#include <cstdio>

#include <algorithm>

#include <cstring>

using namespace std;

const int

MAXN = 1 << 17;

char text[MAXN];

int sa[MAXN];

bool sufCmp(int i, int j)

{

return strcmp(text+i, text+j) < 0;

}

int main()

{

scanf( "%s", &text );

n = strlen(text);

for (int i = 0; i < n; ++i)

sa[i] = i;

sort(sa, sa + n, sufCmp);

for (int i = 0; i < n; ++i)

printf( "%s\n", text + sa[i] );

return 0;

}

#include <stdio.h>

#include <string>

#include <string.h>

#include <algorithm>

#define MAX\_N 90001

using namespace std;

typedef long long lld;

int n, t;

char s[MAX\_N];

int SA[MAX\_N], LCP[MAX\_N];

int tmp[MAX\_N], poz[MAX\_N];

inline bool suff\_compare(int i, int j)

{

if (poz[i] != poz[j]) return (poz[i] < poz[j]);

i += t;

j += t;

if (i < n && j < n) return (poz[i] < poz[j]);

else return (i > j);

}

inline void buildSA()

{

for (int i=0; i<n; i++)

{

SA[i] = i;

poz[i] = s[i];

}

for (t=1; ; t\*=2)

{

sort(SA, SA + n, suff\_compare);

for (int i=0; i<n-1; i++) tmp[i+1] = tmp[i] + suff\_compare(SA[i],SA[i+1]);

for (int i=0; i<n; i++) poz[SA[i]] = tmp[i];

if (tmp[n-1] == n-1) break;

}

}

inline void buildLCP()

{

for (int i=0, k=0; i<n; i++)

{

if (poz[i] != n-1)

{

int j = SA[poz[i]+1];

while (s[i+k] == s[j+k]) k++;

LCP[poz[i]] = k;

if (k) k--;

}

}

}

int main()

{

gets(s);

n = strlen(s);

buildSA();

buildLCP();

for (int i=0;i<n;i++) printf("%d ",SA[i]);

printf("\n");

for (int i=0;i<n;i++) printf("%d ",LCP[i]);

return 0;

}

// lexicographic order for pairs

inline bool leq(int a1, int a2, int b1, int b2) {

return(a1 < b1 || a1 == b1 && a2 <= b2);

}

// and triples

inline bool leq(int a1, int a2, int a3, int b1, int b2, int b3) {

return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3));

} // and triples

// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r

static void radixPass(int\* a, int\* b, int\* r, int n, int K) {// count occurrences

int\* c = new int[K + 1]; // counter array

for (int i = 0; i <= K; i++) c[i] = 0; // reset counters

for (int i = 0; i < n; i++) c[r[a[i]]]++; // count occurrences

for (int i = 0, sum = 0; i <= K; i++) // exclusive prefix sums

{

int t = c[i];

c[i] = sum;

sum += t;

}

for (int i = 0; i < n; i++) b[c[r[a[i]]]++] = a[i]; // sort

delete [] c;

}

// find the suffix array SA of s[0..n-1] in {1..K}ˆn

// require s[n]=s[n+1]=s[n+2]=0, n>=2

void suffixArray(int\* s, int\* SA, int n, int K) {

int n0 = (n+2)/3, n1 = (n+1)/3, n2 = n/3, n02 = n0+n2;

int\* s12 = new int[n02+3]; s12[n02] = s12[n02+1] = s12[n02+2] = 0;

int\* SA12 = new int[n02+3]; SA12[n02] = SA12[n02+1] = SA12[n02+2] = 0;

int\* s0 = new int[n0];

int\* SA0 = new int[n0];

// generate positions of mod 1 and mod 2 suffixes

// the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1

for (int i=0, j=0; i < n + (n0-n1); i++)

if (i%3 != 0) s12[j++] = i;

// lsb radix sort the mod 1 and mod 2 triples

radixPass(s12 , SA12, s+2, n02, K);

radixPass(SA12, s12 , s+1, n02, K);

radixPass(s12 , SA12, s , n02, K);

// find lexicographic names of triples

int name = 0, c0 = -1, c1 = -1, c2 = -1;

for (int i = 0; i < n02; i++) {

if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {

name++;

c0 = s[SA12[i]];

c1 = s[SA12[i]+1];

c2 = s[SA12[i]+2];

}

if (SA12[i]%3 == 1) s12[SA12[i]/3] = name; // left half

else s12[SA12[i]/3 + n0] = name; // right half

}

// recurse if names are not yet unique

if (name < n02) {

suffixArray(s12, SA12, n02, name);

// store unique names in s12 using the suffix array

for (int i = 0; i < n02; i++) s12[SA12[i]] = i + 1;

} else // generate the suffix array of s12 directly

for (int i = 0; i < n02; i++) SA12[s12[i] - 1] = i;

// stably sort the mod 0 suffixes from SA12 by their first character

for (int i = 0, j = 0; i < n02; i++)

if (SA12[i] < n0) s0[j++] = 3\*SA12[i];

radixPass(s0, SA0, s, n0, K);

// merge sorted SA0 suffixes and sorted SA12 suffixes

for (int p = 0, t = n0-n1, k = 0; k < n; k++) {

#define GetI() (SA12[t] < n0 ? SA12[t] \* 3 + 1 : (SA12[t] - n0) \* 3 + 2)

int i = GetI(); // pos of current offset 12 suffix

int j = SA0[p]; // pos of current offset 0 suffix

if (SA12[t] < n0 ? // different compares for mod 1 and mod 2 suffixes

leq(s[i], s12[SA12[t] + n0], s[j], s12[j/3]) :

leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))

{// suffix from SA12 is smaller

SA[k] = i; t++;

if (t == n02) // done --- only SA0 suffixes left

for (k++; p < n0; p++, k++) SA[k] = SA0[p];

} else {// suffix from SA0 is smaller

SA[k] = j; p++;

if (p == n0) // done --- only SA12 suffixes left

for (k++; t < n02; t++, k++) SA[k] = GetI();

}

}

delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;

}

int main() {

return 0;

}