https://blog.csdn.net/weixin\_44052055/article/details/123930856?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522165945219216781818750355%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request\_id=165945219216781818750355&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~first\_rank\_ecpm\_v1~pc\_rank\_34-9-123930856-null-null.142^v39^pc\_rank\_34\_ecpm25&utm\_term=%E5%8D%8E%E4%B8%BA%E6%9C%BA%E8%AF%95&spm=1018.2226.3001.4187

# 1. [字符串](https://so.csdn.net/so/search?q=%E5%AD%97%E7%AC%A6%E4%B8%B2&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_44052055/article/details/_blank)



## TLV解析Ⅰ

题目描述：

TLV 编码是按 [ Tag Length Value ] 格式进行编码的，一段码流中的信元用Tag标识， Tag在码流中 唯一不重复 ，Length表示信元Value的长度，Value表示信元的值。

码流以某信元的Tag开头，Tag固定占 一个字节，Length固定占 两个字节，字节序为 小端序 。

现给定TLV格式编码的码流，以及需要解码的信元Tag，请输出该信元的Value。

输入码流的16进制字符中，不包括小写字母，且要求输出的16进制字符串中也不要包含小写字母；

码流字符串的最大长度不超过50000个字节。

输入描述：

输入的第一行为一个字符串，表示待解码信元的 Tag ；

输入的第二行为一个字符串，表示待解码的 16进制码流 ，字节之间用 空格分隔 。

输出描述:

输出一个字符串，表示待解码信元以16进制表示的 Value 。

示例 1 ：

输入

31

32 01 00 AE 90 02 00 01 02 30 03 00 AB 32 31 31 02 00 32 33 33 01 00 CC

1

2

输出

32 33

1

说明

需要解析的信元的Tag是31，从码流的起始处开始匹配，

Tag为32的信元长度为1（01 00，小端序表示为1）；

第二个信元的Tag是90，其长度为2；

第三个信元的Tag是30，其长度为3；

第四个信元的Tag是31，其长度为2（02 00），所以返回长度后面的两个字节即可，即32 33。

思路分析

字符串转整数时，可以直接转为对应进制的整数。注意是小端，小的在后面。

参考代码：

Java代码实现：

import java.util.Scanner;

public class TLVAnalyize1 {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String tag = in.nextLine();

String[] tlv = in.nextLine().split(" ");

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

int length = Integer.parseInt(tlv[i + 2] + tlv[i + 1], 16); // 将字符串的Length转为16进制，小端，需要反过来

if (tag.equals(tlv[i])) {

StringBuilder sb = new StringBuilder();

for (int j = i + 3; j < i + 3 + length; j++) {

sb.append(tlv[j]).append(" ");

}

System.out.println(sb.toString());

break;

} else {

i += length + 3;

}

}

}

}

## VLAN资源池

题目描述：

VLAN是一种对局域网设备进行逻辑划分的技术，为了标识不同的VLAN，引入VLAN ID(1-4094之间的整数)的概念。

定义一个VLAN ID的资源池(下称VLAN资源池)，资源池中连续的VLAN用开始VLAN-结束VLAN表示，不连续的用单个整数表示，所有的VLAN用英文逗号连接起来。

现在有一个VLAN资源池，业务需要从资源池中申请一个VLAN，需要你输出从VLAN资源池中移除申请的VLAN后的资源池。

输入描述:

第一行为字符串格式的VLAN资源池，第二行为业务要申请的VLAN，VLAN的取值范围为[1,4094]之间的整数。

输出描述:

从输入VLAN资源池中移除申请的VLAN后字符串格式的VLAN资源池，输出要求满足题目描述中的格式，并且按照VLAN从小到大升序输出。

如果申请的VLAN不在原VLAN资源池内，输出原VLAN资源池升序排序后的字符串即可。

示例 1：

输入

1-5

2

1

2

输出

1,3-5

1

说明

原VLAN资源池中有VLAN 1、2、3、4、5，从资源池中移除2后，剩下VLAN 1、3、4、5，按照题目描述格式并升序后的结果为1,3-5。

示例 2：

输入

20-21,15,18,30,5-10

15

1

2

输出

5-10,18,20-21,30

1

说明

原VLAN资源池中有VLAN 5、6、7、8、9、10、15、18、20、21、30，从资源池中移除15后，资源池中剩下的VLAN为 5、6、7、8、9、10、18、20、21、30，按照题目描述格式并升序后的结果为5-10,18,20-21,30。

示例 3：

输入

5,1-3

10

1

2

输出

1-3,5

1

说明

原VLAN资源池中有VLAN 1、2、3，5，申请的VLAN 10不在原资源池中，将原资源池按照题目描述格式并按升序排序后输出的结果为1-3,5。

备注：

输入VLAN资源池中VLAN的数量取值范围为[2-4094]间的整数，资源池中VLAN不重复且合法([1,4094]之间的整数)，输入是乱序的。

参考代码：

Java代码实现：

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.Scanner;

public class VLANResource {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] VLANPool = in.nextLine().split(",");

int VLANNeed = in.nextInt();

// 将字符串中所有包含的数字加入list

List<Integer> list = new ArrayList<>(); // 存储数字数组

for (String vlan : VLANPool) {

String[] tmp = vlan.split("-");

if (tmp.length > 1) {

for (int i = Integer.parseInt(tmp[0]); i <= Integer.parseInt(tmp[1]); i++) {

list.add(i);

}

} else {

list.add(Integer.parseInt(tmp[0]));

}

}

// 如果list包含申请的VLAN，则从list删除

for (int i = 0; i < list.size(); i++) {

if (list.get(i) == VLANNeed) { // list是按照索引删除的，先找到对应索引，字符串的话用equals判断相等

list.remove(i);

}

}

// 将数组排完序后输出

Collections.sort(list);

StringBuilder sb = new StringBuilder();

// 对排序后的数组进行重组，一一遍历，如果下个数等于上个数加1，继续遍历，否则直接加入sb

int i = 0;

while (i < list.size()) {

int first = list.get(i);

int j = 1;

while (j <= list.size() - 1 - i) {

if (list.get(i) + j == list.get(i + j)) {

j++;

} else {

break;

}

}

if (j == 1) { // 如果下个数不等于上个数加1，直接加入sb,同时i++

sb.append(first);

sb.append(",");

i++;

} else { // 否则第一个数-递增的最后一个数加入sb,同时i=i+j

sb.append(String.valueOf(first) + "-" + String.valueOf(first + j - 1) + ",");

i = i + j;

}

}

System.out.println(sb.toString().substring(0, sb.length() - 1)); // 去除最后一个","

}

}

## 字符串统计（全量和占用字符集）

题目描述：

给定两个字符集合， 一个是全量字符集， 一个是已占用字符集， 已占用字符集中的字符不能再使用， 要求输出剩余可用字符集。

输入描述:

输入一个字符串 一定包含@，@前为全量字符集 @后的为已占用字符集

已占用字符集中的字符，一定是全量字符集中的字符，字符集中的字符跟字符之间使用英文逗号隔开

每个字符都表示为字符+数字的形式，用英文冒号分隔，比如a:1标识一个a字符

字符只考虑英文字母，区分大小写，数字只考虑正整型 不超过100

如果一个字符都没被占用，@标识仍存在

输出描述:

输出可用字符集，

不同的输出字符集之间用回车换行，

注意：输出的字符顺序要跟输入的一致，不能输出b:3,a:2,c:2

如果某个字符已全部占用，则不需要再输出

示例 1：

输入

a:3,b:5,c:2@a:1,b:2

1

输出

a:2,b:3,c:2

1

说明：

全量字符集为三个a，5个b，2个c

已占用字符集为1个a，2个b

由于已占用字符不能再使用

因此剩余可用字符为2个a，3个b，2个c

示例 2：

输入

a:3,b:5,c:2@

1

输出

a:3,b:5,c:2@

1

参考代码：

Java代码实现：

import java.util.\*;

import java.util.Scanner;

public class stringStatistics {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] str = in.nextLine().split("@");

String[] allStr = str[0].split(","); // 全量字符集

HashMap<Character, Integer> map = new HashMap<>();

ArrayList<Character> list = new ArrayList<>();

// 统计全量字符集中各个字符的数量

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

String[] aStr = allStr[i].split(":");

char ch1 = aStr[0].charAt(0); // astr[0]是字符

int n1 = Integer.parseInt(aStr[1]); // astr[1]是数字

map.put(ch1, n1);

list.add(ch1);

}

if (str.length > 1) { // 说明有占用字符集

String[] yongStr = str[1].split(",");

// 在全量字符集上减去对应的占用字符集

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

String[] yStr = yongStr[i].split(":");

char ch2 = yStr[0].charAt(0);

int n2 = Integer.parseInt(yStr[1]);

map.put(ch2, map.get(ch2) - n2);

}

} else { // 说明无占用字符集a:3,b:5,c:2@

String res = str[0] + "@";

System.out.println(res);

return;

}

ArrayList<String> list2 = new ArrayList<>(); // 存储最后的String结果

for (int i = 0; i < list.size(); i++) {

char c = list.get(i);

String x = "";

if (map.get(c) > 0) {

x = c + ":" + map.get(c);

list2.add(x);

}

}

// 输出格式

if (list2.size() > 0) {

for (int i = 0; i < list2.size() - 1; i++) {

System.out.print(list2.get(i) + ",");

}

System.out.print(list2.get(list2.size() - 1));

}

}

}

## 无重复字符的元素长度乘积的最大值

题目描述：

给定一个元素类型为小写字符串的数组，请计算两个没有相同字符的元素长度乘积的最大值。

如果没有符合条件的两个元素返回0。

输入描述：

输入为一个半角逗号分割的小写字符串数组

2<= 数组长度 <=100

0< 字符串长度 <=50

输出描述

两个没有相同字符的元素长度乘积的最大值

示例 1：

输入

iwdvpbn,hk,iuop,iikd,kadgpf

1

输出

14

1

说明：

数组中有5个元素。

iwdvpbn与hk无相同的字符，满足条件，iwdvpbn的长度为7，hk的长度为2，乘积为14（7\*2）。

iwdvpbn与iuop、iikd、kadgpf均有相同的字符，不满足条件。

iuop与iikd、kadgpf均有相同的字符，不满足条件。

参考代码：

Java代码实现：

import java.util.Scanner;

public class LongestSubstring {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] words = in.nextLine().split(",");

int maxLength = 0;

for (int i = 0; i < words.length - 1; i++) {

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

char[] word1 = words[i].toCharArray();

char[] word2 = words[j].toCharArray();

boolean flag = true;

for (int m = 0; m < word1.length; m++) {

for (int n = 0; n < word2.length; n++) {

if (word1[m] == word2[n]) {

flag = false;

break;

}

}

}

if (flag) { // 两个字符串不相等

int Length = word1.length \* word2.length;

maxLength = Math.max(maxLength, Length);

}

}

}

System.out.println(maxLength);

}

}

## 非严格递增连续数字序列

题目描述：

输入一个字符串仅包含大小写字母和数字，求字符串中包含的最长的非严格递增连续数字序列的长度（比如12234属于非严格递增连续数字序列）。

输入描述:

输入一个字符串仅包含大小写字母和数字，输入的字符串最大不超过255个字符。

输出描述：

最长的非严格递增连续数字序列的长度

示例 1：

输入

abc2234019A334bc

1

输出

4

1

说明：

2234为最长的非严格递增连续数字序列，所以长度为4

测试用例：

aaaaaa44ko543j123j7345677781 --> 34567778

aaaaa34567778a44ko543j123j71 --> 34567778

345678a44ko543j123j7134567778aa --> 134567778

1

2

3

参考代码：

Java代码实现：

import java.util.Scanner;

public class increaseNumSubsequences {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String str = in.nextLine();

int maxLength = 0;

int length = 0;

char maxChar = '0'; // 存储上一个数字字符

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

char c = str.charAt(i);

if (c >= maxChar && c <= '9') { // 判断是否是非递减

length++;

maxChar = c;

maxLength = Math.max(length, maxLength);

} else if (c >= '0' && c <= '9') { // 是数字，但小于上一个数字

length = 1;

maxChar = c;

} else { // 如果不是数字

length = 0;

maxChar = '0';

}

}

System.out.println(maxLength);

}

}

【编程题目 | 100分】拼接URL [ 100 / 简单 ]

## 拼接URL

题目描述：

给定一个url前缀和url后缀,通过,分割 需要将其连接为一个完整的url

如果前缀结尾和后缀开头都没有/，需要自动补上/连接符

如果前缀结尾和后缀开头都为/，需要自动去重

约束：不用考虑前后缀URL不合法情况

输入描述:

url前缀(一个长度小于100的字符串) url后缀(一个长度小于100的字符串)

输出描述：

拼接后的url

示例 1：

输入

/acm,/bb

1

输出

/acm/bb

1

示例 2：

输入

/abc/,/bcd

1

输出

/abc/bcd

1

示例 3：

输入

/acd,bef

1

输出

/acd/bef

1

示例 4：

输入

,

1

输出

/

1

参考代码：

Java代码实现：

Java接口实现

import java.util.Scanner;

public class URLPre {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] URL = in.nextLine().split(",");

StringBuilder sb = new StringBuilder();

if (URL.length == 0) {

System.out.println("/");

return;

}

sb.append("/");

sb.append(URL[0]);

sb.append("/");

sb.append(URL[1]);

System.out.println((sb.toString()).replaceAll("/+", "/"));

}

}

## 最远足迹

题目描述：

某探险队负责对地下洞穴进行探险。 探险队成员在进行探险任务时，随身携带的记录器会不定期地记录自身的坐标，但在记录的间隙中也会记录其他数据。 探索工作结束后，探险队需要获取到某成员在探险过程中相对于探险队总部的最远的足迹位置。

仪器记录坐标时，坐标的数据格式为(x,y)，如(1,2)、(100,200)，其中0<x<1000，0<y<1000。同时存在非法坐标，如(01,1)、(1,01)，(0,100)属于非法坐标。

设定探险队总部的坐标为(0,0)，某位置相对总部的距离为：x \* x+ y \* y。

若两个座标的相对总部的距离相同，则第一次到达的坐标为最远的足迹。

若记录仪中的坐标都不合法，输出总部坐标（0,0）。 备注：不需要考虑双层括号嵌套的情况，比如sfsdfsd((1,2))。

输入描述:

字符串，表示记录仪中的数据。

如：ferga13fdsf3(100,200)f2r3rfasf(300,400)

输出描述：

字符串，表示最远足迹到达的坐标。

如： (300,400)

示例 1：

输入

ferg(3,10)a13fdsf3(3,4)f2r3rfasf(5,10)

1

输出

(5,10)

1

说明

记录仪中的合法坐标有3个： (3,10)， (3,4)， (5,10)，其中(5,10)是相距总部最远的坐标， 输出(5,10)。

示例 2：

输入

asfefaweawfaw(0,1)fe

1

输出

(0,0)

1

说明

记录仪中的坐标都不合法，输出总部坐标(0,0)

参考代码：

Java代码实现：

import java.util.ArrayList;

import java.util.Scanner;

public class maxDistance {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String res = in.nextLine();

char[] str = res.toCharArray();

int max = 0;

String ans = "(0,0)"; // 默认值，非法时的结果

ArrayList<Integer> left = new ArrayList<>();

ArrayList<Integer> right = new ArrayList<>();

for (int i = 0; i < str.length; i++) { // 统计左右括号的位置，不考虑嵌套，所以两个list一一对应

if (str[i] == '(') {

left.add(i);

}

if (str[i] == ')') {

right.add(i);

}

}

for (int i = 0; i < left.size(); i++) {

String[] s = res.substring(left.get(i) + 1, right.get(i)).split(","); // 记录两个坐标

// 判断是否非法

if (s[0].charAt(0) != '0' && s[1].charAt(0) != '0') {

int num1 = Integer.parseInt(s[0]);

int num2 = Integer.parseInt(s[1]);

if (num1 < 1000 && num2 < 1000 && num1 \* num1 + num2 \* num2 > max) {

max = num1 \* num1 + num2 \* num2;

ans = "(" + s[0] + "," + s[1] + ")";

}

}

}

System.out.println(ans);

}

}

## 报文解压缩 [ 200 / 中等 ]

题目描述

为了提升数据传输的效率，会对传输的报文进行压缩处理。

输入一个压缩后的报文，请返回它解压后的原始报文。

压缩规则：n[str]，表示方括号内部的 str 正好重复 n 次。

注意 n 为正整数（0 < n <= 100），str只包含小写英文字母，不考虑异常情况。

输入描述

输入压缩后的报文：

1）不考虑无效的输入，报文没有额外的空格，方括号总是符合格式要求的；

2）原始报文不包含数字，所有的数字只表示重复的次数 n ，例如不会出现像 5b 或 3[8] 的输入；

输出描述

解压后的原始报文

注：

1）原始报文长度不会超过1000，不考虑异常的情况

示例

输入

3[m2[c]]

1

输出

mccmccmcc

1

说明

m2[c] 解压缩后为 mcc，重复三次为 mccmccmcc

输入

10[k]2[mn3[j2[op]]]

1

输出

kkkkkkkkkkmnjopopjopopjopopmnjopopjopopjopop

1

思路分析

这道题是字符串处理的问题，同时字符串中嵌套括号，根据嵌套的括号进行报文解压缩，所以我们很容易想到用栈去解决问题。

首先把右括号之前的字符入栈。

遇到右括号时，开始进行解压缩，当栈不空的情况下开始出栈

出栈字符为字母时，暂存

出栈字符为数字时，判断如果前一个字符存在，是否仍为数字（处理两位数字，这里没有对数字100进行判断，如果需要再加一个判断即可）

循环num-1次累加暂存的字符串，因为本身有一次

注意：题目要求的压缩规则：n[str]，str只包含小写英文字母。所以左括号左边一定是数字。

3[[m2[c]]2[a]]

1

则不符合压缩规则。

参考代码

import java.util.Scanner;

import java.util.Stack;

public class baoWenJieYaSuo {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String str = in.nextLine();

Stack<Character> stack = new Stack<>();

String res = "";

char[] ch = str.toCharArray();

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

if (ch[i] == ']') {

// 解压缩

String tmpStr = "";

while(!stack.isEmpty()) {

char poll = stack.pop();

if (poll >= 'a' && poll <= 'z') { // 如果出栈的为字母

tmpStr = String.valueOf(poll) + tmpStr;

} else if (Character.isDigit(poll)) { // 如果出栈的为数字

int num = 0;

if (!stack.isEmpty() && Character.isDigit(stack.peek())) {

num = (stack.pop() - '0') \* 10 + (poll - '0');

} else {

num = poll - '0';

}

String waitStr = tmpStr; // 需要将tempStr暂存起来

for (int j = 0; j < num - 1; j++) { // 这里做 num -1 次的字符串相

tmpStr += waitStr;

}

}

}

res += tmpStr;

}

stack.push(ch[i]);

}

System.out.println(res);

}

}

————————————————

版权声明：本文为CSDN博主「小朱小朱绝不服输」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/weixin_44052055/article/details/125821075>



# 2. 栈和队列

## 滑动窗口最大值

本题可使用本地IDE编码，不能使用本地已有代码。无跳出限制，编码后请点击"保存并提交"按钮进行代码提交。

题目描述：

有一个N个整数的数组，和一个长度为M的窗口，窗口从数组内的第一个数开始滑动直到窗口不能滑动为止， 每次窗口滑动产生一个窗口和（窗口内所有数的和），求窗口滑动产生的所有窗口和的最大值。

输入描述：

第一行输入一个正整数N，表示整数个数。（0<N<100000）

第二行输入N个整数，整数的取值范围为[-100,100]。

第三行输入一个正整数M，M代表窗口的大小，M<=100000，且M<=N。

输出描述：

窗口滑动产生所有窗口和的最大值。

示例 1 输入输出示例仅供调试，后台判题数据一般不包含示例

输入

6

12 10 20 30 15 23

3

1

2

3

输出

68

1

思路分析：

这与leetcode的滑动窗口最大值不同，那个需要用单调栈来实现。计算每个窗口的最大值。

这道题可以参考单调栈实现方法，用来统计滑动窗口的最大值。也可以使用双指针来实现。

参考代码：

Java实现：

1. 双端队列

import java.util.LinkedList;

import java.util.Scanner;

public class maxSlidingWindow {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

int n = in.nextInt();

int[] nums = new int[n];

in.nextLine();

String[] s = in.nextLine().split(" ");

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

nums[i] = Integer.parseInt(s[i]);

}

int k = in.nextInt();

int res = 0;

// 双端队列实现

int ans = 0;

LinkedList<Integer> queue = new LinkedList<>();

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

// 添加当前值对应的数组下标，计算当前窗口和

queue.add(i);

ans += nums[i];

// 初始化窗口，等到窗口长度为k时，下次移动时删除过期数值

if (queue.getLast() >= k) {

ans -= nums[queue.getFirst()];

queue.removeFirst();

}

// 窗口长度为k时，后更新所有窗口的最大值

if (i - k + 1 >= 0) {

res = Math.max(ans, res);

}

}

System.out.println(res);

}

}

2. 双指针

import java.util.Scanner;

public class maxSlidingWindow {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

int n = in.nextInt();

int[] nums = new int[n];

in.nextLine();

String[] s = in.nextLine().split(" ");

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

nums[i] = Integer.parseInt(s[i]);

}

int k = in.nextInt();

int res = 0;

// 双指针

int left = 0, sum = 0;

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

sum += nums[right];

while (left <= right && right - left + 1 >= k) {

res = Math.max(res, sum);

sum -= nums[left++];

}

}

System.out.println(res);

}

}

## 最大嵌套括号深度

题目描述：

现有一字符串仅由 ‘(’， ‘)’， ‘{’， ‘}’， ‘[’， ']'六种括号组成。 若字符串满足以下条件之一，则为无效字符串：

①任一类型的左右括号数量不相等；

②存在未按正确顺序（先左后右）闭合的括号。

输出括号的最大嵌套深度，若字符串无效则输出 0。 0≤字符串长度≤100000

输入描述:

一个只包括 ‘(’， ‘)’， ‘{’， ‘}’， ‘[’， ']'的字符串

输出描述：

一个整数，最大的括号深度

示例 1：

输入

[]

1

输出

1

1

说明

有效字符串，最大嵌套深度为1

示例 2：

输入

([]{()})

1

输出

3

1

说明

有效字符串，最大嵌套深度为3

示例 3：

输入

(]

1

输出

0

1

说明

无效字符串，有两种类型的左右括号数量不相等

示例 4：

输入

([)]

1

输出

0

1

说明

无效字符串，存在未按正确顺序闭合的括号

参考代码：

Java代码实现：

import java.util.Scanner;

import java.util.Stack;

public class kuohaoDepth {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String s = in.nextLine();

if (s.equals("")) { // 字符串为空

System.out.println(0);

return ;

}

Stack<Character> stack = new Stack<>();

int i = 0;

int max = 0;

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

char c = s.charAt(i);

if (c == '(' || c == '{' || c == '[') {

stack.push(c);

max = Math.max(max, stack.size());

} else { // 如果是右括号

if (stack.size() == 0) { // 第一个是右括号，直接break

break;

}

if (c == ')') { // 判断括号是否匹配，匹配，则continue，不匹配直接break(这里放到最后整体break)

if (stack.pop() == '(') {

continue;

}

} else if (c == ']') {

if (stack.pop() == '[') {

continue;

}

} else {

if (stack.pop() == '{') {

continue;

}

}

break;

}

}

if (i == s.length() && stack.size() == 0) {

System.out.println(max);

} else {

System.out.println(0);

}

}

}

## 字符串消除

题目描述：

游戏规则： 输入一个只包含英文字母的字符串, 字符串中的两个字母如果相邻且相同,就可以消除。 在字符串上反复执行消除的动作, 直到无法继续消除为止,此时游戏结束。 输出最终得到的字符串长度.

输入描述:

输入原始字符串str 只能包含大小写英文字母,字母的大小写敏感, str长度不超过100

输出描述：

输出游戏结束后,最终得到的字符串长度

示例 1：

输入

gg

1

输出

0

说明

gg可以直接消除 得到空串 长度为0

示例 2：

输入

mMbccbc

1

输出

3

说明

mMbccbc中 可以先消除cc 此时变为mMbbc 再消除 bb 此时变成mMc 此时没有相同且相邻的字符 无法继续消除 最终得到字符串mMc 长度为3

备注：

输入中包含非大小写英文字母时 均为异常输入 直接返回0

参考代码：

Java代码实现：

import java.util.Scanner;

import java.util.Stack;

public class strRemove {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String s = in.nextLine();

Stack<Character> stack = new Stack<>();

// 先判断字符串是否有不是大小写字母的字符

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

char ch = s.charAt(i);

if ((ch < 'a' || ch > 'z') && (ch < 'A' || ch >'Z')) {

System.out.println(0);

return;

}

if (stack.isEmpty() || stack.peek() != ch) {

stack.push(ch);

} else {

stack.pop();

}

}

System.out.println(stack.size());

}

}

## DNA序列

描述

一个 DNA 序列由 A/C/G/T 四个字母的排列组合组成。 G 和 C 的比例（定义为 GC-Ratio ）是序列中 G 和 C 两个字母的总的出现次数除以总的字母数目（也就是序列长度）。在基因工程中，这个比例非常重要。因为高的 GC-Ratio 可能是基因的起始点。

给定一个很长的 DNA 序列，以及限定的子串长度 N ，请帮助研究人员在给出的 DNA 序列中从左往右找出 GC-Ratio 最高且长度为 N 的第一个子串。

DNA序列为 ACGT 的子串有: ACG , CG , CGT 等等，但是没有 AGT ， CT 等等

数据范围：字符串长度满足 1≤n≤1000，输入的字符串只包含 A/C/G/T 字母

输入描述：

输入一个string型基因序列，和int型子串的长度

输出描述：

找出GC比例最高的子串,如果有多个则输出第一个的子串

示例1：

输入

ACGT

2

1

2

输出

CG

1

说明

ACGT长度为2的子串有AC,CG,GT3个，其中AC和GT2个的GC-Ratio都为0.5，CG为1，故输出CG

示例2：

输入

AACTGTGCACGACCTGA

5

1

2

输出

GCACG

1

说明

虽然CGACC的GC-Ratio也是最高，但它是从左往右找到的GC-Ratio最高的第2个子串，所以只能输出GCACG。

思路分析：

题目中主要信息：

输入的字符串中只有ACGT四种字符

限定长度为nnn的子串，求其中CG比例最高的第一个子串

解读： 长度限定的情况下，要找比例越高即找出现次数越多

可以有两种方法：

暴力搜索：遍历字符串每个位置作为起始，然后遍历以这个字符作为起始的长为n的子串，分别统计子串中CG的数量，与之前记录的最大值比较，然后更新记录下最大值及最大CG含量子串的起始位置。

最后根据最终的起始位置和长度n利用substr函数输出，这样由左到右地找出来的就一定是第一个。

时间复杂度：O(mn)，其中m为字符串的长度，n为限定的子串长度，需要遍历字符串每个位置为起点的子串

这里就不写了。

滑动窗口：首先用一个长度为n的窗口覆盖字符串前n部分子串，统计这里的CG数量，并暂时作为最大值。然后窗口右移，如果左边出去的是CG那么窗口内的CG数量减少一个，如果右边进来的是CG那么窗口内的CG数量增加一个，每次滑动都统计窗口内的CG数量，与临时最大值比较，记录下最大窗口的起始下标。

窗口右端抵达字符串末尾时结束，根据下标用substr函数输出字符串含CG最高的子串。

时间复杂度：O(m)，其中m为字符串的长度，窗口滑动相当于遍历字符串

参考代码：

Java代码实现滑动窗口方法：

import java.util.\*;

public class Main {

public static void main (String[] args) {

Scanner in = new Scanner(System.in);

while (in.hasNext()) {

String str = in.nextLine();

int n = in.nextInt();

char[] ch = str.toCharArray();

int start = 0, count = 0, max = 0;

for (int i = 0; i < n; i++) { // 最开始的窗口

if (ch[i] == 'C' || ch[i] == 'G') {

count++;

}

}

max = count; // 最开始窗口的GC数量

int left = 1, right = n; // 从录入窗口的左右点右移一位开始

while (right < ch.length) {

if (ch[left - 1] == 'C' || ch[left - 1] == 'G') { // 左边出去的是CG

count--;

}

if (ch[right] == 'C' || ch[right] == 'G') { // 右边进来的是CG

count++;

}

if (count > max) {

max = count;

start = left;

}

left++;

right++;

}

System.out.println(str.substring(start, start + n));

}

}

}



# 3. [数组](https://so.csdn.net/so/search?q=%E6%95%B0%E7%BB%84&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_44052055/article/details/_blank)

## 猴子吃桃

题目描述：

孙悟空喜欢吃蟠桃，一天他乘守卫蟠桃园的天兵天将离开了而偷偷的来到王母娘娘的蟠桃园偷吃蟠桃。

已知蟠桃园有 N 棵蟠桃树，第 i 棵蟠桃树上有 N[i]（大于 0）个蟠桃，天兵天将将在 H（不小于蟠桃树棵数）小时后回来。

孙悟空可以决定他吃蟠桃的速度 K（单位：个/小时），每个小时他会选择一颗蟠桃树，从中吃掉 K 个蟠桃，如果这棵树上的蟠桃数小于 K，他将吃掉这棵树上所有蟠桃，然后这一小时内不再吃其余蟠桃树上的蟠桃。

孙悟空喜欢慢慢吃，但仍想在天兵天将回来前将所有蟠桃吃完。

求孙悟空可以在 H 小时内吃掉所有蟠桃的最小速度 K（K 为整数）。

输入描述:

从标准输入中读取一行数字，前面数字表示每棵数上蟠桃个数，最后的数字表示天兵天将将离开的时间。

输出描述：

吃掉所有蟠桃的 最小速度 K（K 为整数）或 输入异常时输出 -1。

示例 1：

输入

3 11 6 7 8

1

输出

4

1

说明：

天兵天将8个小时后回来，孙悟空吃掉所有蟠桃的最小速度4。

第1小时全部吃完第一棵树，吃3个，

第2小时吃4个，第二棵树剩7个，

第3小时吃4个，第二棵树剩3个，

第4小时吃3个，第二棵树吃完，

第5小时吃4个，第三棵树剩2个，

第6小时吃2个，第三棵树吃完，

第7小时吃4个，第4棵树剩3个，

第8小时吃3个，第4棵树吃完。

思路分析

二分法的应用，找到满足条件的最小K。可以参考875. 爱吃香蕉的珂珂，原题。

参考代码：

Java代码实现：

import java.util.Arrays;

import java.util.Scanner;

public class MonkeyEatPeach {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] str = in.nextLine().split(" ");

int[] nums = new int[str.length - 1];

for (int i = 0; i < str.length - 1; i++) {

// 判断是否有异常输入,非数字

for (int j = 0; j < str[i].length(); j++) {

char ch = str[i].charAt(j);

if (ch >= '0' && ch <= '9') {

continue;

} else {

System.out.println(-1);

return ;

}

}

nums[i] = Integer.parseInt(str[i]);

}

int H = Integer.parseInt(str[str.length - 1]);

Arrays.sort(nums);

// 使用二分查找，找到最小的满足条件的K

int left = 1, right = nums[nums.length - 1];

while (left < right) {

int mid = left + (right - left) / 2;

if (!check(nums, H, mid)) {

left = mid + 1;

} else {

right = mid;

}

}

System.out.println(left);

}

private static boolean check(int[] nums, int H, int K) { // 判断选定的天数是否满足条件

int time = 0;

for (int num : nums) {

time += (num + K - 1) / K;

}

return time <= H;

}

}

43

另一种写法：

import java.util.Arrays;

import java.util.Scanner;

public class MonkeyEatPeach {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] str = in.nextLine().split(" ");

int[] nums = new int[str.length - 1];

for (int i = 0; i < str.length - 1; i++) {

// 判断是否有异常输入,非数字

for (int j = 0; j < str[i].length(); j++) {

char ch = str[i].charAt(j);

if (ch >= '0' && ch <= '9') {

continue;

} else {

System.out.println(-1);

return ;

}

}

nums[i] = Integer.parseInt(str[i]);

}

int H = Integer.parseInt(str[str.length - 1]);

Arrays.sort(nums);

int left = 1, right = nums[nums.length - 1];

while (left < right) {

int mid = (right - left) / 2 + left;

int time = 0;

for (int num : nums) {

time += (num + mid - 1) / mid;

}

if (time <= H) {

right = mid;

} else {

left = mid + 1;

}

}

System.out.println(left);

}

}

## k 对元素最小值

题目描述：

给定两个整数数组array1 array2，数组元素按升序排列假设从arr1 arr2中分别取出一个元素，可构成一对元素。

现在需要取出k对元素，并对取出的所有元素求和，计算和的最小值。

注意：两对元素对应arr1 arr2的下标是相同的视为同一对元素

输入描述:

输入两行数组arr1 arr2

每行首个数字为数组大小size 0<size<=100

arr1，2中的每个元素 0<arr[i]<1000

接下来一行 正整数k 0<k<=arr1.size \* arr2.size

输出描述：

满足要求的最小值

示例 1：

输入

3 1 1 2

3 1 2 3

2

1

2

3

输出

4

1

说明

用例中需要取两个元素，

取第一个数组第0个元素与第二个数组第0个元素组成一个元素[1,1]

取第一个数组第1个元素与第二个数组第0个元素组成一个元素[1,1]

求和为1+1+1+1=4 ，满足要求最小

参考代码

Java代码实现：

import java.util.PriorityQueue;

import java.util.Scanner;

public class minValueOfK {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] str1 = in.nextLine().split(" ");

int m = Integer.parseInt(str1[0]);

int[] nums1 = new int[m];

for (int i = 1; i < str1.length; i++) {

nums1[i - 1] = Integer.parseInt(str1[i]);

}

String[] str2 = in.nextLine().split(" ");

int n = Integer.parseInt(str2[0]);

int[] nums2 = new int[n];

for (int i = 1; i < str2.length; i++) {

nums2[i - 1] = Integer.parseInt(str2[i]);

}

int k = in.nextInt();

// 优先队列实现,小顶堆

int min = 0;

PriorityQueue<int[]> queue = new PriorityQueue<>((a, b) -> (nums1[a[0]] + nums2[a[1]]) - (nums1[b[0]] + nums2[b[1]]));

// 把nums1索引加入队列

for (int i = 0; i < Math.min(m, k); i++) {

queue.add(new int[]{i, 0});

}

while (k > 0 && !queue.isEmpty()) {

int[] idx = queue.poll();

min += (nums1[idx[0]] + nums2[idx[1]]);

// 为了避免重复，把nums2的索引增加

if (idx[1] + 1 < n) {

queue.add(new int[]{idx[0], idx[1] + 1});

}

k--;

}

System.out.println(min);

}

}

## 绘图机器

题目描述：

绘图机器的绘图笔初始位置在原点(0,0) 机器启动后按照以下规则来进行绘制直线

尝试沿着横线坐标正向绘制直线 直到给定的终点E

期间可以通过指令在纵坐标轴方向进行偏移 offsetY为正数表示正向偏移,为负数表示负向偏移

给定的横坐标终点值E 以及若干条绘制指令 请计算绘制的直线和横坐标轴以及x=E的直线组成的图形面积

输入描述:

首行为两个整数N 和 E 表示有N条指令, 机器运行的横坐标终点值E

接下来N行 每行两个整数表示一条绘制指令x offsetY

用例保证横坐标x以递增排序的方式出现 且不会出现相同横坐标x

取值范围: 0<N<=10000

0<=x<=E<=20000

-10000<=offsetY<=10000

输出描述：

一个整数表示计算得到的面积 用例保证结果范围在0到4294967295之内

示例 1：

输入

4 10

1 1

2 1

3 1

4 -2

1

2

3

4

5

输出

12

1

说明

通过操作机器最后绘制了如下图形（蓝色为绘制笔绘制的直线）

计算图中阴影部分面积，其值为

1\*1+2\*1+3\*1+1\*6=12

1

示例 2：

输入

2 4

0 1

2 -2

1

2

3

输出

4

1

思路分析：

简单的模拟题，模拟计算所围矩形的面积。

参考代码

Java代码实现：

import java.util.Scanner;

public class drawMachine {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] str = in.nextLine().split(" ");

int n = Integer.parseInt(str[0]);

int end = Integer.parseInt(str[1]);

int[] x = new int[n];

int[] offsetY = new int[n];

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

x[i] = in.nextInt();

offsetY[i] = in.nextInt();

}

int area = 0;

int a = 0, b = 0;

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

area += (x[i] - a) \* Math.abs(b);

a = x[i];

b += offsetY[i];

}

area += (end - a) \* Math.abs(b);

System.out.println(area);

}

## 最大社交距离

题目描述：

疫情期间需要大家保证一定的社交距离，公司组织开交流会议。

座位一排共 N 个座位，编号分别为 [0, N - 1] , 要求员工一个接着一个进入会议室，并且可以在任何时候离开会议室。

满足：

每当一个员工进入时，需要坐到最大社交距离（最大化自己和其他人的距离的座位）；

如果有多个这样的座位，则坐到 索引最小 的那个座位。

输入描述：

会议室座位总数 seatNum 。(1 <= seatNum <= 500)

员工的进出顺序 seatOrLeave 数组，元素值为 1，表示进场；元素值为负数，表示出场（特殊：位置 0 的员工不会离开）。

例如 - 4 表示坐在位置 4 的员工离开（保证有员工坐在该座位上）

输出描述：

最后进来员工，他会坐在第几个位置，如果位置已满，则输出 - 1 。

示例 1：

输入

10

[1,1,1,1,-4,1]

1

2

输出

5

1

说明

seat -> 0 , 空在任何位置都行，但是要给他安排索引最小的位置，也就是座位 0

seat -> 9 , 要和旁边的人距离最远，也就是座位 9

seat -> 4 , 要和旁边的人距离最远，应该坐到中间，也就是座位 4

seat -> 2 , 员工最后坐在 2 号座位上

leave [4] , 4 号座位的员工离开

seat -> 5 , 员工最后坐在 5 号座位上

解题思路

这道题坐的位置是满足最大距离下的最小索引，可以使用TreeSet有序去重的集合来存放已经坐过的位置，第一个进来的，只能坐到位置0，最后一个进来的只能坐到最后一个位置，中间进来的就需要找，离坐下的最远距离的最小索引位置，需要遍历已经坐下的位置，找到中间索引最小的最大距离。

参考代码

Java代码实现：

import java.util.\*;

public class maxSocialDistance {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

int seatNum = in.nextInt();

in.nextLine();

String seat = in.nextLine();

String[] c = seat.substring(1, seat.length() - 1).split(",");

int[] seatOrLeave = new int[c.length];

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

seatOrLeave[i] = Integer.parseInt(c[i]);

}

int ans = seatDistance(seatNum, seatOrLeave);

System.out.print(ans);

}

public static int seatDistance(int seatNum, int[] seatOrLeave) {

TreeSet<Integer> seatedNums = new TreeSet<>(); // 使用TreeSet有序集合记录被坐过的座位

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

int op = seatOrLeave[i];

if (op > 0) {

if (seatedNums.size() == 0) { // 如果是第一个坐

if (i == seatOrLeave.length - 1) { // 如果只有一个位置

return 0;

}

seatedNums.add(0);

} else if (seatedNums.size() == 1) { // 第二个人进来，坐在最右边

seatedNums.add(seatNum - 1);

if (i == seatOrLeave.length - 1) { // 如果只有两个位置

return seatNum - 1;

}

} else if (seatedNums.size() > 1 && seatedNums.size() < seatNum) { // 坐到中间的位置

int[] ints = new int[seatedNums.size()];

int count = 0;

for (Integer seatedNum : seatedNums) { // 将已经坐过的位置存入到数组中

ints[count++] = seatedNum;

}

int maxLen = 0;

int start = 0;

for (int j = 0; j < ints.length - 1; j++) { // 计算最远距离

int len = ints[j + 1] - ints[j];

if (len / 2 > maxLen) {

maxLen = len / 2;

start = ints[j];

}

}

seatedNums.add(start + maxLen); // 将对应的起始位置加上最远距离加入seatedNums

if (i == seatOrLeave.length - 1) {

return start + maxLen;

}

} else { // 位置坐满

return -1;

}

} else { // 如果是负数，则将该座位移出

seatedNums.remove(-op);

}

}

return 0;

}

}

## 玩牌高手

题目描述：

给定一个长度为n的整型数组，表示一个选手在n轮内可选择的牌面分数。选手基于规则选牌，请计算所有轮结束后其可以获得的最高总分数。

选择规则如下：

在每轮里选手可以选择获取该轮牌面，则其总分数加上该轮牌面分数，为其新的总分数。

选手也可不选择本轮牌面直接跳到下一轮，此时将当前总分数还原为3轮前的总分数，若当前轮次小于等于3（即在第1、2、3轮选择跳过轮次），则总分数置为0。

选手的初始总分数为0，且必须依次参加每一轮。

输入描述

第一行为一个小写逗号分割的字符串，表示n轮的牌面分数，1<= n <=20。

分数值为整数，-100 <= 分数值 <= 100。

不考虑格式问题。

输出描述

所有轮结束后选手获得的最高总分数。

示例 1：

输入

1,-5,-6,4,3,6,-2

1

输出

11

1

说明

总共有7轮牌面。

第一轮选择该轮牌面，总分数为1。

第二轮不选择该轮牌面，总分数还原为0。

第三轮不选择该轮牌面，总分数还原为0。

第四轮选择该轮牌面，总分数为4。

第五轮选择该轮牌面，总分数为7。

第六轮选择该轮牌面，总分数为13。

第七轮如果不选择该轮牌面，则总分数还原到3轮1前分数，即第四轮的总分数4，如果选择该轮牌面，总分数为11，所以选择该轮牌面。

因此，最终的最高总分为11。

思路分析

简单的模拟题。按规则走就行了。

参考代码

Java代码实现：

import java.util.\*;

public class playerOfTop {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] str = in.nextLine().split(",");

in.close();

int[] nums = new int[str.length];

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

nums[i] = Integer.parseInt(str[i]);

}

int[] score = new int[str.length];

int sum = 0;

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

if (i < 3) {

if (nums[i] <= 0) {

score[i] = 0;

} else {

score[i] = sum + nums[i];

}

} else {

if (nums[i] > 0) {

score[i] = sum + nums[i];

} else {

score[i] = Math.max(sum + nums[i], score[i - 3]);

}

}

sum = score[i];

}

System.out.println(score[str.length - 1]);

}

}

————————————————

版权声明：本文为CSDN博主「小朱小朱绝不服输」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

## 最大差

题目描述：

判断一组不等式是否满足约束并输出最大差

给定一组不等式，判断是否成立并输出不等式的最大差(输出浮点数的整数部分)， 要求：

1）不等式系数为double类型，是一个二维数组；

2）不等式的变量为int类型，是一维数组；

3）不等式的目标值为double类型，是一维数组；

4）不等式约束为字符串数组，只能是：“>”,“>=”,“<”,“<=”,“=”，例如,不等式组：

a11\*x1+a12\*x2+a13\*x3+a14\*x4+a15\*x5<=b1;

a21\*x1+a22\*x2+a23\*x3+a24\*x4+a25\*x5<=b2;

a31\*x1+a32\*x2+a33\*x3+a34\*x4+a35\*x5<=b3;

1

2

3

4

5

最大差=max

{ (a11\*x1+a12\*x2+a13\*x3+a14\*x4+a15\*x5-b1),

(a21\*x1+a22\*x2+a23\*x3+a24\*x4+a25\*x5-b2),

(a31\*x1+a32\*x2+a33\*x3+a34\*x4+a35\*x5-b3) }，

1

2

3

4

5

6

类型为整数(输出浮点数的整数部分)

输入描述

1）不等式组系数(double类型)：

a11,a12,a13,a14,a15 a21,a22,a23,a24,a25 a31,a32,a33,a34,a35

1

2）不等式变量(int类型)：

x1,x2,x3,x4,x5

1

3）不等式目标值(double类型)：

b1,b2,b3

1

4）不等式约束(字符串类型):

<=,<=,<=

1

输入：

a11,a12,a13,a14,a15;

a21,a22,a23,a24,a25;

a31,a32,a33,a34,a35;

x1,x2,x3,x4,x5;

b1,b2,b3;

<=,<=,<=

1

2

3

4

5

6

7

8

9

10

11

输出描述

true 或者 false, 最大差

示例 1：

输入

2.3,3,5.6,7,6;11,3,8.6,25,1;0.3,9,5.3,66,7.8;1,3,2,7,5;340,670,80.6;<=,<=,<=

1

输出

false 458

1

说明

// 2.3 × 1 + 3 × 3 + 5.6 × 2 + 7 × 7 + 6 × 5 - 340 = -238.5

// 11 × 1 + 3 × 3 + 8.6 × 2 + 25 × 7 + 5 × 1 - 670 = -452.8

// 0.3 × 1 + 9 × 3 + 5.3 × 2 + 66 × 7 + 7.8 × 5 - 80.6 = 458.3

1

2

3

前两个不满足条件，所以输出为false。

然后计算三个差值中的最大值，取整数部分。

示例 2：

输入

2.36,3,6,7.1,6;1,30,8.6,2.5,21;0.3,69,5.3,6.6,7.8;1,13,2,17,5;340,67,300.6;<=,>=,<=

1

输出

false 758

1

2

思路分析

题目臭长，纯模拟问题，这道题充分考察了一个字符串分割，提取，计算，拼接，判断。

参考代码

Java代码实现：

import java.util.Scanner;

public class maxCha {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] str = in.nextLine().split(";");

in.close();

int num\_eq = str[str.length - 1].split(",").length; // 3,等式的数量

int num\_x = str[0].split(",").length; // 5，系数和未知数的数量

double[][] a= new double[num\_eq][num\_x];

int[] x = new int[num\_x];

double[] b = new double[num\_eq];

String[] eq = new String[num\_eq];

int[] res = new int[num\_eq];

int max = 0;

boolean flag = true;

// 分别处理a,x,b,符号

for (int i = 0; i < num\_eq; i++) { // 处理a

String[] tmp1 = str[i].split(",");

for (int j = 0; j < tmp1.length; j++) {

a[i][j] = Double.valueOf(tmp1[j]);

}

}

String[] tmp2 = str[num\_eq].split(",");

for (int i = 0; i < tmp2.length; i++) { // 处理x

x[i] = Integer.parseInt(tmp2[i]);

}

String[] tmp3 = str[num\_eq + 1].split(",");

for (int i = 0; i < tmp3.length; i++) { // 处理b

b[i] = Double.valueOf(tmp3[i]);

}

String[] tmp4 = str[num\_eq + 2].split(",");

for (int i = 0; i < tmp4.length; i++) { // 处理符号

eq[i] = tmp4[i];

}

for (int i = 0; i < num\_eq; i++) {

double tmp = 0.0;

for (int j = 0; j < num\_x; j++) {

tmp += a[i][j] \* x[j];

}

if ("<=".equals(eq[i])) {

flag = tmp <= b[i] ? flag && true : flag && false;

} else if ("<".equals(eq[i])) {

flag = tmp < b[i] ? flag && true : flag && false;

}else if ("=".equals(eq[i])) {

flag = tmp == b[i] ? flag && true : flag && false;

}else if (">=".equals(eq[i])) {

flag = tmp >= b[i] ? flag && true : flag && false;

}else if (">".equals(eq[i])) {

flag = tmp > b[i] ? flag && true : flag && false;

}

res[i] =(int) ((tmp - b[i]) / 1);

}

for (int i = 0; i < num\_eq; i++) {

max = Math.max(max, res[i]);

}

System.out.println(flag + " " + max);

}

}

————————————————

版权声明：本文为CSDN博主「小朱小朱绝不服输」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/weixin_44052055/article/details/124064445>

题目描述

## 【区间交集】

给定一组闭区间，其中部分区间存在交集。

任意两个给定区间的交集，称为公共区间(如:[1,2],[2,3]的公共区间为[2,2]，[3,5],[3,6]的公共区间为[3,5])。

公共区间之间若存在交集，则需要合并(如:[1,3],[3,5]区间存在交集[3,3]，需合并为[1,5])。

按升序排列输出合并后的区间列表。

输入描述

一组区间列表，区间数为 N: 0<=N<=1000;区间元素为 X: -10000<=X<=10000。

输出描述

升序排列的合并区间列表

备注

1、区间元素均为数字，不考虑字母、符号等异常输入。

2、单个区间认定为无公共区间。

示例：

输入

1 3 2 4 4 8 5 9

1

输出

2 3 4 4 5 8

1

说明

[1,3]、[2,4]、[4,8]、[5,9] 四个区间

[1,3]与[2,4]交集为[2,3]，[1,3]与[4,8]、[5,9]没有交集

[2,4]与[4,8]]交集为[4,4]。[2,4]与[5,9]没有交集

[4,8]与[5,9]的交集为[5,8]

所以最终的输出为[2,3]、[4,4]、[5,8]

1

2

3

4

5

输入

1 6 2 5 5 7

1

输出

2 6

1

说明

[1,6]、[2,5]的交集为[2,5]，[1,6]、[5,7]的交集为[5,6]

[2,5]、[5,7]的交集为[5,5]

最后的输出为：2 6

1

2

3

输入

1 2 3 4

1

输出

None (这里没看到题目上具体要求输出什么，根据题目情况临场发挥即可)

1

注：这道题目的输入输出有多个版本，有一行的，有分行的，有带中括号列表的，我是按一行读取，只是输入输出的不同而已，题目解法是一样的。

思路分析

这道题目的要求简单的说就是当各个区间有交集的时候取交集，再求交集的并集。

求区间交集，双指针方法，可以参考leetcode：986. 区间列表的交集

求区间并集，升序然后判断重叠，可以参考leetcode：56. 合并区间

参考代码

import java.util.\*;

import java.util.Scanner;

public class quJianJiaoJi {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String[] str = in.nextLine().split(" ");

int[] arr = new int[str.length];

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

arr[i] = Integer.parseInt(str[i]);

}

// 先计算交集

List<int[]> res = new ArrayList<>();

for (int i = 0; i < arr.length; i += 2) {

for (int j = i + 2; j < arr.length; j += 2) {

int left = Math.max(arr[i], arr[j]);

int right = Math.min(arr[i + 1], arr[j + 1]);

if (left <= right) {

res.add(new int[]{left, right});

}

}

}

// 计算完交集，按从小到大排序，左边界升序，相同，有边界升序

int[][] ans = res.toArray(new int[res.size()][]);

Arrays.sort(ans, (a, b) -> (a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]));

// 求交集的并集

int[][] result = new int[ans.length][2];

int index = -1;

for (int[] an : ans) {

if (index == -1 || an[0] > result[index][1]) {

result[++index] = an;

} else {

result[index][1] = Math.max(result[index][1], an[1]);

}

}

int[][] last = Arrays.copyOf(result, index + 1);

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

System.out.print(last[i][0]);

System.out.print(" ");

System.out.print(last[i][1]);

if (i != last.length - 1) {

System.out.print(" ");

}

}

}

}

————————————————

版权声明：本文为CSDN博主「小朱小朱绝不服输」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/weixin_44052055/article/details/125833252>

## 叠积木

题目描述

给出一个列表如[[6,7,],[5,4],[3,2]],表示木块的长和宽，当木块的长和宽不大于另个木块的长和宽时，就可以放在上面，此外数组还可以左右翻转。求最多能搭多少层。

输入描述

一个二维数组，里面是每个积木的长和宽，可以左右翻转。

输出描述

最多能搭多少层。

样例

输入

[[5,4],[6,3],[6,7],[6,6],[4,6]]

1

输出

4

1

思路分析

首先对输入的积木进行处理，统一大的做长放第一个位置，小的做宽放第二个位置。

自定义排序，所有积木降序排，长度降序，相同则宽度降序。

动态规划求最大。定义一个 dp 数组，dp[i] 表示如果积木为 i 时，最大积木层数。j 表示前 i - 1 个积木，如果前 i - 1 个积木中宽度大于当前积木，dp[i]就等于两者最大值，则状态转移方程：

if (nums[j][1] >= nums[i][1]) {

dp[i] = Math.max(dp[i], dp[j] + 1); // 当前值，或从0到i-1中找到宽大于当前积木的

}

1

2

3

参考代码：

import java.util.Arrays;

import java.util.Scanner;

public class dieJiMu {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

String str = in.nextLine();

// 去掉字符串的所有括号，这里注意下字符串的解析

str = str.replaceAll("\\[", "");

str = str.replaceAll("\\]", "");

// str = str.replaceAll("\\s+", ""); // 去掉所有空格

String[] str1 = str.split(",");

int[][] nums = new int[str1.length / 2][2];

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

int num1 = Integer.parseInt(str1[2 \* i]);

int num2 = Integer.parseInt(str1[2 \* i + 1]);

nums[i][0] = Math.max(num1, num2); // 大的为长度

nums[i][1] = Math.min(num1, num2); // 小的为宽度

}

// 先对积木自定义排序，按长度降序，相同按宽度降序

Arrays.sort(nums, (o1, o2) -> o1[0] == o2[0] ? o2[1] - o1[1] : o2[0] - o1[0]);

// 动态规划，dp[i] 表示如果积木为 i 时，最大积木嵌套数

int[] dp = new int[nums.length];

Arrays.fill(dp, 1); // 初始化为1

int max = 1;

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

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

if (nums[j][1] >= nums[i][1]) {

dp[i] = Math.max(dp[i], dp[j] + 1); // 当前值，或从0到i-1中找到宽大于当前积木的

}

}

max = Math.max(dp[i], max);

}

System.out.println(max);

}

}

————————————————

版权声明：本文为CSDN博主「小朱小朱绝不服输」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/weixin_44052055/article/details/124261937>

## 高效的任务规划

题目描述：

你有 n 台机器编号为 1~n，每台都需要完成完成一项工作，机器经过配置后都能完成独立完成一项工作。

假设第 i 台机器你需要花 B 分钟进行设置，然后开始运行，J 分钟后完成任务。

现在，你需要选择布置工作的顺序，使得用最短的时间完成所有工作。

注意，不能同时对两台机器进行配置，但配置完成的机器们可以同时执行它们各自的工作。

注：此题对效率有要求，请考虑高效的实现方式

输入描述：

第一行输入代表总共有 M 组任务数据（1<M<=10）。

每组数第一行为一个整数，指定机器的数量 N（0<N<=1000）。

随后的 N 行每行两个整数，第一个表示 B（0<=B<=10000），第二个表示 J（0<=J<=10000）。

每组数据连续输入，不会用空行分隔。

各组任务单独计时。

输出描述：

对于每组任务，输出最短完成时间，且每组的结果独占一行。

例如，两组任务就应该有两行输出。

示例 1：

输入

1

1

2 2

1

2

3

输出

4

1

说明

第一行1为一组任务，

第二行1代表只有一台机器，

第三行表示该机器配置需2分钟，执行需要2分钟。

示例 1：

输入

2

2

1 1

2 2

3

1 1

2 2

3 3

1

2

3

4

5

6

7

8

输出

4

7

1

2

说明

第一行2代表两组任务，

第二行2代表第一组任务有2个机器，

第三行1 1代表机器1配置需要1分运行需要1分，

第四行2 2代表机器2配置需要2分运行需要2分，

第五行3代表第二组任务需要3个机器，

第6-8行分别表示3个机器的配置与运行时间。

参考代码：

解题思路：

题中要求总耗时最短，而且注意到题中：每次只能配置一台机器，那么一个简单道理就是让任务工作时间最长的机器先运行；

最浅显的解释是：如果让任务工作时间最短的在前面运行，那么同段时间的时间利用率就不是最高，因此并行数量越多越好，这样才会获得总体最短时间；

动态规划：dp[i]表示当前机器工作完成经过的总时间。

转移方程：因为第i台机器开始配置并工作必须是前i-1台机器都完成了配置，当前机器之前所有机器的总配置时间用last来表示，则第i台机器完成工作所用的总时间dp[i] = last + machine[i][0] + machine[i][1];(包含了此前所有机器的配置时间last)；

设最短的完成时间res，初始化res=0,last=0，则 res = max(res, dp[i]);

Java代码实现：

import java.util.\*;

public class assignmentArrangement {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

int M = in.nextInt();

for(int m = 0; m < M; m++) {

int N = in.nextInt();

// 动态规划，让任务工作时间最长的机器先运行，这样总体时间才最短啊

int[] dp = new int[N]; // dp[i]第i台机器工作完的时间

int last = 0; // 之前机器配置完成的时间

int res = 0; //总时间

int[][] machine = new int[N][2]; // 每组任务的N台机器的配置时间和工作时间

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

int B = in.nextInt();

int J = in.nextInt();

machine[i][0] = B;

machine[i][1] = J;

}

// lambda按第二元素降序排序，即按工作时间降序排序

Arrays.sort(machine, (e1, e2) -> (e2[1] - e1[1]));

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

dp[i] = last + machine[i][0] + machine[i][1];

last += machine[i][0];

res = Math.max(res, dp[i]);

}

System.out.println(res);

}

}

}

30

dp数组空间优化，用常量表示数组：

import java.util.\*;

public class assignmentArrangement {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

int M = in.nextInt();

for(int m = 0; m < M; m++) {

int N = in.nextInt();

// 动态规划，让任务工作时间最长的机器先运行，这样总体时间才最短啊

// 空间优化

int time = 0;

int last = 0; // 之前机器配置完成的时间

int res = 0; //总时间

int[][] machine = new int[N][2]; // 每组任务的N台机器的配置时间和工作时间

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

int B = in.nextInt();

int J = in.nextInt();

machine[i][0] = B;

machine[i][1] = J;

}

// lambda按第二元素降序排序，即按工作时间降序排序

Arrays.sort(machine, (e1, e2) -> (e2[1] - e1[1]));

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

time = last + machine[i][0] + machine[i][1];

last += machine[i][0];

res = Math.max(res, time);

}

System.out.println(res);

}

}

}

————————————————

版权声明：本文为CSDN博主「小朱小朱绝不服输」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/weixin\_44052055/article/details/123996124