Kmp

Pattern : a b a c → (M)

Text : b a c a b a c d a (N)

LPS[] : we store length of longest prefix which is suffix

Pattern & Text

LPS[]

a c [a b a c] d a

1 0 1 2 3 4 0 1

→ Memorize? :

→ ① Algo

→ ② Pattern → Suffix

Prefix → Suffix

Perfom → Suffix

Zero

⇒ generate New String

→ Str = Pattern & Text : 1

→ Calculate lps[] of Str : 5

→ Iterate of lps[] if lps[i] == (M) : 1

→ We can also get no of occurrence of pattern in Text

TC: O(N+M)

SC: O(N+M+1)

[New String + lps[] space]

Actual Algo

TC: O(N+M)

SC: O(M)

Idea: To Reduce Space Complexity

obs1: Pattern & Text

LPS[] → What can be the max LPS[] value: M

To get lps[i], if we have lps[i-1] & from [0, M] character of that suffices to get lps[i]

obs2: To get lps[i]

a = lps[i-1] : a <= M

str[i] == str[a] : [0, M]

str[i<=M]

[We will only be needing from [0, M] characters]

We only Need Pattern & Text

Text[i] == pattern[a]

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

0 1 2 3 4 5 6 7 8 9 10 11

Pattern : a b a b a

LPS[] : 0 0 1 2 3

[You get your Text count++]

Idea: Assume That Text is placed after pattern, & we are calculating lps on of Text on fly

→ Did we create a new string? No

→ SC: O(M)

→ Given Pattern, Text

M N

P = P + '$'

Construct lps[] of P

for loop:

int pre = 0;

for (int i = 0; i < N; i++) {

for a = pre;

while (a > 0 && P[a] != T[i]) {

a = LPS[a-1];

}

if (P[a] == T[i]) {

a++;

}

if for P's, we have lps[]

if (a == M) {

cnt++;

}

prev = a;

→ lps[i]

}

⇒ Pattern: M character

Text: [Continuous stream of characters]

[when ever new character comes, Just get lps for that character

→ lps of previous character

→ lps[] of prefix pattern[]

Total No of occurrence of pattern in Text:

Final: (Prakhar)

⇒ Given a string, How many distinct substrings are there?

Ex: a c a c b

| | | | | |
|---|---|---|---|---|
| a - 1 | c - 6 | a - 3 | c - 5 | b - 12 |
| ac - 2 | ca - 7 | ac - 4 | cb | |
| aca - 3 | cac - 8 | acb | 13 | |
| acac - 4 | cacb | 10 | | |
| - acacb | 9 | | | |

1) Insert all substring in hashset

Insert string

TC: O(N²) + O(N)

SC: O(N²) + O(N)

To insert a string will take 800 space

TC: O(N³) SC: O(N²)

Ex: a c a c b + c

New Strings which are added

c → [Already Present]

bc ✓

cbc ✗

acbc ✓

cacbc ✓

acacbc ✓

5 → 12+5 → 17

Rev

c b c a c a

New we have all new strings as prefix right

Say we calculate lps[] of new string & say

c b c a c a

0 0 1 0 1 0

cnt = cnt + (newlength - max(LPS[])) string

Add a :

a c b c a c a

0 0 0 0 1 2 1

If [ac] is present, a is also present

cnt = cnt + [7 - 2] : 5

⇒ String = [abcbacab]

Str: a] Calculate lps[]

Str: ba] Calculate lps[]

Str: cba] Calculate lps[]

Str: bcbac]

TC: N*N

SC: O(N)

Ex: a b a b a

a | b | a | b | a

ab | ba | ab | ba

aba | bab | baba

abab | babab

ababa

⇒ Distinct Substring : 9

Now Say we add b at end

a b a b a + b

⇒ a b a b a b ✓ 2 more

b a b a b ✗

a b a b ✗

b a b ✗

a b ✗

b ✗

Prefix substring

[0-2] is already present in string

0 b a b a b a

lps[]: 0 0 1 2 3 4

If [0-2] is present

→ [0-2] ✓

→ [2-4] ✓

→ [0-0] ✓

4 are already present

[0-4] & [0-5] are not present

⇒ cnt = cnt + [(String length) - max(LPS[])]

⇒ Given a string, re-arrange string such that no boring substring.

Str: a b b d

a d b b ✓

b b d a ✗ → adjacent

b b d a ✓

Constraint:

→ length 2

→ [Prefix character]

Idea:

Which 2 character can always come together

| | |
|---|---|
| 1 3 5 7 | 2 4 6 8 |
| a c e h | b d f i |
| All the above can always come together | All the above can always come together |
| odd parity | even parity |

⇒ Sol: Generate all permutations & check one by one

TC: N! * N

SC: O(N)

Ex: a a c b c d a

| odd parity | even parity |
|---|---|
| a a a c c | b d |

[a a a c c → b d] → No boring substring

⇒ Sol: odd parity + even parity ⇒ Ex: a c d b

odd parity | even parity

[a, c, e, g, ..] | [b, f, ..]

odd parity | even parity

a c | b d

a c b d → Boring substrings

Between while diff: [a, c, g] | [b, f]

[b, f, g]

Even parity + Odd Parity → Check both of them

or

Odd Parity + Even Parity → If both boring substring Then we cannot re-arrange

Ex: a c e g [h d] f

A[N] | A[0] | A[N-1]

B[N] | B[0] | B[N-1]

Then diff