Action
11	S[23] == S[11] c == e → false	→ No, $x = lps[x-1]$ $x = lps[10]$ → 5
5	S[23] == S[5] c == d → false	→ No, $x = lps[x-1]$ $= lps[4] \Rightarrow 2$
2	S[23] == S[2] c == c → true	→ Yes, $lps[i] = x+1$ $lps[23] = 3$



i = 7

x = lps[i-1]

x = lps[6] = ③

```
if(x==0) {
    //don't move 'x'
    lps[i]=0; --as'
}
```

```
int[] lpsArray(String s){
    n = s.length();
    int[] lps = new int[n];
    Lps[0] = 0;  → why? think about it

    for(int i=1; i<n; i++){
        int x = lps[i-1];
        while( s[x] != s[i]){
            if(x==0){
                x=-1;
                break;
            }
            x = lps[x-1];
        }
        Lps[i] = x+1;
    }
    return Lps;
}
```

Time Complexity:

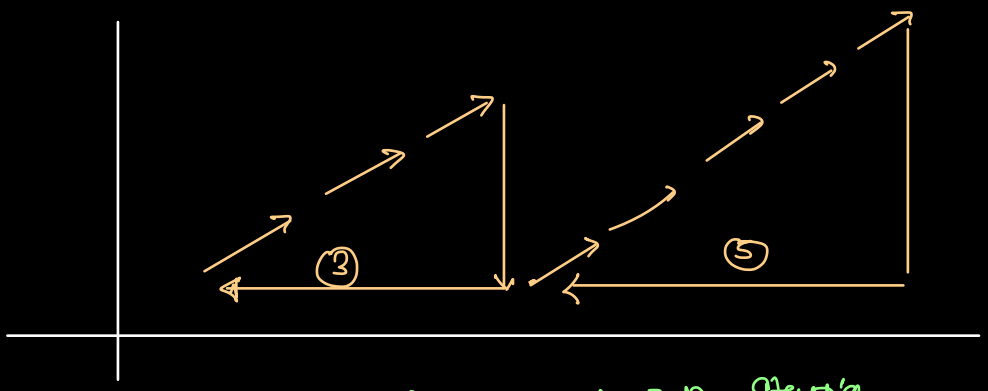
```
int[] lpsArray(String s){
    n = s.length();
    int[] lps = new int[n];
    lps[0] = 0;  → why? think about it
    for(int i=1; i<n; i++){
        int x = lps[i-1];
        while(s[x] != s[i]){
            if(x==0){
                x = -1;
                break;
            }
            x = lps[x-1];
        }
        lps[i] = x+1;
    }
    return lps;
}
```

String s: a b a y a b a b

lps:

TODO: Dry Run.

Time Complexity:



total left moment = n iterations

total right moment = n iterations

total iteration = n+n = 2n

T.C: O(n)

Problem: Count occurrence of pattern P in text T.

Eg. 1 Text = a a b a c d

Pattern = a b a c

ans = 1

Eg. 2 Text = a b a d c a b a b a e

Pattern = a b a

ans = 3

Brute force:

Using sliding window we can compare characters.

T.C: O(n * m)

n → string length

m → pattern length

n < m → ans = 0

KMP Algorithm: Knuth Morris Pratt

Text = a b a d c a b a b a c

Pattern = a b a

Searching the occurrence of pattern in given text.

String = pattern + '#' + text

separator

→ it can be any unique character which is not part of text & pattern.

text = a b a d c a b a b a c

pattern = a b a

str =	a	b	a	#	a	b	a	d	c	a	b	a	b	a	c
Lps =	0	0	1	0	1	2	3	0	0	1	2	3	2	3	0

count if lps[i] is equal to pattern length

count = 3

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

P = a b d a

String = P + # + T

Str = a b d a # a b d a b d a a b e a b d a a

lps → [0 0 0 1 0 1 2 3 4 2 3 4 1 2 0 1 2 3 4 1]

```
if( lps[i] == pattern.length){
|    count++;
}
```

pseudocode :

```
int[] lps = lpsArray( pattern + "#" + text);
int count=0;
for(int ele: lps){
|      if( ele == pat.length()){
|      |  count++;
|      }
}
```

lps Array(pattern + "#" + text) → n+m str.

if block → n+m str.

Total str = 2n + 2m

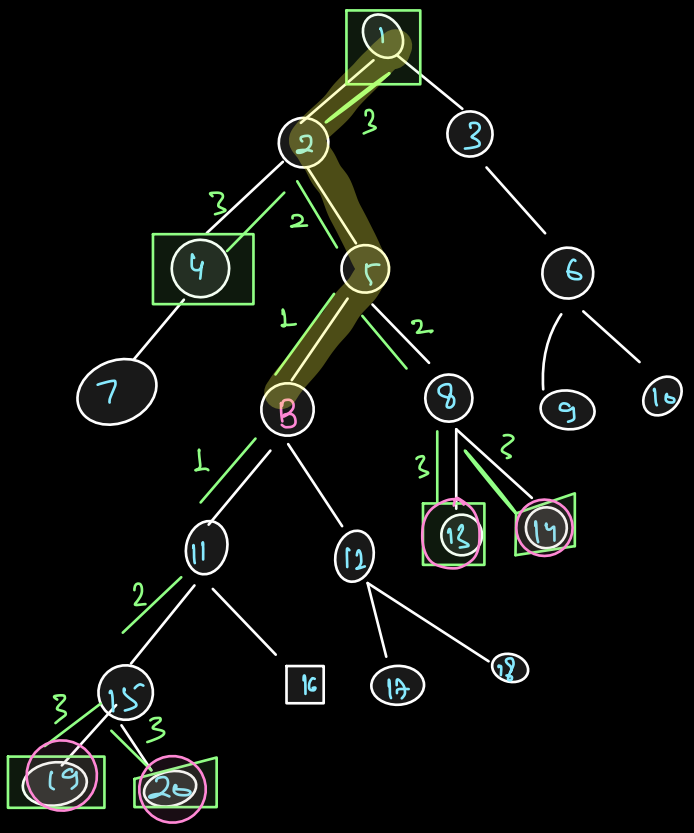
T.C: O(n+m)

S.C: O(n+m)

KMP Algo ✓

Z-Algo → problem solving.

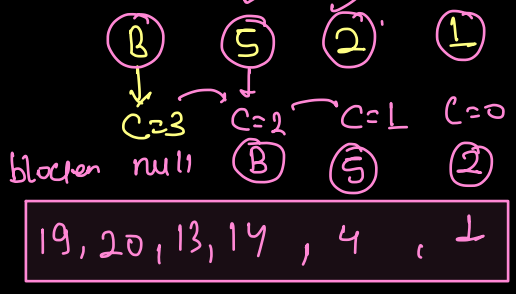
doubt session:



c = 3

* node to root path →

for 8



19, 20, 13, 14, 4, 1 → DSA mein

K-away

TODO * node to root path

* K down → print all elements which are K-down from Root