S	M	T	W	T	F	S
31					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

MAR 2024

String

'24 FEBRUARY

07th Week 043-323

12 MONDAY

String Module

To Provide flexibility for programmer Python introduced String module → Inbuilt module.

08

ascii_letters = list of all lower and upper case character.

09 ascii_lowercase = list of all lower case character.

ascii_uppercase = list of all upper case character.

10 digits = list of all the digits.

hex digit = list of all digit & char in hexadecimal.

11 Whitespace = list of all the white spaces.

Punctuation = list of all special character in keyboards.

12

Reverse a string.

01 (a) S= input()

(b) print(S[::-1])

02 rev = ""

for i in S:

rev = i + rev

03 Print (rev) ← ये 'i' beginning में जोड़ता है

a=97, A=65 } unicode Value

04 # given string is rotation of another string (or) not.

def fn(S1, S2):

05 if len(S1) != len(S2)

return False.

O(n²) 06 temp = S1 + S1

return temp.find(S2) != -1

S1 = 'ABCD'

S2 = 'CDAB'

S1+S2 = ABCD CDAB ×

S1+S2 = ABCDABCD

07

Palindrome

M1 return S == S[::-1]

M2 two pointer concepts.

Sort the String

def fn(S):

l = list(S)

l.sort()

return ''.join(l)

list to string

remove duplicate

def fn(s):

l = []

for ch in s:

if ch not in l:

l.append(ch)

return ''.join(l)

FEBRUARY 13 '24 TUESDAY
07th Week 044-322

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

FEB 2024

ord("a") = integer representation of character
chr(65) = character for ASCII value

String is subsequence of other (or) not

A subsequence of a string is obtained by removing zero (or) more character and whatever the character we pick, character that we don't remove, we need to put those characters in the same order as they appear in the original string (don't disturb relation position)

eg: 'ABC' = '', 'A', 'B', 'C', 'AB', 'AC', 'BC', 'ABC' → 2^n.

(a)
```
def subseq(S1, S2):
    i, j = 0, 0
    while i < len(S1) and j < len(S2):
        if S1[i] == S2[j]:
            j = j + 1
        i = i + 1
    if j == len(S2):
        return True
    else:
        return False
```
TC = O(n)

(b)
```
def subseq(S1, S2, m, n)
    if n == 0:
        return True
    if m == 0:
        return False
    if S1[m-1] == S2[n-1]:
        return subseq(S1, S2, m-1, n-1)
    else:
        return subseq(S1, S2, m-1, n)
```
TC = O(n)

Substring (or) not

Characters has to be in consecutive and there should same characters as they appear in given string.

eg: 'ABCD' → '', 'A', 'B', 'C', 'D', 'AB', 'AC', 'AD', 'BC', 'BD', 'CD', 'ABC', 'BCD'

```
S1 = 'geeks for geeks'
S2 = 'geeks'
print(S2 in S1) # True
print(S2 not in S1) # False
```

digit = '[0-9]'
spaces = '[ ]'
special char = '[^a-z A-Z 0-9]'

Count No of vowels

(a)
```
import re
def fun(s):
    return len(re.findall('[aeiou]', s))
s = input()
print(fun(s))
```

(b)
```
count = 0
for i in s:
    if i in 'aeiou':
        count += 1
return count
```

Count Consonants

```
import re
def fun(s):
    return len(re.findall('[^aeiou]', s))
s = input()
```

```
count = 0
for i in s:
    if i not in('aeiou'):
        count += 1
return count
```

Count alphabet

```
import re
def fun(s):
    return len(re.findall('[a-zA-Z]', s))
s = input()
```

^[aeiou] = matches string that starts with a vowel.
[^aeiou] = matches any character that is not vowel.

Count spaces

(a)
```
def fun(s)
    return s.count(' ')
```

(b) re.findall('[ ]')

Count No. of uppercase letters.

```
count = 0
for ch in s:
    if ch in string.ascii_uppercase:
        count += 1
print(count)
```

re.findall('[A-Z]')

Count No. of digit in string

```
count = 0
for ch in s:
    if ch in string.digits:
        count += 1
print(count)
```

String Function in Python.

(a) max() → return max character of it (ascii value maximum)
(b) min() → " min " " " (" " minimum)
(c) sorted() → " Sorted list with all character in ascending order.
(d) sorted (s, reverse = True) → return list with all character in descending order.
(e) ord() → Function to get unicode value.
(f) chr() → function to get character.
(g) in → membership operator.
(h) rstrip() → to remove spaces at right hand side.
(i) lstrip() → to remove spaces at left hand side.
(j) strip() → to remove spaces both sides.
(k) find() → return index of first occurrence of the given substring.
↳ If it is not available then we will get -1.
(l) index() → return index of first occurrence of the given substring.
↳ but if it is not available them we will get Value Error.
(m) count(substring) → to find the No. of occurrence of substring through out the string.
(n) replace (old-string, new-string) → to replacing old string.
(o) split(seperator) → We can split the given string according to specified seperator by using split method.
↳ by default it split around the spaces.
↳ return type is always list.
(p) ".join() → We can join a group of string w.r.t given seperator. [list to string]
↳ we can also convert list to string.
(q) upper() → to convert all character to upper case.
(r) lower() → " " " " " lower case.
(s) swapcase() → " " (lower ⇌ upper) case.
(t) title() → first character of every word is capital.
(u) capitalize() → only first character is capital. start
(v) startswith() → return True if the string with provided string.
(w) endswith() → " " " " " end " " " .
(x) isalpha() → (a to z, A to Z)
(y) isalnum() → (a to z, A to Z, 0 to 9)
(z) isdigit() → (0-9)
(aa) islower() → (a) isupper() (ab) istitle() (ac) isspace().

→ sort() method is available for list only because in python string is immutable.

→ Custom sorting with key

sorted(string, key = value/ str.lower() /str.upper()/ lambda x: ord(x))

FEBRUARY
15 '24
THURSDAY
07th Week
046-320

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

FEB 2024

Convert even index chas into uppercase & odd into lower case.

```
08  def fun(s):
        res = ''
09      for i in range(len(s)):
            if i%2 != 0:
10              res = res + s[i].upper()
            else:
11              res = res + s[i].
        return res.
12  print(fun(s))
```

reverse even index words

```
def fun(s):
02      l=[]
        index = 0
03      for i in s.split(' '):
            if index%2==0:                    → may be odd
04              l.append(i[::-1])             → may be lower case / upper case
            else:
05              l.append(i)
            index+=1
06      return ' '.join(l)
```

Middle character of string

```
def fun(s)
    if len(s)%2 !=0:
        return s[len(s)//2]
    else:
        return s[len(s)//2-1 : len(s)//2+1]
```

→ No. of times of occurrence also be same

Anagram (both string contain same characters)

M-1
```
def anag(S1, S2)
    if len(S1) != len(S2)
        return False
    S1 = sorted(S1)
    S2 = sorted(S2)
    return (S1 == S2)
```
O(nlogn)

Left most repeating character.

(a)
```
def fun(S):
    for i in range(len(S)):
        for J in range(i+1, len(str)):
            if St[i] == St[J]
                return i
    return -1
```
⇒ TC — O(n²)

(b) — abccbd # 1
```
char = 256
def ch(str):
    count = [0] * char
    for i in range(len(st
        count[ord(St[i])]
            += 1
    for i in range(len(st))
        if count[ord(St[i]
            > 1:
            return i
    return -1
Str = 'abccbd' # 1
```

Left most NONe repeating char.

(a)
```
def NONrep(str):
    for i in range(len(str)):
        flag = False
        for J in range(i+1, len(str)):
            if str[i] == str[J]:
                flag = True
                break
        if flag == False
            return i
    return -1.
```

(b)
```
char = 256
def nonrep(str):
    count = [0] * char
    for i in str:
        count[ord(i)] += 1
    for i in range(len(str)):
        if (count[ord(str[i])]
            == 1:
            return i
    return -1.
```

list to string = ''.join( )

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

FEB 2024

Word count — No. of spaces count करो and add (+1).

Reverse order of words.

i/p — Anubhav Kumar Gupta o/p — Gupta Kumar Anubhav

```
S= input(' ')
l = s.split()
l1 = l[::-1]
output = " ".join(l1)
print(output)
```

kth smallest element

```
def fun(arr,n,k):
    arr.sort()
    print(arr[k-1])
```

Palindrome or not?

M-2

```
S=input(" ")
low= 0
high = len(str)-1
while low < high:
    if S[low] != S[high]:
        print("no")
        break
    low = low+1
    high= high-1
else:
    print("yes")
```

Check if string is rotated (or) not.

```
def rot(s1, s2, k):
    if len(s1) != len(s2):
        return False

    left rotated = s1[k:]+s1[:k]
    right rotated = s1[-k:] + s1[:-k]

    return s2 == left rotated or s2 == right rotated.
```

Pangram = all 26 letters must be there.

```
def Pan(s):
    return len(set(s)) == 26
```

S	M	T	W	T	F	S
31					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

MAR 2024

Application : Search a word in word/file.
google search is also pattern search
DNA matching
Regular Expression.

FEBRUARY '24
08th Week
049-317

18
SUNDAY

Pattern searching

i/p = text = 'geeks for geeks"
pat = 'eks'
o/p = 2 10

i/p - text = 'aaaaaa'
pat = 'aaa'
o/p = 0 1 2

→ Pattern searching algorithms are sometimes also refferd to as a string search algorithm and are considered as a part of the String algorithm.
These algorithms are useful in the case of searching a string within another string.

Features :—
(a) Pattern searching algo should recognize familiar pattern quickly & accurately
(b) Recognize and classify unfamiliar patters.
(c) identify pattern even when Partly hidden.
(d) Recognize quickly with ease and with automaticity.

→ Naive : $\theta((n-m+1)*m)$ } No preProcessing
→ Naive when all character of pattern are distinct : $O(n)$ }

m → length path
n → text length
$1 \leq m \leq n$

→ Rabin Karp : $\theta((n-m+1)*m)$ } preProcess pattern
But better than Naive on Average

→ KMP : $O(n)$

Suffix tree : $O(m)$ preprocess text
→ ~~Data structure~~ (Trie)

Apart from KMP — Boyer Moore, 2 algorithms are also in linear time

FEBRUARY
19 '24
MONDAY
08th Week
050-316

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

FEB 2024

↳ Google search, Youtube search, Ctrl+F is also a pattern searching.

Pattern Search

i/p = txt = 'geeksforgeeks' o/p — 0 10
pat = 'geeks'

i/p = txt = "AAAAA" o/p — 0 1 2
pat = 'AAA'

⟹ txt = input()
pat = input()
pos = txt.find(pat)
while pos >= 0:
    print(pos)
    pos = txt.find(pat, pos+1)

Search the pattern from pos+1, so that this particular occurrence that I found already is not appeared again

Naive Pattern Searching

text = [A|B|A|A|B|C]
pat = [A|B|C]

[idea is slide the pattern over the text one by one

```
def naivepatt(text, pat):
    m = len(pat)
    n = len(txt)
    for i in range(n-m+1):
        J = 0
        while J<m:
            if pat[J] != txt[i+J]:
                break
            J = J+1
        if J==m:
            print(i, end=" ")
```

i=0 : J=1, J=2, J=3, J=4 break
i=1 : J=0 break
i=2 : J=0 break
i=3 : J=0, J=1, J=2, J=3

TC = O((n-m+1) x m)

Sorrow's best antidote is employment

Substring : consecutive character
Subsequence : May not be consecutive character.

Sliding Window Protocol

Longest Substring w/o Repeating character.

(I) abcdebe adf..

(II) ~~abcabc~~ ababc

l = 1 → a b a b c
len = 2 → ab ba ab bc
len = 3 → aba bab (abc)
len = 4 → abab babc
len = 5 → ababc

Initially window length = 0, ↳ all must be unique on that window then increase the length of that window by '1'

यदि कोई भी repeate कर रहा तो पहला repeating character को हटा दें

We have to find longest NON repeating character.

↳ We can solve using two pointer as well. — O(n2)

abcdee ↳ TC = O(n)

जब तक वो repeating character बाहर आ हो जाए, move (F)
इसलिए इस concept को sliding window बोलते हैं

F → first, S → second

abcdecbeadf ⇒ len = 0

abcdecbeadf ⇒ l = 1

abcdecbeadf ⇒ l = ~~2~~ ~~3~~ ~~4~~ 5

abcdecbeadf → decbeadf

remove it

(len = 6) ⇐ cbeadf

FEBRUARY
21 '24
08th Week
052-314
WEDNESDAY

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

FEB 2024

means दो pointer के बीच के सारे element unique हो।
आगे कोई भी unique नहीं, निकलता है जिसका आगर second pointer कर रहा तो first को आगे move करेंगा।

↳ longest substring with Distinct character.

i/p: str = "abcdabc" | i/p: str = "aaa" | i/p: str = ""
o/p: 4 | o/p: 1 | o/p = 0

Naive Approach

```
def distinct(str, i, J):
    visited = [0] * 256
    for K in range(i, J+1):
        if visited[ord(str[K])] == True:
            return False
        visited[ord(str[K])] = True
    return True

def longestDistinct(str):
    n = len(str)
    res = 0
    for i in range(n):
        for J in range(i, n):
            if distinct(str, i, J):
                res = max(res, J - i + 1)
    return res
```

TC = O(n^3)

We are considering all substring one by one and checking for each substring whether it contain all unique character or not.

Better Approach

```
def distinct(str):
    n = len(str)
    res = 0
    for i in range(n):
        visited = [0]*256
        for J in range(i, n):
            if visited[ord(str[J])]
                == True:
                break
            else:
                res = max(res, J-i+1
                visited[ord(str[J]
                    = True
    return res
```

TC = O(n^2)

The idea is to use window sliding. Whenever we see repetition, we remove the previous occurrence and slide the window.

Linear time Solution

This solution uses extra space to store the last indexes of already visited characters. The idea is to scan the string from left to right, keep track of the maximum length Non-Repeating character substring seen so far in res.
When we traverse the string, to know the length of current window we need two index ~~as ending~~.

(i) Ending index(j) – we consider current index as ending index.

(ii) Starting index(i) – It is same as previous window if current character was not present in the previous window.
To check if the current character was present in the previous window (or) not we store last index of every character in an array lastIndex[].
If lastIndex[str[j]] + 1 is more than previous start, then we update the start index i.
else we keep same i.

```
def longest(string):
    last_index = {}
    max_len = 0
    start_index = 0
    for i in range(0, len(string)):
        if string[i] in last_index:
            start_index = max(start_index, last_index[string[i]] + 1)
        max_len = max(max_len, i - start_index + 1)
        last_index[string[i]] = i
    return max_len
```

TC = O(n)

a b c a d b d
maxEnd(j): 1 2 3 3 4 4 2

(or)

```
def longest(str):
    res = 0
    prev = [-1]*256
    i = 0
    for j in range(len(str)):
        i = max(i, prev[ord(str[j])] + 1
        maxEnd = j - i + 1
        res = max(res, maxEnd)
        prev[ord(str[j])] = j
    return res.
```

TC = O(n)

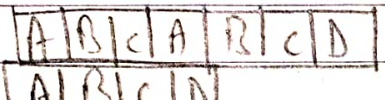
Assumption that all the characters in the patterns are distinct.

Improved Naive Pattern Searching for Distinct

08 i/p : text = "ABCABCD" pat = "ABCD"
09 o/p : 3

i/p : txt = "~~Geeks~~ GEEKSFOR GEEKS" pat = "EKS"
o/p : 2 10

10 Idea: If we see a mismatch after J matches, then we increment i by J.

11


12 ✓ ✓ ✓ ✗

i = 0 : J = 0, J = 1, J = 2, J = 3 ← mismatch, break (1st)

i = 3 : J = 0, J = 1, J = 2, J = 3, J = 4, Print(3) (II nd)

01
02
03

```
def distpatt search(txt, patt):
04      m,n = len(patt), len(txt)
        i = 0
05      while i <= (n-m):
            for j in range(m):
06              if patt[j] != txt[i+j]:
                    break
07          if j == m:
                print(i, end = " ")
            if j == 0:
                i += 1
            else:
                i += j
```

TC: O(n-m+1)

n = 4, n = 11
i = 0, J = 0, J = 1, J = 2, J = 3, break
i = 1 : J = 0, break
i = 4 : J = 0, J = 1, J = 2, break
i = 6 : J = 0, break
i = 7 : J = 0, J = 1, J = 2, J = 3, J = 4, Print(7)

FEBRUARY
23
FRIDAY
'24
08th Week
054-312

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

FEB 2024

Rabin Karp String Searching Algorithms.

i/p : txt = "abdabcbabc"	i/p : txt = "aaaaaa"	i/p : txt = "abcd"
pat = "abc"	pat = "aaa"	pat = "key 2"
o/p : 3 7	o/p = 0 1 2	o/p = Not found.

The naive String matching algorithms slides the pattern one by one. after each slide, it one by one checks character at the correct shift and if all character match then print the match.

Rabin-Karp algorithms also slides the pattern one by one. but unlike the naive algorithms, Rabin Karp algorithms matches the hash value of the pattern with the hash value of current substring of text, and if the hash value matches then only it starts matching individual characters.

So Rabin Karp algo needs to calculate hash value for following strings.

(i) Pattern itself

(ii) All the substring of the text of length m.

→ We compare hash value of pattern with current value of text, if value match then only we compare individual character and if individual character also match then we will print particular index value and move to next value.

0 1 2 3 4 5 6 7 8 9

txt = "abdabcbabc"

patt = "abc"

→ dog = god [Same hash Value.]

a = 1, b = 2, c = 3, d = 4, e = 5

p = hash value of pattern

t = hash value of current window of text.

→ p = (1+2+3) = 6

i = 0 : t = (1+2+4) = 7

i = 1 : t = (2+4+1) = 7

i = 2 : t = (4+1+2) = 7

i = 3 : t = (1+2+3) = 6 (match)

i = 4 : t = (2+3+2) = 7

i = 5 : t = (3+2+1) = 6 (Spurious hits)

i = 6 : t = (2+2+1) = 5

i = 7 : t = (1+2+3) = 6 (match)

{ Simple hash : Sum of values.
problem : spurious hits - [we match the hash value but character were not matching.

Print (3, 7)

Rolling hash: $t_{i+1} = t_i + txt[i+m] - txt[i]$ (window at index $i+1$; existing hash value; length of pattern; last character of i^{th} window)

08 Improved hash function. d=5, a=1, b=2, c=3, d=4, e=5

09 h("abc") = 1×d² + 2×d¹ + 3×d⁰ = (1×5² + 2×5¹ + 3×5⁰) = 38
h("dab") = 4×d² + 1×d¹ + 2×d⁰ = (4×5² + 1×5¹ + 2×5⁰) = 107

10

Rolling: $t_{i+1} = d(t_i - txt[i] \times d^{m-1}) + txt[i+m]$ [m → length of pattern]

11

eg txt = "abdabcbabc" | pat = "abc"

12 $t_0 = 1\times5^2 + 2\times5^1 + 4\times5^0 = 39$ | $p = 1\times5^2 + 2\times5 + 3\times1 = 38$

$t_1 = 5\times(t_0 - 1\times5^2) + 1 = 71$

01 $t_2 = 5\times(t_1 - 2\times5^2) + 2 = 107$

$t_3 = 5\times(t_2 - 4\times5^2) + 3 = 38$

02 Implementation

```
d = 256
03 def RKSearch(pat, txt, q):                    # q → prime no.
       m, n = len(pat), len(txt)
04     h = 1
       for i in range(1, m):                     # compute (d^(m-1))%q
           h = (h*d) % q
       p, t = 0, 0
05     for i in range(m):                        # compute p and t0
           p = ((p*d + ord(pat[i])) % q)
06         t = ((t*d + ord(txt[i])) % q)
       for i in range(n-m+1):
           if p == t:                            # check for spurious hit
               for j in range(m):
                   if txt[i+j] != pat[j]:
                       break
               else:
                   print(i, end=" ")
07         if i < (n-m):                         # compute t_{i+1} using t_i
               t = ((d*t - ord(txt[i])*h) + ord(txt[i+m])) % q
               if t < 0:
                   t = t + q
```

TC = O((n-m+1) * m))

FEBRUARY
25
SUNDAY
'24
09th Week
056-310

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

FEB 2024

Anagram Search

08

i/p: txt = "geeksforgeeks" pat = "frog"
o/p: yes [coz it's permutation is present in it], i/p = txt = "geeksforgeeks" pat = "rseek" o/p = NO

09

10

M-1 Naive Approach

11

```
Char = 256
def anagram(pat, text, i):
12      count=[0]*Char
        for J in range(len(pat)):
01          count[ord(pat[J])] += 1
            count[ord(text[i+J])] -= 1
02      for J in range(Char):
            if (count[J] != 0):
03              return False
        return True
04  def isPresent(txt, pat):
        n=len(txt)
05      m=len(pat)
        for i in range(n-m+1):
            if anagram(pat, txt, i):
06              return True
07
```

TC = O((n-m+1) x m)

M-2 Sliding window

```
Char = 256
def anagram(ct, cp):
    for i in range(Char):
        if ct[i] != cp[i]:
            return False
    return True

def present(txt, pat):
    n = len(txt)
    m = len(pat)
    ct = [0] * Char
    cp = [0] * Char
    for i in range(m):
        ct[ord(txt[i])] += 1
        cp[ord(pat[i])] += 1
    for i in range(m, n):
        if anagram(ct, cp):
            return True
        ct[ord(txt[i])] += 1
        ct[ord(txt[i-m])] -= 1
    return False
```

TC = O(n x char)

Lexicographic Rank

i/p: str = "BAC" , i/p: str = "CBA" , i/p: str = "DCBA"
o/p: 3 o/p: 6 o/p: 24

↳ First Sort all the character of "BAC" now generate all permutation of these 3 character in increasing / Lexicographic increasing order.

String	Rank
"ABC"	1
"ACB"	2
"BAC"	3 ← ✓
"BCA"	4
"CAB"	5
"CBA"	6 ← ✓

Efficient solution: Count lexicographically Smaller Strings + 1.

"STRING"

R _ _ _ _ _	S R _ _ _ _	S T I _ _ _
I _ _ _ _ _	S I _ _ _ _	S T N _ _ _
N _ _ _ _ _	S N _ _ _ _	S T G _ _ _
G _ _ _ _ _	S G _ _ _ _	
S T R G _ _	S T R I G _	

$4 \times \underline{|5} + 4 \times \underline{|4} + 3 \times \underline{|3} + 1 \times \underline{|2} + 1 \times \underline{|1} = 480 + 96 + 18 + 2 + 1$ = **597**

T.C
O(n×char)
= O(n)

```
Char = 256
def lexRank (str):
    res = 1
    n = len (str)
    mul = fact (n)
    count = [0] * char
    for i in range (n): count [ord (str[i])] += 1
    for i in range (1, char): count [i] += count [i-1]
    for i in range (n-1): mul = mul // (n-i)
        res += count [ord (str [i]) - 1] * mul
    for j in range [ord (st [i]), char):
        count [j] -= 1
    return res
```

String

0	..	1	0	1	..	0	1		1	1	T	..	0
		G	H	I		M	N		R	S	T		

after 2nd loop

0	..	1	1	2	..	2	3	..	4	5	6	..	6
		G	H	I		M	N		R	S	T		

mul = 6! = 720

i = 0 : mul = 120, res = 1 + 4 × 120 = 481

i = 1 ⇒ mul = 24, res = 481 + 4×24 = 577

i = 2, mul = 6, res = 577 + 3×6 = 595

i = 3 res = 595 + 2×1 = **597**

FEBRUARY
27
TUESDAY
'24
09th Week
058-308

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

FEB 2024

→ [Knuth Morris Pratt] → O(N) [WC]

KMP Algorithm for pattern searching.

Constructing longest proper Prefix Suffix array (LPS) is a prerequisite for understanding KMP algorithms.

Proper prefix of "abcd" → "", "a", "ab", "abc"

Suffix of "abcd" → "", "d", "cd", "bcd", "abcd"

i/p: str = "ababc"
o/p: lps[] = [0,0,1,2,0]
first entry is always going to be 0.

i/p: str = "aaaa" → for same string
o/p: lps = [0,1,2,3]

i/p = str = "ababab"
o/p = lps = [0,0,1,2,3,4]

i/p: str = "abcd" → for distinct string.
o/p: lps = [0,0,0,0]

↳ Check first Prefix the suffix then find longest and same.

eg: str = "abacabad"
lps[] = [0,0,1,0,1,2,3,0]

eg: str = "aabaab"
lps[] = [0,1,0,1,2,3]

Naive

```
def longPPS(str, n):
    for i in range(n-1, -1, -1):
        for j in range(i):
            if str[j] != str[n-i+j]:
                break
        else:
            return i
    return 0
```

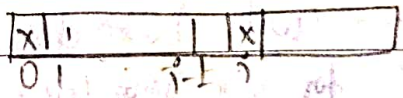
```
def fillLPS(str, lps):
    lps[0] = 0
    for i in range(1, len(str)):
        lps[i] = longPPS(str, i+1)
```

TC = O(n³)

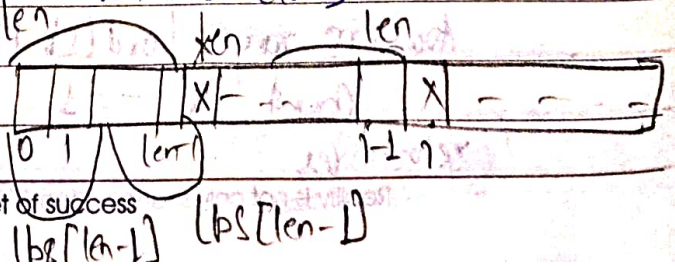
Efficient approach

① If len = lps[i-1] and str[len] and str[i] are same, then lps[i] = len + 1

② If str[i] and str[len] are not same
(a) If len == 0, lps[i] = 0



(b) else, we recursively apply lps[] len = lps[len-1] then compare str[i] with str[len]



0 1 2 3 4 5 6 7

Str = a b c b a b c a

lps[0] = 0, len = 0

08 lps[1] = 0, len = 0 [Case 2.a]

lps[2] = 0, len = 0 "

09 lps[3] ≠ 0, len = 0 "

lps[4] = 1, len = 1 [Case 1]

10 lps[5] = 2, len = 2 "

lps[6] = 3, len = 3 "

11 lps[7] = [Case 2.b]

len = lps[len-1] = lps[2] = 0

12 [Case 1] lps[i] = 1, len = 1

```
def fillLps(str, lps):
    lps[0] = 0
    i = 1
    len = 0
    while (i < n):
        if str[i] == str[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            if len == 0:
                lps[i] = 0
                i += 1
            else: len = lps[len-1]
```

TC = O(n)

01

02

The KMP Algorithm uses degenerating property (pattern having same sub-pattern appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP's algorithm is whenever we detect a mismatch (after some matches), we already know some of the character in the text of the next window.

03

04

05

We take advantage of this information to avoid matching the characters that we know will anyway match.

06

07