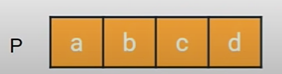
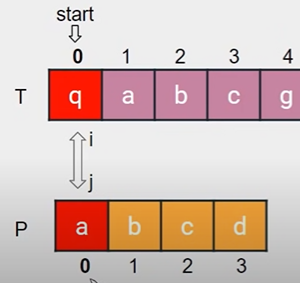
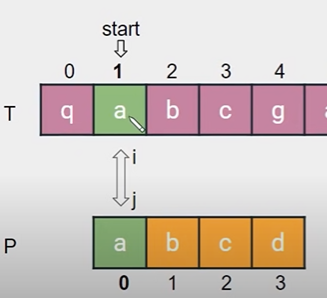
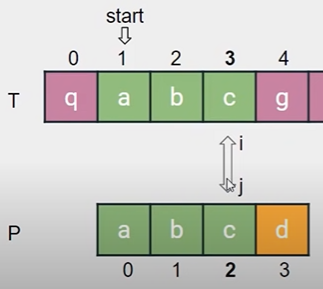
# KMP algo – Pattern matching

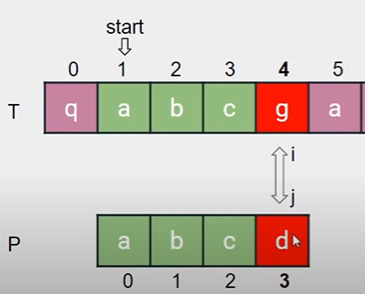
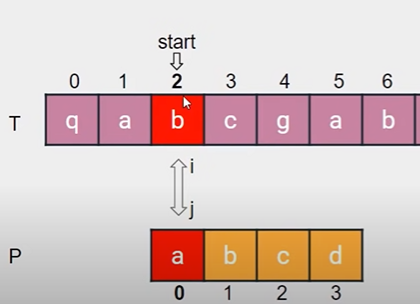
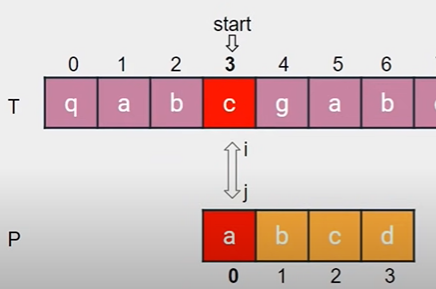
<https://www.youtube.com/watch?v=4jY57Ehc14Y>

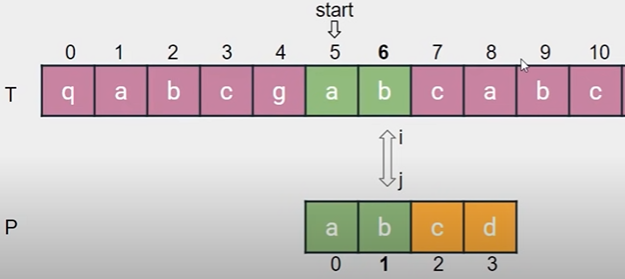
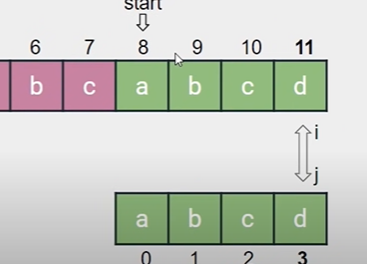
## Bruit force method:



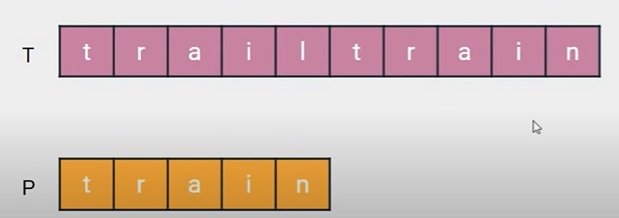
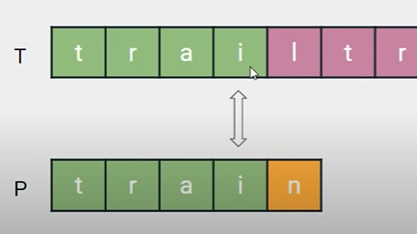


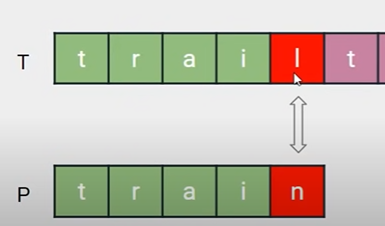
  

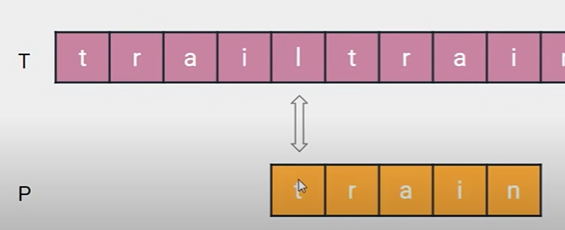
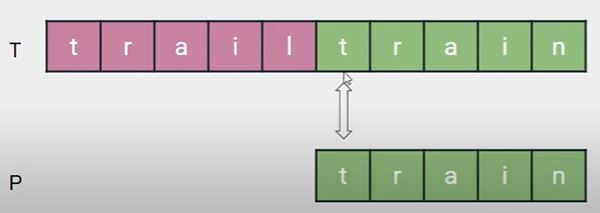
  

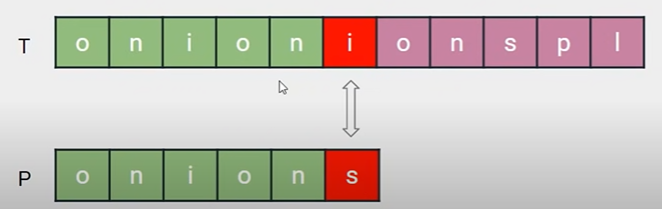
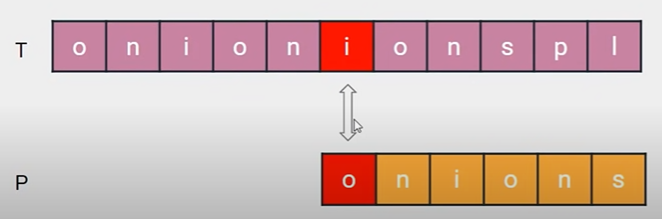
Disadvantage of brute force is O(n2 ), suppose if we skip all the matching chars?

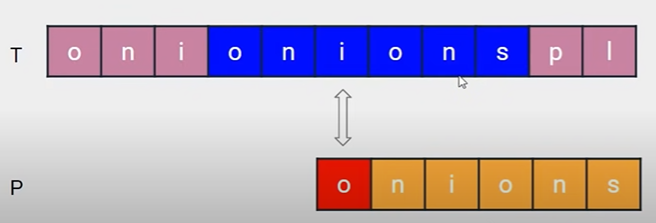


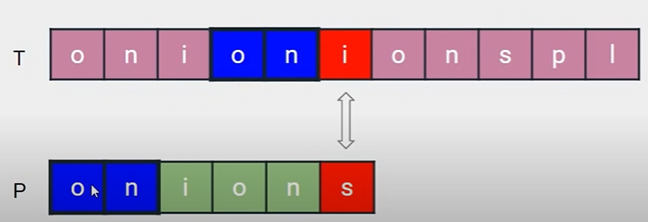
we already know that the firest 4 char are a match, so we will skip the 4 chars in the text string and start comp  

one more example

Here there are 5 chars match and 6th is mismatch, so we are skipping first 5 chars and start at 6th char with first char of pattern. We are missing the overlapping charectors. How do we know how many chars to skip?





## Suffix and prefix

There are suffixes and prefixes in a string.

o n i o n s : prefixes suffixes

o n i o n s ---5th prefix ‘s’ is “onion” o n i o n s 0 ‘o’ –suffix “nions”

o n i o n s --- 4th prefix ‘n’ is “onio” o n i o n s 1 ‘n’ – suffix “ions”

o n i o n s -- 3 char ‘o’ is “oni” o n i o n s 2 ‘i’ – “ons”

o n i o n s --- 2 ‘I’ –“on” o n i o n s 3 ‘o’ -“ns”

o n i o n s -- 1 ‘n’ –“o” o n i o n s 4 ‘n’ - “s”

o n i o n s- no prefix at 0th pos it is -1 o n i o n s - no sufix at 5th pos it is -1

preff:{“o”, “on”, “oni”, “onio”, “onion”} suff ={“nions”, “ions”, “ons”, “ns”, “s”}

preff:{“o”, “on”, “oni”, “onio”, “onion”}

suff ={“nions”, “ions”, “ons”, “ns”, “s”}

## LPS

we do not consider the mismatched pos i.e 5th

0 1 2 3 4 {pre}, {suf} longest common prefix & sufix lps

o n i o n {}, {nion} 0

o n i o n {o}, {ion} {o}, {ion} 0

o n i o n {on}, {on} {on}, {on} 2 longest match

o n i o n {oni}, {n} {oni}, {n} 0

o n i o n {onio}, {} 0

longest common prefix/sufix LPS 2

for the pattern “onion” the lps is 2

## substrings:

now calculate lps for every posiable substring of pattern

pattern: {“onion”}

substrings: {“o”, “on”, “oni”, “onio”, “onion”}

pos sub pref suff lps

0 “o” -1 -1 -1

1 “on” {“o”} {“n”} 0

2 “oni” {“on”, “o”} {“ni”, “i”} 0

3 “onio” {“oni”, “on”, “o”} {“nio”, “io”, “o”} 1

4 “onion” {“onio”, “oni”, “on”, “o”} {“nion”,”ion”,”on”,”n”} 2

Pattren: “onion”

substrings: {“o”, “on”, “oni”, “onio”, “onion”}

Lps: { -1, 0, 0, 1, 2}

2ch 3ch 4ch 5char match

Example

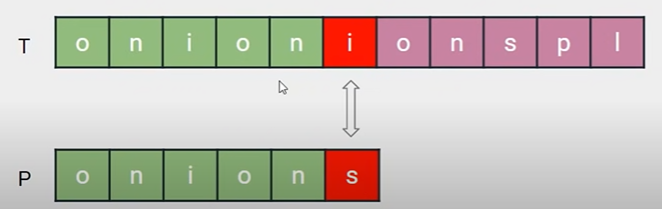
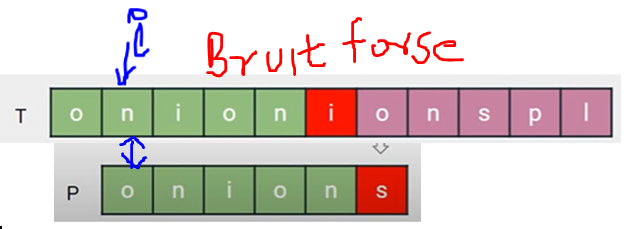
if “onion” is matched (5 chars match) in given text then skip 2 chars (lps[5])

if “onio” is matched (4 chars match) then skip 1 char.(lps[4])

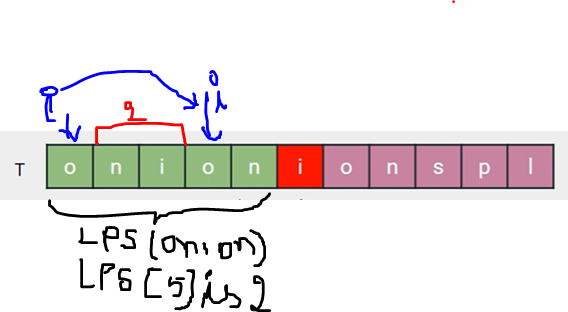
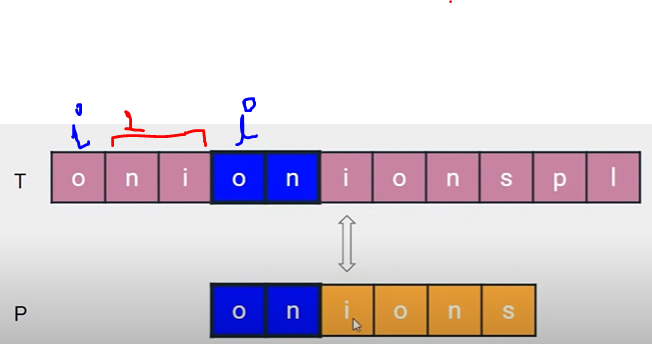
if “oni” is matched (3 chars match) then skip 0 char.(lps[3])

if “on” is matched (2 chars match) then skip 0 char.(lps[2])

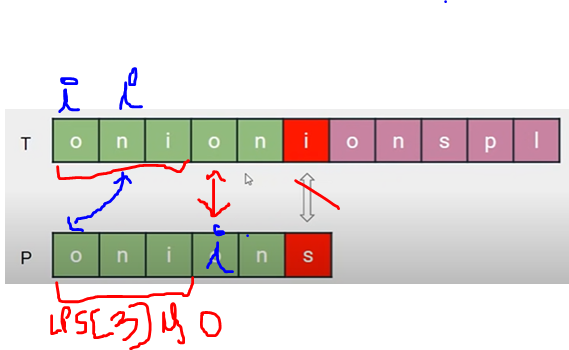
Here 5 chars match, instead of doing bruit forse increment, we use lps to skip the max number of chars.

Lps[5], 5char substring “onion”match, so skip I by 2, i+=lps[5]

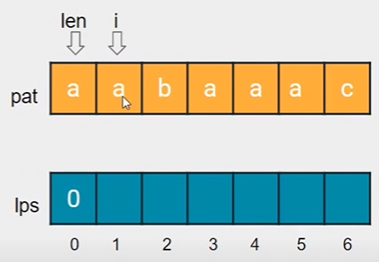
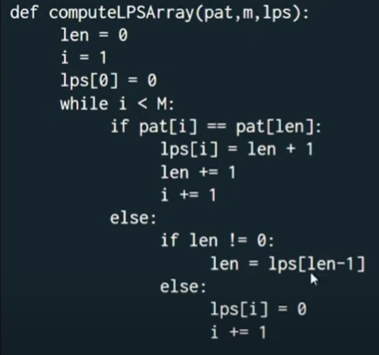
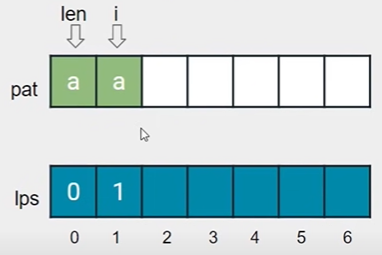
 

If there is another pattern like “oniin” and match is of 3 chars “oni” then skip lps[3] which is 0, so i will go to next pos.



## Lps coding:

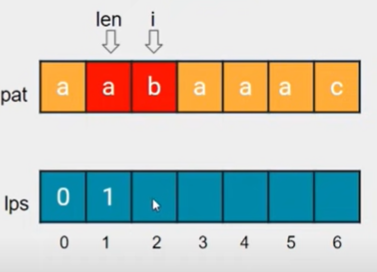
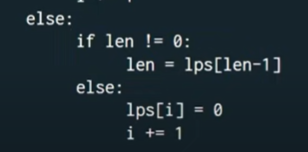
Initialize:

Case 1: match:

If the chars at indexes len and i are matching then lps[i] = len + 1, increment the len of matching chars, i++, len++, move both indexes

Case 2: not match:

Case sub1: Check if len !=0, means len is not at the initial starting position and we have already calculated some previous values of lps.

So take the prev value for len, i.e len = lps[ len -1]; at this point ‘I’ is not incremented, only the value of len is changed

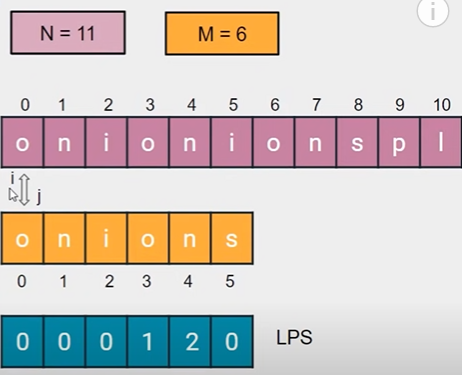
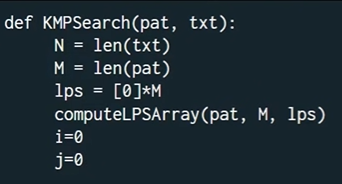
Case sub2: if len is not zero then:

As there is a mismatch so the lps[i] is zero, and increment i only,

i.e lps [ i] = 0 and i++;

## Pattern search

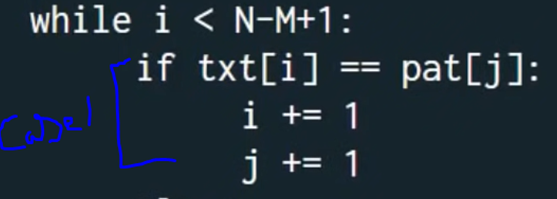
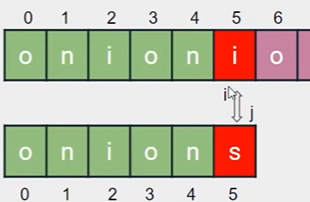
Initialize N, m,I, j and Compute the LPS array. Loop until i < n-m+1

Case 1: if char matches in both text and pattern

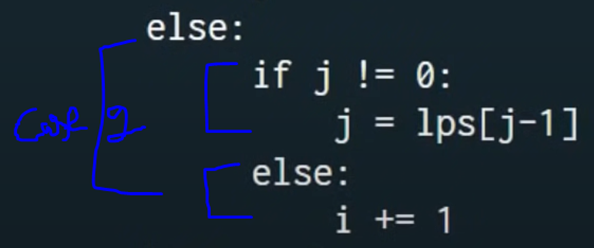
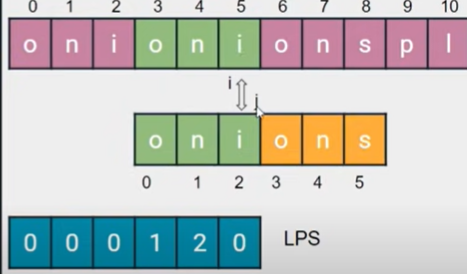
increment i and j (indexes in text and pattern)

case 1 case 2 (mismatch)

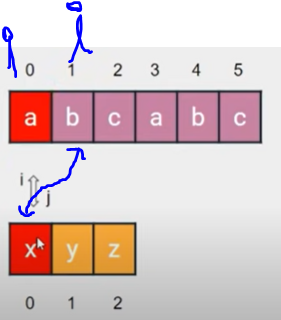
case 2: if chars did not match: mismatch: skip/set j to lps[ j-1] (kmp formula)

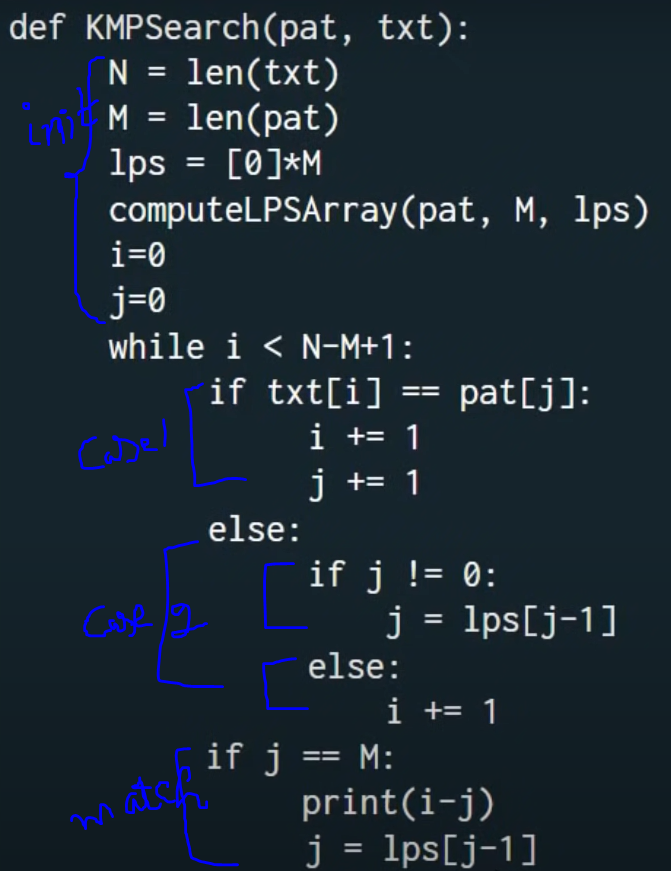
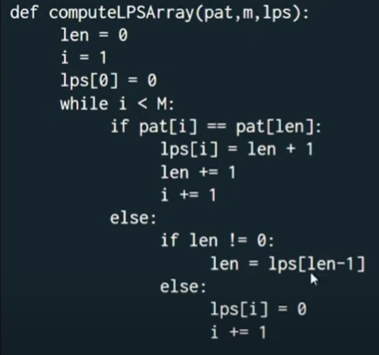
i is at the same pos

Case 2 subcase1 : check if ‘ j’ is not zero, means it is not the first char in the pattern , that means we have already have some values in lps, which gives us the optimized skip location, so set ‘j’ to the prev calculated value in lps.

Case 2 subcase2: if ‘j ‘ is zero , it is the first char of the pattern to compare and it is not matched so just increment i , the index of text.

 txt[0] != pat[0], i.e j is zero, so i++ and continue the loop

<https://leetcode.com/problems/rotate-string/submissions/>

<https://leetcode.com/problems/implement-strstr/>

<https://leetcode.com/problems/repeated-substring-pattern/>

Pattern : “abcabcabc”

substrings: {“a”, “ab”, “abc”, “abca”, “abcab”, abcabc, abcabca, abcabcab, abcabcabc}

pos sub pref suff lps

0 “a” -1 -1 -1

1 “ab” {“a”} {“b”} 0

2 “abc” {“ab”, “a”} {“bc”, “c”} 0

3 “abca” {“abc”, “ab”, “a”} {“bca”, “ca”, “c”} 1

4 “abcab” {“abca”, “abc”, “ab”, “a”} {“bcab”,”cab”,”ab”,”a”} 2

5 abcabc {“abcab”, “abca”, “abc”, “ab”, a} {“bcabc”,”cabc”,”abc”,”bc”} 3

6 abcabca {“abcabc”, “abcab”, “abca”, “abc”, ab} {“bcabca”,”cabca”,”abca”,”bca”} 4

7 abcabcab {“abcabca”, “abcabc”, “abcab”, “abca”, abc} 5

{“bcabcab”,”cabcab”,”abcab”,”bcab”}

8 abaabaaba 6

The idea is simple: if a string ***s*** is made up of ***t*** of length m repeated n + 1 times, then the longest prefix that is also a suffix has length lps(***s***) = nm (assuming ***t*** is not a repeated pattern of some smaller string)

Substring abc of len 3 is repeated for 3 times

S = abcabcabc, t = abc , len of t is m = 3 and repeated for n=3

Lps(s) will be (n-1)\*m = 2 \* 3 = 6

So it is enough to just calculate the lps and check if the total length divides m.

----------------

<https://leetcode.com/problems/repeated-string-match/>

Graphical user interface, text, application

Description automatically generated