Q. Given a string representation of a number, we need to return whether there exist any permutation of the string that is divisible by 8.

n = 1 3 2 5 4 6 9 0 8 7 6 1 5 8 9 7 0,

now P(n) % 8 = 0 ?

Divisibility rule of 8 : If number formed using the last 3 digits is divisible by 8, then the said number is divisible by 8

↓

hash the digits and their corresponding frequencies.

↓ 0 – 999

for each ~~3 digit~~ number divisible by 8, check if the digit frequency in that number is less than or equal to frequency in hash table.

↓

If it is then return true, else false.

(false)
↑
prime length

Q. Given a string, return whether there exists a substring of length > 1, repeatedly concatenating which we can get the same string.

10 − 7 = 3 10 % 3 ≠ 0

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

a b c d d d a b c 9 − 3 = 6
0 0 0 0 0 0 1 2 3

Approach 1: append the same string to double the size. Start checking from i=1, to i=n-1 if the original string is found in the new string from index i to i+n, and this returns ~~true~~ false, for all iterations, then ~~true~~ false is returned and if in any case even 1 case ~~fails~~ passes, then ~~false~~ true is returned.

a b a b a b c a b a b a b a b a b c a b a b

$O(n+2n) \to O(3n)$
$\downarrow$
$O(n)$

a b a b a b a b a b a b

Approach 2: LPS applied to the original string
↓
$(n\%(n-x)==0?)$ if the last index value is subtracted from the length of the string and the subtracted value is a factor of the length, then we can say that such a string exists.

** proper prefix which is a proper suffix

$c_1\ c_2\ c_3\ c_4\ c_5$

$c_1 = c_2$
$c_2 = c_3$
$c_3 = c_4$
$c_4 = c_5$

all chars same
aaaa

$c_1 c_2 c_3 c_4 c_5 c_6$

$c_1 = c_3$
$c_2 = c_4$

$c_3 = c_5$
$c_4 = c_6$

$c_1 c_2 = c_3 c_4$
$c_3 c_4 = c_5 c_6$
$\therefore$
$c_1 c_2 = c_5 c_6$

Hence substrings can be said to repeat in the length of 2 passes

abc abcabcabcabcabc

passes

abcabcabcabc

approach 1

test cases

fails abcddabcabcddabc

fails aaaaaadaaaaaad

would require a pattern matching algorithm

aaaaaaa aaaaaaa (passes)

---

abcad abcad

$c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10}$

$c_1 = c_6$
$c_2 = c_7$
$c_3 = c_8$
$c_4 = c_9$
$c_5 = c_{10}$

$\therefore c_1 c_2 c_3 c_4 c_5 = c_6 c_7 c_8 c_9 c_{10}$

if one substring of length $< \frac{n}{2}$ = other substrings

then we can say that repeating one substring, we can get the org. string.

* If there exists such a string, then its length should be a factor of the length of the string.

↓

LPS : longest prefix which is suffix

↓

longest prefix which is repeated as suffix in the string.

↓

if the length of LPS = 6, for the comp. string,

then that means that the first l characters are identical to last l characters.

length of LPS → (0, n-1) n-1, when all characters are same, n-2 when they occur in a pair or a pair is repeated.

ABABABABAB, lps = 8

↓

first 2 characters are same as the last n-2 characters.

↓

ABCDABCDABCDABCDABCD

total characters : 20

value of LPS : 16

↓

so the first 16 characters are identical to the last 16 in order. It indicates that the first 4 characters are identical to the next 4 characters.

(because beginning 4 characters are beginning of prefix and next 4 characters are beginning of suffix)

↓

similarly next 4 characters are identical to next 4 characters because they are 5th to 8th characters of prefix and suffix respectively.

Scanned by CamScanner

* z algorithm

O(m+n)
↓
(easy to understand)

(i) creation of z array
(equivalent of LPS in KMP array.)

↓

at every index, the longest substring starting from that index which is also a prefix *

↓

lps gave us the length till that index, whereas z array gives us from that index

change in direction. ←

↓

a	a	b	x	a	a	y	a	a	b
0	1	0	0	2	1	0	3	1	0

→ z array

↓

a	a	a	a	a	a
0	5	4	3	2	1

O(n*m) ?
↓
naive approach to construct z array, using the conventional approach for finding the longest matching prefix for each index.

pattern matching:
concatenate ~~append~~ the pattern in front of the original string using a sentiment element

a	b	c	$	x	a	b	c	a	b	z	a	b	c
0	0	0	0	0	3	0	0	2	0	0	3	0	0

~~for each index~~
for each index, start from 0th index and assign count of same chars.

the indexes at which the value == length of the pattern string, then we can claim to have found the pattern in the text.

→ using this approach → pattern can be matched using O(1) space optimising this to O(m+n)

This approach helps us see if any prefix has been repeated in our string anywhere.

a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	1	0	0	4	1	0	0	0	8	1	0	0	5	1	0	0	1	0

↓

longest string which is a prefix of length 4

↓

so the next 3 characters are same as the next 3 characters of the 0th character of the prefix,

∴ Their values and these values would match and therefore an operation is not required for these characters again, and they can be copied from those indexes, given they don't cross the right limit of the matched substring

↓

if it does, then the match for the next index character has to be done beginning from the 0-th index

(**) [index + z [left ptr]] > r

ababababab
0020202020

abcabcabcabc
000300300300

aaaaaa
054321

↓

Z algorithm to solve the previous problem.

right limit

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
a a b x a a b x c a a b x a a b x a y
0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0 0 1 0

13+4 > 16

17

Search again

because the the end of the right limit is okay, but there could be more matches for these indexes.

z[i] is the length of the longest common prefix between s and the suffix of s starting at the ith position.

O(m+n) since we are moving in front only and not really coming back.

0 1 2 3 4 5
a a a a a a

k = 2
l = 1
r = 5

```
vector<int> z_function(string s){
    int n = s.length();
    vector<int> z(n);
    for(int i=1, l=0, r=0; i<n; i++){
    if( i<=r )
        z[i] = min(r-i+1, z[i-l]);
    while(i+z[i]<n && s[z[i]] == s[i+z[i]])
        ++z[i];
    if(i+z[i]-1 > r)
        l=i, r=i+z[i]-1;
    }
return z;
```

Scanned by CamScanner

Two

Q. strings containging charactees with & w/o brackets. we need to return, whether both strings are same or not.

↓

length of characters (variables will be same)

(a+b+c)-c
a+b
} invalid input

-(a+b+c)
-a-b-c
} valid input

(i) preprocess both strings to include + at places ~~except~~ no sign is present, so make a helper function for the same

(ii) Now use a steck and a m-count to create a new string for s1 by parsing the string character by character. If the ~~next~~ ~~character~~ current symbol is a '(' bracket, and the previous character is '-', then inc. m-count & push it to the stack. and for the following signs, check the m-count and accordingly append the new character as '+' and '-'. If) bracket is encountered pop the top of stack and if it is '-', then dec. the m-count, otherwise don't.