# **E卷-查找接口成功率最优时间段[100分]（ Java | Python3 | C++ | C语言 | JsNode | Go ）**

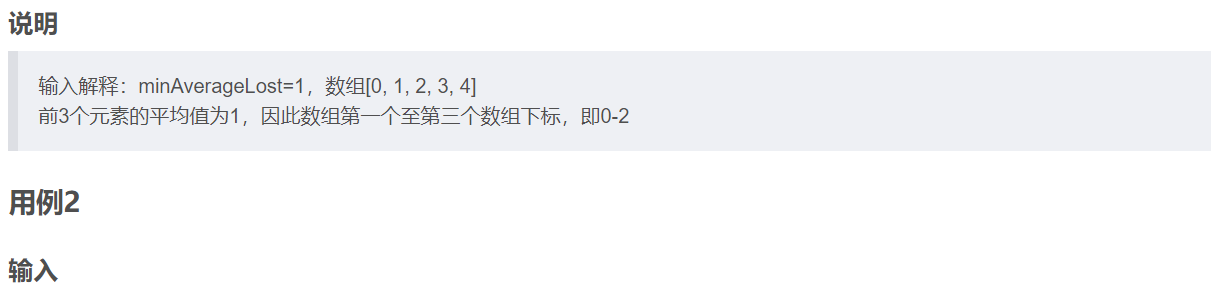


1

0 1 2 3 4



0-2

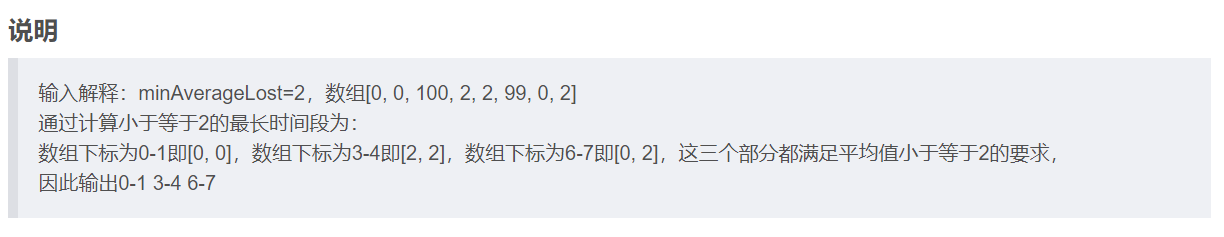


2

0 0 100 2 2 99 0 2



0-1 3-4 6-7





import java.util.\*;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int k = scanner.nextInt(); // 读取第一个输入值，表示平均失败率容忍值minAverageLost

scanner.nextLine();

String[] parts =

scanner.nextLine().split(" "); // 读取第二行输入，并分割成字符串数组

int[] a = Arrays.stream(parts).mapToInt(

Integer::parseInt).toArray(); // 转换成整数数组

int target =

0; // 初始化target为0，用于记录找到的最长时间段的长度

int n = a.length; // n表示数组的长度

List<int[]> res = new ArrayList<>(); // 初始化结果列表

// 第一个双层循环用来找到符合条件的最长时间段的长度

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

int now = 0; // 初始化当前段的失败率和

for (int j = i; j < n; j++) {

now += a[j]; // 累加当前段的失败率

int sz = 1 + j - i; // 计算当前段的长度

if (now <= sz \* k) { // 检查当前段的平均失败率是否满足条件

target = Math.max(target, sz); // 更新最长时间段的长度

}

}

}

// 如果没有找到符合条件的时间段，输出NULL并退出程序

if (target <= 0) {

System.out.println("NULL");

return;

}

// 第二个双层循环用来收集所有符合最长时间段条件的下标对

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

int now = 0; // 初始化当前段的失败率和

for (int j = i; j < n; j++) {

now += a[j]; // 累加当前段的失败率

int sz = 1 + j - i; // 计算当前段的长度

if (sz == target &&

now <= sz \* k) { // 检查当前段是否是最长时间段且满足平均失败率条件

res.add(new int[] {i, j}); // 将当前段的下标对添加到结果列表中

}

}

}

// 按起始下标进行排序

res.sort(Comparator.comparingInt(o -> o[0]));

// 构造输出字符串列表

List<String> b = new ArrayList<>();

for (int[] range : res) {

b.add(range[0] + "-" + range[1]);

}

// 打印输出字符串

System.out.println(String.join(" ", b));

}

}



k = int(input()) # 读取第一个输入值，表示平均失败率容忍值minAverageLost

a = list(map(int, input().split())) # 读取第二行输入，将其分割成整数数组

target = 0 # 初始化target为0，用于记录找到的最长时间段的长度

n = len(a) # n表示数组的长度

res = [] # 初始化结果列表

# 第一个双层循环用来找到符合条件的最长时间段的长度

for i in range(n):

now = 0 # 初始化当前段的失败率和

for j in range(i, n):

now += a[j] # 累加当前段的失败率

sz = 1 + j - i # 计算当前段的长度

if now <= sz \* k: # 检查当前段的平均失败率是否满足条件

target = max(target, sz) # 更新最长时间段的长度

# 如果没有找到符合条件的时间段，输出NULL并退出程序

if target <= 0:

print("NULL")

exit(0)

# 第二个双层循环用来收集所有符合最长时间段条件的下标对

for i in range(n):

now = 0 # 初始化当前段的失败率和

for j in range(i, n):

now += a[j] # 累加当前段的失败率

sz = 1 + j - i # 计算当前段的长度

if sz == target and now <= sz \* k: # 检查当前段是否是最长时间段且满足平均失败率条件

res.append((i, j)) # 将当前段的下标对添加到结果列表中

# 按起始下标进行排序

res.sort(key=lambda x: x[0])

# 构造输出字符串列表

b = []

for x, y in res:

b.append(f"{x}-{y}")

# 打印输出字符串

print(" ".join(b))



#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

using namespace std;

int main() {

int k;

cin >> k; // 读取第一个输入值，表示平均失败率容忍值minAverageLost

cin.ignore();

string line;

getline(cin, line); // 读取第二行输入

vector<int> a; // 整数数组

// 将输入转换为整数数组

size\_t pos = 0;

while ((pos = line.find(" ")) != string::npos) {

a.push\_back(stoi(line.substr(0, pos)));

line.erase(0, pos + 1);

}

a.push\_back(stoi(line));

int target =

0; // 初始化target为0，用于记录找到的最长时间段的长度

int n = a.size(); // n表示数组的长度

vector<pair<int, int>> res; // 初始化结果列表

// 第一个双层循环用来找到符合条件的最长时间段的长度

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

int now = 0; // 初始化当前段的失败率和

for (int j = i; j < n; ++j) {

now += a[j]; // 累加当前段的失败率

int sz = 1 + j - i; // 计算当前段的长度

if (now <= sz \* k) { // 检查当前段的平均失败率是否满足条件

target = max(target, sz); // 更新最长时间段的长度

}

}

}

// 如果没有找到符合条件的时间段，输出NULL并退出程序

if (target <= 0) {

cout << "NULL" << endl;

return 0;

}

// 第二个双层循环用来收集所有符合最长时间段条件的下标对

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

int now = 0; // 初始化当前段的失败率和

for (int j = i; j < n; ++j) {

now += a[j]; // 累加当前段的失败率

int sz = 1 + j - i; // 计算当前段的长度

if (sz == target &&

now <= sz \*

k) { // 检查当前段是否是最长时间段且满足平均失败率条件

res.emplace\_back(i, j); // 将当前段的下标对添加到结果列表中

}

}

}

// 按起始下标进行排序

sort(res.begin(), res.end(), [](const pair<int, int>& a,

const pair<int, int>& b) {

return a.first < b.first;

});

// 构造输出字符串列表

vector<string> b;

for (const auto& p : res) {

b.push\_back(to\_string(p.first) + "-" + to\_string(p.second));

}

// 打印输出字符串

cout << b[0];

for (size\_t i = 1; i < b.size(); ++i) {

cout << " " << b[i];

}

cout << endl;

return 0;

}



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// 定义一对整数的结构体

typedef struct {

int first;

int second;

} Pair;

// 比较函数，用于qsort排序

int compare(const void\* a, const void\* b) {

return ((Pair\*)a)->first - ((Pair\*)b)->first;

}

int main() {

int k;

scanf("%d",

&k); // 读取第一个输入值，表示平均失败率容忍值minAverageLost

getchar(); // 处理换行符

char line[1024];

fgets(line, sizeof(line), stdin); // 读取第二行输入

int a[100]; // 整数数组

char\* token = strtok(line, " ");

int n = 0; // n表示数组的长度

// 将输入转换为整数数组

while (token != NULL) {

a[n++] = atoi(token);

token = strtok(NULL, " ");

}

int target =

0; // 初始化target为0，用于记录找到的最长时间段的长度

Pair res[100]; // 初始化结果数组

int res\_size = 0; // 结果数组的大小

// 第一个双层循环用来找到符合条件的最长时间段的长度

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

int now = 0; // 初始化当前段的失败率和

for (int j = i; j < n; ++j) {

now += a[j]; // 累加当前段的失败率

int sz = 1 + j - i; // 计算当前段的长度

if (now <= sz \* k) { // 检查当前段的平均失败率是否满足条件

if (sz > target) {

target = sz; // 更新最长时间段的长度

}

}

}

}

// 如果没有找到符合条件的时间段，输出NULL并退出程序

if (target <= 0) {

printf("NULL\n");

return 0;

}

// 第二个双层循环用来收集所有符合最长时间段条件的下标对

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

int now = 0; // 初始化当前段的失败率和

for (int j = i; j < n; ++j) {

now += a[j]; // 累加当前段的失败率

int sz = 1 + j - i; // 计算当前段的长度

if (sz == target &&

now <= sz \* k) { // 检查当前段是否是最长时间段且满足平均失败率条件

res[res\_size].first = i;

res[res\_size].second = j;

res\_size++; // 将当前段的下标对添加到结果数组中

}

}

}

// 按起始下标进行排序

qsort(res, res\_size, sizeof(Pair), compare);

// 构造输出字符串列表

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

if (i > 0) {

printf(" ");

}

printf("%d-%d", res[i].first, res[i].second);

}

printf("\n");

return 0;

}



const readline = require("readline");

// 创建接口来读取标准输入

const rl = readline.createInterface({

input: process.stdin,

output: process.stdout,

});

let input = [];

rl.on("line", (line) => {

input.push(line); // 读取输入并存储到数组中

if (input.length === 2) {

rl.close();

}

});

rl.on("close", () => {

const k = parseInt(input[0]); // 读取第一个输入值，表示平均失败率容忍值minAverageLost

const a = input[1].split(" ").map(Number); // 读取第二行输入，并分割成整数数组

const n = a.length; // 数组的长度

let target = 0; // 初始化target为0，用于记录找到的最长时间段的长度

let res = []; // 初始化结果列表

// 第一个双层循环用来找到符合条件的最长时间段的长度

for (let i = 0; i < n; i++) {

let now = 0; // 初始化当前段的失败率和

for (let j = i; j < n; j++) {

now += a[j]; // 累加当前段的失败率

const sz = 1 + j - i; // 计算当前段的长度

if (now <= sz \* k) {

// 检查当前段的平均失败率是否满足条件

target = Math.max(target, sz); // 更新最长时间段的长度

}

}

}

// 如果没有找到符合条件的时间段，输出NULL并退出程序

if (target <= 0) {

console.log("NULL");

return;

}

// 第二个双层循环用来收集所有符合最长时间段条件的下标对

for (let i = 0; i < n; i++) {

let now = 0; // 初始化当前段的失败率和

for (let j = i; j < n; j++) {

now += a[j]; // 累加当前段的失败率

const sz = 1 + j - i; // 计算当前段的长度

if (sz === target && now <= sz \* k) {

// 检查当前段是否是最长时间段且满足平均失败率条件

res.push([i, j]); // 将当前段的下标对添加到结果列表中

}

}

}

// 按起始下标进行排序

res.sort((a, b) => a[0] - b[0]);

// 构造输出字符串列表

const b = res.map((pair) => `${pair[0]}-${pair[1]}`);

// 输出拼接后的字符串

console.log(b.join(" "));

});



package main

import (

"fmt"

"sort"

"strings"

)

// 定义一对整数的结构体

type Pair struct {

First int

Second int

}

func main() {

var arr,a []int

var k, num, n int

for {

if \_, err := fmt.Scanf("%d", &num); err == nil {

arr = append(arr, num)

} else {

break // 如果没有更多输入，则退出循环

}

}

k = arr[len(arr) - 1]

a = arr[1 : len(arr)]

n = len(a)

target := 0 // 初始化target为0，用于记录找到的最长时间段的长度

res := []Pair{} // 初始化结果数组

// 第一个双层循环用来找到符合条件的最长时间段的长度

for i := 0; i < n; i++ {

now := 0 // 初始化当前段的失败率和

for j := i; j < n; j++ {

now += a[j] // 累加当前段的失败率

sz := 1 + j - i // 计算当前段的长度

//fmt.Println(i,j,now)

if now <= sz\*k { // 检查当前段的平均失败率是否满足条件

if sz > target {

target = sz // 更新最长时间段的长度

}

}

}

}

// 如果没有找到符合条件的时间段，输出NULL并退出程序

if target <= 0 {

fmt.Println("NULL")

return

}

// 第二个双层循环用来收集所有符合最长时间段条件的下标对

for i := 0; i < n; i++ {

now := 0 // 初始化当前段的失败率和

for j := i; j < n; j++ {

now += a[j] // 累加当前段的失败率

sz := 1 + j - i // 计算当前段的长度

if sz == target && now <= sz\*k { // 检查当前段是否是最长时间段且满足平均失败率条件

res = append(res, Pair{i, j}) // 将当前段的下标对添加到结果数组中

}

}

}

// 按起始下标进行排序

sort.Slice(res, func(i, j int) bool {

return res[i].First < res[j].First

})

// 构造输出字符串列表

b := []string{}

for \_, p := range res {

b = append(b, fmt.Sprintf("%d-%d", p.First, p.Second))

}

// 输出拼接后的字符串

fmt.Println(strings.Join(b, " "))

}