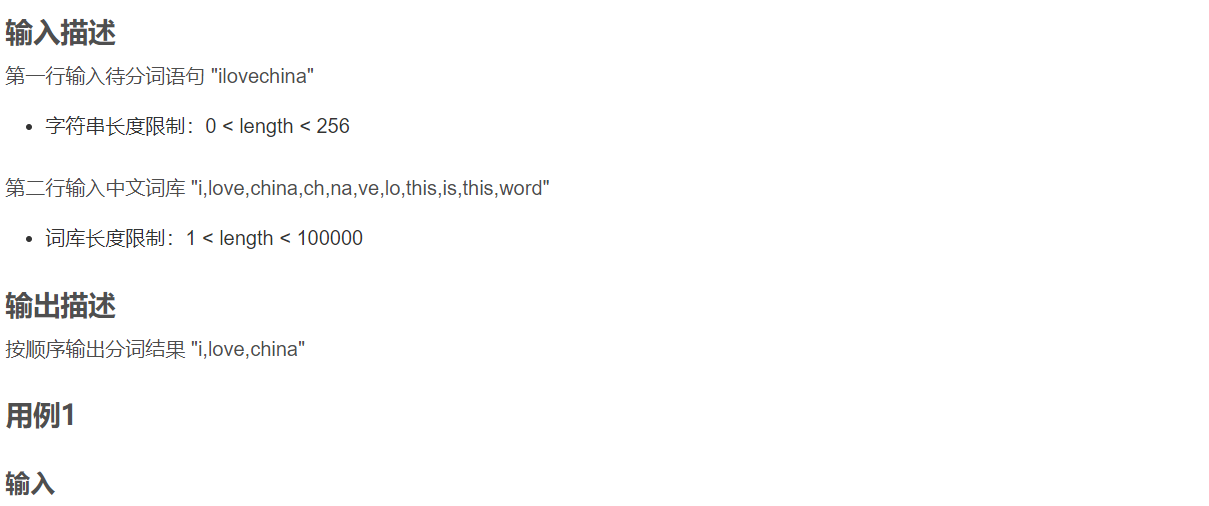
# **E卷-中文分词模拟器[200分]（ Java | Python3 | C++ | C语言 | JsNode | Go ）**



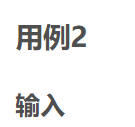


ilovechina

i,love,china,ch,na,ve,lo,this,is,the,word



i,love,china

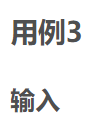


**iat**

**i,love,china,ch,na,ve,lo,this,is,the,word,beauti,tiful,ful**



i,a,t



ilovechina,thewordisbeautiful

i,love,china,ch,na,ve,lo,this,is,the,word,beauti,tiful,ful



i,love,china,the,word,is,beauti,ful





import java.util.\*;

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// 读取待分词语句

String s = sc.next();

// 读取词库并分割为数组

String []p = sc.next().split(",");

// 创建一个HashSet用于存储词库，便于快速查找

Set<String> st = new HashSet<>();

// 将词库中的词逐个添加到HashSet中

for(String x : p) st.add(x);

// 初始化结果字符串

String ans = "";

// 分词标记，用于标志是否应该加逗号

int f = 0;

// 遍历待分词字符串

for(int i = 0; i < s.length();) {

// 如果当前字符为标点符号，直接添加到结果中，并跳过该字符

if(s.charAt(i) > 'z' || s.charAt(i) < 'a') {

ans += s.charAt(i++);

f = 0;

continue;

}

// 首次处理字符时不加逗号，其余情况下添加逗号

if(f == 0) f = 1;

else ans += ",";

// 初始化当前最大分词索引和临时分词字符串

int idx = i + 1;

String tmp = "" + s.charAt(i);

StringBuilder now = new StringBuilder("");

now.append(s.charAt(i));

// 从当前字符开始尝试最大匹配词

for(int j = i + 1; j < s.length(); j++) {

now.append(s.charAt(j));

// 如果当前拼接的单词存在于词库中，则更新最大匹配索引和临时分词字符串

if(st.contains(now.toString())) {

idx = j + 1;

tmp = now.toString();

}

}

// 移动起始索引到最大匹配索引位置

i = idx;

// 将当前临时分词字符串添加到结果中

ans += tmp;

}

// 输出最终分词结果

System.out.println(ans);

}

}



def exact\_word\_segmentation(s, p):

import sys

input = sys.stdin.read

# 读取词库并分割为数组

dictionary = p.split(',')

# 创建一个Set用于存储词库，便于快速查找

st = set(dictionary)

# 初始化结果字符串

ans = []

# 分词标记，用于标志是否应该加逗号

f = False

# 遍历待分词字符串

i = 0

while i < len(s):

# 如果当前字符为标点符号，直接添加到结果中，并跳过该字符

if not ('a' <= s[i] <= 'z'):

ans.append(s[i])

i += 1

f = False

continue

# 首次处理字符时不加逗号，其余情况下添加逗号

if f:

ans.append(',')

f = True

# 初始化当前最大分词索引和临时分词字符串

idx = i + 1

tmp = s[i]

longest\_match = s[i]

# 从当前字符开始尝试最大匹配词

for j in range(i + 1, len(s)):

longest\_match += s[j]

# 如果当前拼接的单词存在于词库中，则更新最大匹配索引和临时分词字符串

if longest\_match in st:

idx = j + 1

tmp = longest\_match

# 移动起始索引到最大匹配索引位置

i = idx

# 将当前临时分词字符串添加到结果中

ans.append(tmp)

# 输出最终分词结果

return ''.join(ans)

# 测试用例

s = input()

dictionary = input()

print(exact\_word\_segmentation(s, dictionary))



#include <iostream>

#include <vector>

#include <unordered\_set>

#include <string>

#include <sstream>

using namespace std;

vector<string> split(const string& str, char delimiter) {

vector<string> tokens;

string token;

istringstream tokenStream(str);

while (getline(tokenStream, token, delimiter)) {

tokens.push\_back(token);

}

return tokens;

}

string exact\_word\_segmentation(string s, string p) {

// 读取词库并分割为数组

vector<string> dictionary = split(p, ',');

// 创建一个Unordered\_Set用于存储词库，便于快速查找

unordered\_set<string> st(dictionary.begin(), dictionary.end());

// 初始化结果字符串

string ans = "";

// 分词标记，用于标志是否应该加逗号

bool f = false;

// 遍历待分词字符串

for (size\_t i = 0; i < s.length();) {

// 如果当前字符为标点符号，直接添加到结果中，并跳过该字符

if (s[i] > 'z' || s[i] < 'a') {

ans += s[i++];

f = false;

continue;

}

// 首次处理字符时不加逗号，其余情况下添加逗号

if (f) ans += ",";

f = true;

// 初始化当前最大分词索引和临时分词字符串

size\_t idx = i + 1;

string tmp = string(1, s[i]);

string longest\_match = tmp;

// 从当前字符开始尝试最大匹配词

for (size\_t j = i + 1; j < s.length(); ++j) {

longest\_match += s[j];

// 如果当前拼接的单词存在于词库中，则更新最大匹配索引和临时分词字符串

if (st.find(longest\_match) != st.end()) {

idx = j + 1;

tmp = longest\_match;

}

}

// 移动起始索引到最大匹配索引位置

i = idx;

// 将当前临时分词字符串添加到结果中

ans += tmp;

}

// 输出最终分词结果

return ans;

}

// 测试用例

int main() {

string s;

cin>>s;

string dictionary;

cin>>dictionary;

cout << exact\_word\_segmentation(s,

dictionary) << endl;

return 0;

}



#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_LENGTH 256

#define MAX\_WORDS 100000

// 用于分割字符串的函数

char\*\* split(const char\* str, char delimiter, int\* size) {

char\*\* result = malloc(MAX\_WORDS \* sizeof(char\*));

char\* token;

char\* str\_copy = strdup(str);

char\* rest = str\_copy;

int count = 0;

while ((token = strtok\_r(rest, ",", &rest))) {

result[count++] = strdup(token);

}

\*size = count;

free(str\_copy);

return result;

}

bool contains(const char\* word, char\*\* dictionary, int dict\_size) {

for (int i = 0; i < dict\_size; ++i) {

if (strcmp(word, dictionary[i]) == 0) {

return true;

}

}

return false;

}

void exact\_word\_segmentation(char\* s, char\* p) {

// 读取词库并分割为数组

int dict\_size;

char\*\* dictionary = split(p, ',', &dict\_size);

// 初始化结果字符串

char ans[MAX\_LENGTH \* 2] = "";

// 分词标记，用于标志是否应该加逗号

bool f = false;

// 遍历待分词字符串

for (int i = 0; i < strlen(s);) {

// 如果当前字符为标点符号，直接添加到结果中，并跳过该字符

if (s[i] > 'z' || s[i] < 'a') {

strncat(ans, &s[i], 1);

i++;

f = false;

continue;

}

// 首次处理字符时不加逗号，其余情况下添加逗号

if (f) strcat(ans, ",");

f = true;

// 初始化当前最大分词索引和临时分词字符串

int idx = i + 1;

char tmp[256] = "";

strncat(tmp, &s[i], 1);

char longest\_match[256] = "";

strncat(longest\_match, &s[i], 1);

// 从当前字符开始尝试最大匹配词

for (int j = i + 1; j < strlen(s); ++j) {

int len = strlen(longest\_match);

longest\_match[len] = s[j];

longest\_match[len + 1] = '\0';

// 如果当前拼接的单词存在于词库中，则更新最大匹配索引和临时分词字符串

if (contains(longest\_match, dictionary, dict\_size)) {

idx = j + 1;

strcpy(tmp, longest\_match);

}

}

// 移动起始索引到最大匹配索引位置

i = idx;

// 将当前临时分词字符串添加到结果中

strcat(ans, tmp);

}

// 输出最终分词结果

printf("%s\n", ans);

for (int i = 0; i < dict\_size; ++i) {

free(dictionary[i]);

}

free(dictionary);

}

// 测试用例

int main() {

char s[260];

scanf("%s", s);

char dictionary[100001];

scanf("%s", dictionary);

exact\_word\_segmentation(s, dictionary);

return 0;

}



const rl = require("readline").createInterface({ input: process.stdin, output: process.stdout, });

var iter = rl[Symbol.asyncIterator]();

const readline = async () => (await iter.next()).value;

void async function () {

function exactWordSegmentation(s, p) {

// 读取词库并分割为数组

const dictionary = p.split(',');

// 创建一个Set用于存储词库，便于快速查找

const st = new Set(dictionary);

// 初始化结果字符串

let ans = '';

// 分词标记，用于标志是否应该加逗号

let f = false;

// 遍历待分词字符串

for (let i = 0; i < s.length;) {

// 如果当前字符为标点符号，直接添加到结果中，并跳过该字符

if (s[i] > 'z' || s[i] < 'a') {

ans += s[i++];

f = false;

continue;

}

// 首次处理字符时不加逗号，其余情况下添加逗号

if (f) ans += ",";

f = true;

// 初始化当前最大分词索引和临时分词字符串

let idx = i + 1;

let tmp = s[i];

let longestMatch = s[i];

// 从当前字符开始尝试最大匹配词

for (let j = i + 1; j < s.length; j++) {

longestMatch += s[j];

// 如果当前拼接的单词存在于词库中，则更新最大匹配索引和临时分词字符串

if (st.has(longestMatch)) {

idx = j + 1;

tmp = longestMatch;

}

}

// 移动起始索引到最大匹配索引位置

i = idx;

// 将当前临时分词字符串添加到结果中

ans += tmp;

}

// 输出最终分词结果

return ans;

}

// 测试用例

let s = (await readline());

const dictionary = (await readline());

console.log(exactWordSegmentation(s, dictionary));

}()



package main

import (

"fmt"

"strings"

)

// 分割字符串的工具函数

func split(str, delimiter string) []string {

return strings.Split(str, delimiter)

}

// 判断字典中是否存在指定单词

func contains(word string, dictionary []string) bool {

for \_, dictWord := range dictionary {

if word == dictWord {

return true

}

}

return false

}

func exactWordSegmentation(s, p string) string {

// 读取词库并分割为数组

dictionary := split(p, ",")

// 初始化结果字符串

var ans strings.Builder

// 分词标记，用于标志是否应该加逗号

f := false

i := 0

// 遍历待分词字符串

for i < len(s) {

// 如果当前字符为标点符号，直接添加到结果中，并跳过该字符

if s[i] > 'z' || s[i] < 'a' {

ans.WriteByte(s[i])

i++

f = false

continue

}

// 首次处理字符时不加逗号，其余情况下添加逗号

if f {

ans.WriteByte(',')

}

f = true

// 初始化当前最大分词索引和临时分词字符串

idx := i + 1

tmp := string(s[i])

longestMatch := string(s[i])

// 从当前字符开始尝试最大匹配词

for j := i + 1; j < len(s); j++ {

longestMatch += string(s[j])

if contains(longestMatch, dictionary) {

idx = j + 1

tmp = longestMatch

}

}

// 移动起始索引到最大匹配索引位置

i = idx

// 将当前临时分词字符串添加到结果中

ans.WriteString(tmp)

}

// 输出最终分词结果

return ans.String()

}

func main() {

var s string

var dictionary string

fmt.Scanln(&s)

fmt.Scanln(&dictionary)

fmt.Println(exactWordSegmentation(s, dictionary))

}