KMP AND Z ALGORITHMS

ISIS 2801

```
“strings”;
```

Pattern matching search

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at aliquam velit. Quisque scelerisque dapib**us** porta. Aenean maxim**us** luct**us** augue, sed consectetur magna fringilla ac. Morbi in pellentesque ris**us**. Integer orci nunc, sollicitudin nec pretium sed, efficitur non massa. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sem odio, volutpat vel pur**us** non, curs**us** sagittis sapien. Phasell**us** sed elit rhonc**us**, ullamcorper quam ut, vehicula odio. Proin gravida luct**us** bibendum.

All matches for pattern: **us**

HOW TO FIND ALL INSTANCES OF A GIVEN PATTERN

Pattern matching search

- us Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at aliquam velit. Quisque scelerisque dapibus porta. Aenean maximus luctus augue, sed consectetur magna fringilla ac. Morbi in pellentesque risus. Integer orci nunc, sollicitudin nec pretium sed, efficitur non massa. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sem odio, volutpat vel purus non, cursus sagittis sapien. Phasellus sed elit rhoncus, ullamcorper quam ut, vehicula odio. Proin gravida luctus bibendum.

Pattern matching search

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at aliquam velit. Quisque scelerisque dapibus porta. Aenean maximus luctus augue, sed consectetur magna fringilla ac. Morbi in pellentesque risus. Integer orci nunc, sollicitudin nec pretium sed, efficitur non massa. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sem odio, volutpat vel purus non, cursus sagittis sapien. Phasellus sed elit rhoncus, ullamcorper quam ut, vehicula odio. Proin gravida luctus bibendum.

Pattern matching search

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at aliquam velit. Quisque scelerisque dapibus porta. Aenean maximus luctus augue, sed consectetur magna fringilla ac. Morbi in pellentesque risus. Integer orci nunc, sollicitudin nec pretium sed, efficitur non massa. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sem odio, volutpat vel purus non, cursus sagittis sapien. Phasellus sed elit rhoncus, ullamcorper quam ut, vehicula odio. Proin gravida luctus bibendum.

110, 131

Pattern matching search

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at aliquam velit. Quisque scelerisque dapibus porta. Aenean maximus luctus augue, sed consectetur magna fringilla ac. Morbi in pellentesque risus. Integer orci nunc, sollicitudin nec pretium sed, efficitur non massa. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sem odio, volutpat vel purus non, cursus sagittis sapien. Phasellus sed elit rhoncus, ullamcorper quam ut, vehicula odio. Proin gravida luctus bibendum.

110, 131, 138

Pattern matching subsequence

```
def search(pat, txt):
    M = len(pat)
    N = len(txt)
    # A loop to slide pat[] one by one
    for i in range(N - M + 1):
        j = 0
        # For index i, check pattern match
        while(j < M):
            if (txt[i + j] != pat[j]):
                break
            j += 1
        if (j == M):
            print("Pattern found at index ", i)
```

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

CAN WE DO BETTER?

Knuth-Morris-Pratt algorithm

Definition KMP uses information from previous comparisons to reduce que number of comparisons **for match subsequences of the pattern**.

KMP algorithm

It0

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
S	E	V	E	N		S	E	E																											

KMP algorithm

It0

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
S	E	V	E	N		S	E	E																											

It1

KMP algorithm

It0

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
S	E	V	E	N		S	E	E																											

It1

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
	S	E	V	E	N		S	E	E																										

KMP algorithm

It0

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
S	E	V	E	N		S	E	E																											

It1

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
	S	E	V	E	N		S	E	E																										

It9

KMP algorithm

It0

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
S	E	V	E	N		S	E	E																											

It1

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
	S	E	V	E	N		S	E	E																										

It9

…

KMP algorithm

It10

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
										S	E	V	E	N		S	E	E																	

first match, check if the rest of the pattern matches too

KMP algorithm

It10

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
										S	E	V	E	N		S	E	E																	

first match, check if the rest of the pattern matches too

It10-9

0	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	31	32	33	34
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S
										S	E	V	E	N		S	E	E																

KMP algorithm

It10-9

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
										S	E	V	E	N		S	E	E																	

Already checked, shouldn't bother checking again

KMP algorithm

It10-9

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
										S	E	V	E	N		S	E	E																	

Already checked, shouldn't bother checking again

It10-9

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																		S	E	V	E	N		S	E	E									

KMP algorithm

It11 (16)

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																S	E	V	E	N		S	E	E											

It12 (19)

0	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	31	32	33	34	3
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																			S	E	V	E	N		S	E	E								

KMP algorithm

It11 (16)

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																S	E	V	E	N		S	E	E											

It12 (19)

0	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	31	32	33	34	3
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																			S	E	V	E	N		S	E	E								

KMP algorithm

It11 (16)

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																S	E	V	E	N		S	E	E											

It12 (19)

0	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	31	32	33	34	3
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																			S	E	V	E	N		S	E	E									

KMP algorithm

It17 (24) - 9

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																								S	E	V	E	N		S	E	E			

It18 (30)

0	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	31	32	33	34	35
I		D	I	S	L	I	K	E		S	E	V	E	N		S	E	V		B	U	T		S	E	V	E	N		S	E	E		S	E
																														S	E	V	E	N	

KMP algorithm

1. Search **sequentially** for a match of the pattern
2. If there is a match, search **sequentially** in the pattern
    1. If the pattern maches, get the position of the match and jump to the next **uncheck** position (offset by the **border**)
    2. If the pattern doesn't match, jump to the first unmatched position(offset by the **border**)
3. Return the match positions

KMP implementation

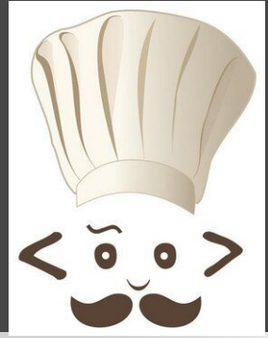
KMP implementation

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N]; // T = text, P = pattern
int b[MAX_N], t_len, p_len; // b = back table
void kmpPreprocess() { // call this before calling kmpSearch()
 int i = 0, j = -1; b[0] = -1; // starting values
 while (i < p_len) { // pre-process the pattern string P
  while (j >= 0 && P[i] != P[j])
   j = b[j]; // different, reset j using b
  i++; j++;
  b[i] = j;
 }
}
```

KMP implementation

```
void kmpSearch() {
 int i = 0, j = 0; // starting values
 while (i < t_len) { // search through string T
  while (j >= 0 && T[i] != P[j])
   j = b[j]; // different, reset j using b
  i++; j++;
  if (j == p_len) { // a match found
   printf("P is found at index %d in T\n", i - j);
   j = b[j]; // prepare j for the next possible match
  }
 }
}
```

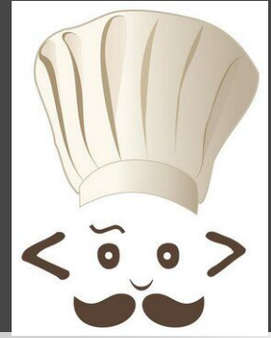
Shift The String



Shift The String

You are given two strings A and B of the same length. Each string contains N Lower case Latin character (from 'a' to 'z'). **A shift operation will remove the first character of a string and add the same character at the end of that string.** For example after you perform a shift operation on a string 'abcd', the new string will be 'bcda'. If you perform this operation two times, the new string will be 'cdab'. You need to use some (maybe none) shift operations on the string B to **maximize the length of the longest common prefix of A and B**. If more than one result can be found pick the one that use smallest number of shift operations.

Shift The String



Input
The first line of the input contains a single integer N. The second and the third line contains the string A and B respectively.

Output
Contains a single integer which is the number of shift operations.

Example
Input:
5
ccadd
bddcc

Output:
3

Shift The String

Do it yourself

Shift The String

Search (prefixes of) A in BB

Do it yourself

Shift The String

Search (prefixes of) A in BB

Do it yourself

```
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int n; string a, b;
    cin >> n >> a >> b;
    b += b;
   cout << KMPMultipleTimes(b, a) << endl;
    return 0;
}
```

Shift The String

Search (prefixes of) A in BB

Do it yourself

```
int KMPMultipleTimes (string text, string pattern) {
    int best = 0, result = 0, ans = 0;
    vector<int> lps = constructTempArray(pattern);
    int j = 0, i = 0;
    // i --> text, j --> pattern
    while (i < (int) text.length()) {
        if (text[i] == pattern[j]) ++i, ++j, best++;
        else {
            best = 0;
            if (j != 0) j = lps[j - 1];
            else ++i;
        }
        if (best > result) result = best, ans = i - j;
        if (j == (int) pattern.length()) {
            break;
        }
    }
    return ans;
}
```

Shift The String

Search (prefixes of) A in BB

Do it yourself

```
vector<int> constructTempArray(string pattern) {
    vector<int> lps(pattern.length());
    int index = 0;
    for (int i = 1; i < (int) pattern.length(); ) {
        if (pattern[i] == pattern[index])
            lps[i] = index + 1, ++index, ++i;
        else {
            if (index != 0) index = lps[index - 1];
            else lps[i] = index, ++i;
        }
    }
    return lps;
}
```

Z

Find the longest prefix substring

Given a string **s**, fin the length of the longest substring of **s** that starts at a given position, and is a prefix of **s**

easy for the first position

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
	-	0	0	5	0	0	2	0	2	0

HOW TO BUILD THE ARRAY

Z algorithm

	0	1	2	3	…			k-1	k	…								n
s	s_0	s_1	s_2	s_3	…			s_{k-1}	s_k									s_n
z	-	p_1	p_2	p_3	…			p_{k-1}	-1									-1
w						**x**					**y**							

window containing the prefix of s

Z algorithm

1.

	0	1	2	3	…								k-1	k	…			n
s	s_0	s_1	s_2	s_3	…								s_{k-1}	s_k				s_n
z	-	p_1	p_2	p_3	…								p_{k-1}	-1				-1
w						**x**					**y**							

No information, we need to calculate the prefix. Calculate using the naïve approach

Z algorithm

y≥k and k + z[k−x]≤y

2.

	0	1	2	3	…			k-1	k	…								n
s	s_0	s_1	s_2	s_3	…			s_{k-1}	s_k									s_n
z	-	p_1	p_2	p_3	…			p_{k-1}	-1									-1
w						**x**						**y**						

k−x

$$z[k] = z[k-x]$$

Z algorithm

y≥k and k + z[k−x]>y

3.

	0	1	2	3	…			k-1	k	…								n
s	s_0	s_1	s_2	s_3	…			s_{k-1}	s_k									s_n
z	-	p_1	p_2	p_3	…			p_{k-1}	-1									-1
w						**x**						**y**						

$$z[k] \geq y - k + 1$$

use the naïve approach for y−k+1 and y+1

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	-	-	-	-	-	-	-	-	-

k	1
x	–
y	–
k–x	–

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	-	-	-	-	-	-	-	-	-

compare the array [B]
with the prefix [A]

k	1
x	–
y	–
k–x	–

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	**0**	-	-	-	-	-	-	-	-

k	2
x	–
y	–
k–x	–

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	**0**	-	-	-	-	-	-	-	-

compare the array [C]
with the prefix [A]

k	2
x	–
y	–
k–x	–

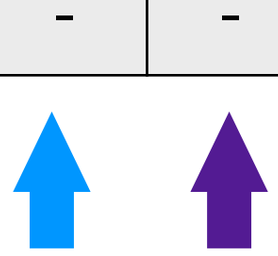
Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	**0**	-	-	-	-	-	-	-

k	3
x	–
y	–
k–x	–

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	**0**	-	-	-	-	-	-	-



[A] with prefix [A]

k	3
x	–
y	–
k–x	–

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	**0**	-	-	-	-	-	-	-

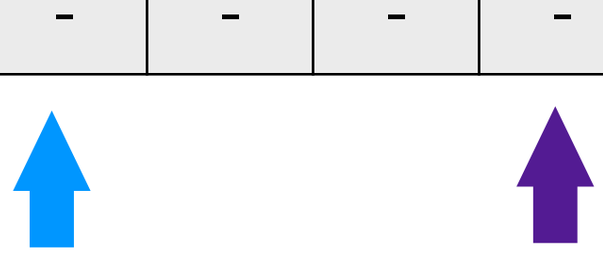
[A] with prefix [A]

[B] with prefix [B]

k	3
x	–
y	–
k–x	–

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	**0**	-	-	-	-	-	-	-



[A] with prefix [A]

[B] with prefix [B]

[C] with prefix [C]

k	3
x	–
y	–
k–x	–

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	**0**	-	-	-	-	-	-	-

```
[A] with prefix [A]
[B] with prefix [B]
[C] with prefix [C]
[B] with prefix [B]
```

k	3
x	–
y	–
k−x	–

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	**0**	-	-	-	-	-	-	-

	3
x	–
y	–
k–x	–

```
[A] with prefix [A]
[B] with prefix [B]
[C] with prefix [C]
[B] with prefix [B]
[A] with prefix [C]
```

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	0	**5**	-	-	-	-	-	-

Case 2. `z[4] = z[1]`

k	4
x	**3**
y	**7**
k−x	1

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	0	5	**0**	-	-	-	-	-

Case 2. `z[5] = z[2]`

k	5
x	**3**
y	**7**
k−x	2

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	0	5	0	**0**	-	-	-	-

Case 3. compare
y−k+1=s [2]
y+1=s [8]

k	6
x	**3**
y	**7**
k−x	3

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	0	5	0	0	**2**	-	-	-

Case 2.
$z[k]=z[k-x]$

k	7
x	**6**
y	**7**
k–x	1

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	0	5	0	0	2	**0**	-	-

Case 1. naïve

k	8
x	**6**
y	**7**
k−x	2

Example

	0	1	2	3	4	5	6	7	8	9
s	A	B	C	A	B	C	A	B	A	B
z	-	0	0	5	0	0	2	0	2	**0**

Case 2.

k	9
x	**8**
y	**9**
k−x	1

Z algorithm

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

Z algorithm

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

Omer and Strings

Today while studying Omar reads many words in books and references. He feels bored enough to stop reading he has noticed something strange. All the words in some statement can be read the same from left to right and from right to left. Later, Omar discovered that this type of strings is called palindromic strings.

After some thinking Omar wants to create a new type of strings and gives that type a name derived from his name, so he invents a new type of strings and calls it omeric strings. Omar's friend Hazem loves prefixes and Eid loves suffixes so Omar will take this into consideration while inventing the new type. To **make a string omeric from a string s you should concatenate the longest palindromic suffix with the longest palindromic prefix**.

Then Omar wants to know **how many times each prefix of his omeric string can be found in this string as a substring**. Substring of the string can be defined by two indices L and R and equals $s_L s_{L+1} \ldots s_R$.

Omer and Strings

Input:
The first line of the input contains a string s of length N, $1 \le N \le 10^5$.
The given string consists only of lowercase characters.
Output:
Print the omeric string in the first line. Print the frequency of each prefix in the second line.

SAMPLE INPUT
aabb

SAMPLE OUTPUT
bbaa
2 1 1 1

Explanation
Longest palindromic suffix equals "bb". Longest palindromic prefix equals "aa". Omeric string is "bbaa".
Only the first prefix "b" has appeared 2 times as a substring, while "bb", "bba", "bbaa" have appeared only 1 time in the omeric string.

Omer and Strings

Do it yourself

Omer and Strings

Do it yourself

```cpp
int fail1 [200000 + 10]; int fail2 [200000 + 10]; int fail3 [200000 + 10];
int freq [200000 + 10];

int main() {
    string s, c , prefix = "", suffix = "", omeric = "";
    cin >> s;
    c = s;
    reverse(c.begin(), c.end());
    prefix = s + "#" + c;
    suffix = c + "#" + s;
    for(int i = 1, k = 0; i < prefix.size(); i++){
        while(k > 0 && prefix[i] != prefix[k]) k = fail1[k - 1];
        if(prefix[i] == prefix[k]) k++;
        fail1[i] = k;
    }
    for(int i = 1, k = 0; i < suffix.size(); i++){
        while(k > 0 && suffix[i] != suffix[k]) k = fail2[k - 1];
        if(suffix[i] == suffix[k]) k++;
        fail2[i] = k;
    }
    string res1 = s.substr(s.size() - fail2[suffix.size() - 1], fail2[suffix.size() - 1]);
    string res2 = s.substr(0 , fail1[prefix.size() - 1]);

    omeric = res1 + res2;
    s = omeric;
    cout << omeric << "\n";

    for(int i = 1, k = 0; i < s.size(); i++){
        while(k > 0 && s[i] != s[k]) k = fail3[k - 1];
        if(s[i] == s[k]) k++;
        fail3[i] = k;
    }
    for(int i = 0; i < s.size(); i++) freq[ fail3[i] ]++;
    for(int i = s.size() - 1; i > 0; i--) freq[ fail3[i - 1] ] += freq[i];
    for(int i = s.size(); i > 0; i--) freq[i]++;
    for(int i = 1; i <= s.size(); i++) cout << freq[i] << " ";

    return 0;
}
```