Todays Content:

- → Prefix & Suffix Strings
- → LPS of a given string
- → LPS[] of a given string

Problems based on LPS[]:

- → Pattern Matching by LPS[]
- → Cyclic Rotations
- → Min characters to be added at start to make string palindrome

Q]: Given Patten (P) and Text (T) check, if pattern is present as a substring in T

P_K = a c d a (0 1 2 3)

T_N = b c a c d a (0 1 2 3 4 5)

Dry run:

Pat

sub [0 3] = b cac == acda * : 4

sub [1 4] = cacd == acda *

sub [2 5] = acda = acda ✓

Ideal: For every substring of len k in Text (T) compare = Patten (P)

No: of sub of len = k # TC to comp to strings of len = k

→ TC: (N-k+1) * (k) SC: O(1)

k = N/2

→ (N - N/2 + 1) (N/2) ≈ (N/2 + 1)(N/2) ≈ O(N²)

Note: N is length of Text & k is length of Patten

Given a String S of N size:

Prefix Strings : All Substrings starting at index 0

Suffix Strings : All Substrings ending at last index

Ex: S = a b c a (indices 0 1 2 3)

Prefix Strings	Suffix Strings
a	a
a b	c a
a b c	b c a
a b c a	a b c a

Ex2: S = d e a (indices 0 1 2)

Suffix

a

e a

d e a

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

↓ value =

Note : Other than entire string

Eg1: S = a b c a b (0 1 2 3 4) LPS = 2

Prefix		Suffix
a	✗	b
a b	✓	a b
a b c	✗	c a b
a b c a	✗	b c a b
a b c a b		a b c a b

LPS calculation avoid given str

Eg2: S = a (0) LPS = 0

Prefix		Suffix
a	*	a

Eg3: S = a a a a a (0 1 2 3 4) LPS = 4

Prefix		Suffix
a	✓	a
a a	✓	a a
a a a	✓	a a a
a a a a		a a a a

How to calculate it LPS?

$S = S_0\ S_1\ S_2\ S_3\ S_4$

Prefix		Suffix	Iterations
S_0	⟷	S_4	1
$S_0 S_1$	⟷	$S_3 S_4$	2
$S_0 S_1 S_2$	⟷	$S_2 S_3 S_4$	3
$S_0 S_1 S_2 S_3$	⟷	$S_1 S_2 S_3 S_4$	4

Total iterations = 10

$S_N = S_0\ S_1\ S_2\ S_3 \ldots S_{N-3}\ S_{N-2}\ S_{N-1}$

Prefix:	Suffix	Iterations
S[0 0]	S[N-1 N-1] :	1
S[0 1]	S[N-2 N-1] :	2
S[0 2]	S[N-3 N-1] :	3
S[0 3]	S[N-4 N-1] :	4
⋮	⋮	
S[0 N-2]	S[1, N-1] :	N-1

TC: $1 + 2 + \ldots N-1 \Rightarrow \frac{(N)(N-1)}{2}$

↳ ≈ $O(N^2)$

Given S_N return LPS[N],

TC : N * O(N^2) % TC: O(N^3)

↳ We are calculating lps for N strings * for a single string to calculate LPS = O(N^2)

Calculating lps[] for a string of N size = O(N)

Coming class ⇒ DFS

LPS[i] = LPS value of substring [0 i]

Eg: S =

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

LPS[7] = 0 1 0 1 2 3 4

LPS[0] = lps of substring S[0-0] = 0

LPS[1] = lps of substring S[0-1] = 1

LPS[2] = lps of substring S[0-2] = 0

LPS[3] = lps of substring S[0-3] = 1

LPS[4] = lps of substring S[0-4] = 2

LPS[5] = lps of substring S[0-5] = 3

LPS[6] = lps of substring S[0-6] = 4

Q1) Search for a given Pattern P in Text T

Note: Both Contains only lower case alphabets

Eg1:

	0	1	2	3	4	5
T_N =	a	a	b	a	c	d

P_K = a b a c

Idea: Since lps helps with prefix substrings, append Pattern (P) at the start of Text

C_{N+K} =

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

P ..... found in T → T

lps[10] = 0 0 1 0 1 1 2 3 4 0

↳ Note: 4 = pattern length pattern is found in Text.

Steps: Given P_K, T_N

→ C_{K+N} = P + @ + T

↳ //Separator added to avoid edgeCases

→ Create Lps[] for newly concatenated String C : TC : (N+K)

→ If lps[] contains = K :

True In that Case pattern found

False Pattern not found

TC: O(N+K) SC: O(N+K)

Edge Case:

P_4 = a b a b

T_5 = a b a c a

patten is not present in Tent

C_9 =

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

P ——— T

Lps[9] = 0 0 1 2 3 4 _ _ _

↳ // pattem is present in Tent

↳ // But we don't have pattem ?

// Add a char separtr, between Patton & Tent

↳ // any char which is not present in both Patton & Tent

↳ Separtr: @,

P_4 = a b a b T_5 = a b a c a

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

C =

Lps[10] = 0 0 1 2 0 1 2 3 0 1

// In Tent pattem is not present, as in Lps[] none of value = k

Q2) # Count no: of Substrings of T are same as Pattern (P)

P: a a T_N: a a a a

k

0 1 2 3

9:57 → 10:02

0 1 2 3 4 5 6

C : a a @ a a a a

Lps[7] : 0 1 0 1 2 2 2

Count of 2 in Lps[] = 3 = Freq of Patten in Text

Idea: Count no: of k in Lps[]

TC: O(N+M) SC: O(N+M)

Q3) Given a Binary String S_N

Find no: of start → end rotations of S will give same string S

Note: At max N rotations are possible →

Ex: S = 1 0 1 0 (0 1 2 3) ans = 2

After 1 = 0 1 0 1 == S

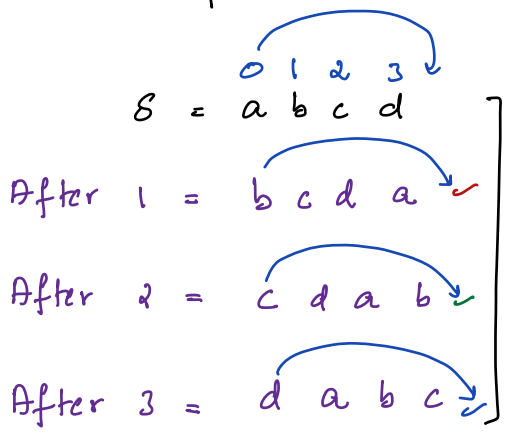
After 2 = 1 0 1 0 == S

After 3 = 0 1 0 1 == S

After 4 = 1 0 1 0 == S

After 5 = not possible

// Rotation explanation



After 4 = a b c d // same as original string, count >=1

[last rotation will match to original string]

// obs: After each rotation we are comparing with same string S

[We can say Pattern P = S

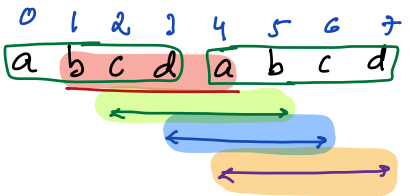
Text = S + S : all rotations of original string

How many times patter P comes in Text ?

TC: O(3N) → O(N)

SC: O(3N) → O(N)]

Text =



obs: In Text we will get a extra count of 1 ? In Text original is present twice ideally it should be present only once : return cnt-1

Dry run: S = 1 0 1 0

P = 1 0 1 0

T = 1 0 1 0 1 0 1 0

	0	1	2	3	4	5	6	7	8	9	10	11	12
C =	1	0	1	0	@	1	0	1	0	1	0	1	0

Lps[13] = 0 0 1 2 0 1 2 3 4 3 4 3 4 : cnt = 3

↳ // return cnt-1 = 2

4Q) Given a String S_N, min characters to be added at start of String to make entire string palindrome

Ex1: S = d c a d a c d ans = 2

S = d c a a b b a a c d ans = 3

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

S = g h a a a e a a a g h ans = 2

S = a b a d a a d a b a ans = 4

S = c b a b c ans = 2

ans = N - {length of longest palindromic substring start at 0}

= N - {longest prefix palindromic substring} ?

Idea: S = a b a c d (0 1 2 3 4)

↳ rev String & q concatenated

C = a b a c d d c a b a (0 1 2 3 4 5 6 7 8 9)

Lps[10] = 0 0 1 0 0 0 0 1 2 3

[longest prefix palindrome]

Edge Case : S = a a a

```
        0 1 2 3 4 5
C =     a a a a a a
```

Lps[C) = 0 1 2 3 4 (5) ⇒ longest prefix palindrome

Note: To avoid edge case need separator : @

S = a a a

```
        0 1 2 3 4 5 6
C =     a a a @ a a a
```

Lps[7] = 0 1 2 0 1 2 (3) ⇒ longest prefix palindrome

Steps : Given S_N

→ C = S_N + '@' + rev(S) (@ : 1, rev(S) : N)

→ For string C calculate Lps[2N+1]

→ ans = N − Lps[2N] // longest prefix palindrome

↳ // min characters to be added at start of original string S to make it palindrome

TC : O(2N) → O(N) SC : O(2N) → O(N)

//Lps[] : How to optimize this

For a given S_N, to calculate Lps[N]

```
Lps[0] = 0 ✓
i = 1; i < N; i++) {   TC → O(N) ✓
    //Say we good Lps[i] ✓
     x = Lps[i-1]
     while (s[i] != s[x])
        if (x == 0) { x = -1; break}
         x = Lps[x-1]
     }
     Lps[i] = x + 1
}
```