od题

华为OD机考2025B卷 - 寻找最大价值的矿堆（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 观看文艺汇演问题（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 人气最高的店铺（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 书籍叠放（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 矩阵匹配（Java & Python& JS & C++ & C ）_华为od机试 2025b卷 - 矩阵匹配-CSDN博客

华为OD机考2025B卷 - 士兵过河（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 启动多任务排序（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 恢复数字序列（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 查找单入口空闲区域（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 食堂供餐（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 构成正方形的数量（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 核酸检测（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 字符串分割转换（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 停车场最大距离/找车位（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 小明减肥（Java & Python& JS & C++ & C ）_c++小明有n个可选运动,每个运动有对应卡路里,想选出其中k个运动且卡路里和为t。k,-CSDN博客

华为OD机考2025B卷 - 字符串加密（Java & Python& JS & C++ & C ）_华为od2025b卷 *?字符串-CSDN博客

华为OD机考2025B卷 - 无向图染色（Java & Python& JS & C++ & C ）_华为od机试真题 python 实现【无向图染色】-CSDN博客

华为OD机考2025B卷 - 差值数组不同的字符串（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 货币单位换算（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 最长的顺子（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 寻找最大价值的矿堆（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

给你一个由‘0’(空地)、‘1’(银矿)、‘2’(金矿) 组成的的地图，矿堆只能由上下左右相邻的金矿或银矿连接形成。超出地图范围可以认为是空地。
假设银矿价值1，金矿价值2 ，请你找出地图中最大价值的矿堆并输出该矿堆的价值。

输入描述

地图元素信息如：

> 22220
> 00000
> 00000
> 11111

- 地图范围最大 300*300
- 0 ≤ 地图元素 ≤ 2

输出描述

矿堆的最大价值

用例1

输入

Plain Text

```
22220
00000
00000
01111
```

输出

```
8
```

说明

用例2

输入

```
22220
00020
00010
01111
```

输出

```
15
```

说明

用例3

输入

```
20000
00020
00000
00111
```

输出

```
3
```

说明

解题思路

这道题目要求在一个由‘0’、‘1’和‘2’组成的地图中找到最大价值的矿堆。具体来说：

题意分析

1. **地图元素**：
   - ‘0’表示空地，没有价值。
   - ‘1’表示银矿，价值为 1。
   - ‘2’表示金矿，价值为 2。
2. **矿堆的定义**：
   - 矿堆由相邻的金矿和银矿组成，相邻指的是上下左右相连的单元格。
   - 每个矿堆的价值是其内部所有金矿和银矿的价值总和。
3. **目标**：
   - 计算并输出地图中所有矿堆中最大价值的矿堆。

输入与输出

- **输入**：一个地图的字符串表示，每行代表地图的一行，字符之间不需要空格。
- **输出**：单个整数，表示最大矿堆的价值。

示例解释

- **示例 1**：

```plain text
输入：
22220
00000
00000
01111
输出：
8
```

  - 这个地图中有一个金矿堆（包含四个‘2’）和一个银矿堆（包含五个‘1’）。金矿堆的价值是 $2 + 2 + 2 + 2 = 8$ 2+2+2+2=8 2+2+2+2=8，银矿堆的价值是 $1 + 1 + 1 + 1 + 1 = 5$ 1+1+1+1+1=5 1+1+1+1+1=5。所以，最大价值是 8。
- **示例 2**：

```
输入:
22220
00020
00010
01111
输出:
15
```

- 这个地图有多个金矿和银矿组成的矿堆。计算2+2+2+2+2+1+1+1+1=1，最大矿堆的价值为 15。

- **示例 3**：

```
输入:
20000
00020
00000
00111
输出:
3
```

- 在这个例子中，只有一个金矿和一个银矿形成的矿堆，其价值总和为 3。

Plain Text

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// 地图矩阵
int map[100][100];

// 上下左右，四个方向的偏移量
int offsets[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

int main() {
    // 读入地图信息
    vector<string> lines;
    string line;
    while (getline(cin, line)) {
        lines.push_back(line);
    }

    // 构建地图矩阵
    int rows = lines.size();
    int cols = lines[0].size();
    for (int i = 0; i < rows; i++) {
        line = lines[i];
        for (int j = 0; j < cols; j++) {
            map[i][j] = line[j] - '0';
        }
    }

    // 记录最大矿堆价值
    int maxVal = 0;

    // 遍历地图矩阵
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            // 如果点(i,j)没有被访问过，且点(i,j)上有矿，则进入深搜
            if (map[i][j] > 0) {
                queue<pair<int, int>> q;
                q.push(make_pair(i, j));

                int sum = 0;

                while (!q.empty()) {
                    pair<int, int> pos = q.front();
                    q.pop();
```

```
                        int x = pos.first, y = pos.second;

                        sum += map[x][y];
                        map[x][y] = 0;

                        // 遍历四个方向
                        for (auto offset : offsets) {
                            int newX = x + offset[0];
                            int newY = y + offset[1];

                            // 如果新位置在地图范围内，且有矿，则加入队列中
                            if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && map[newX][newY] > 0) {
                                q.push(make_pair(newX, newY));
                            }
                        }
                    }

                    // 更新最大矿堆价值
                    maxVal = max(maxVal, sum);
                }
            }
        }

        cout << maxVal << endl;

        return 0;
    }
```

Java

Java

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Scanner;

public class Main {
    // 地图矩阵
    static int[][] map;

    // 上下左右，四个方向的偏移量
    static int[][] offsets = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // 读入地图信息
        ArrayList<String> lines = new ArrayList<>();
        while (sc.hasNextLine()) {
            lines.add(sc.nextLine());
        }

        // 构建地图矩阵
        int rows = lines.size();
        int cols = lines.get(0).length();
        map = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            String line = lines.get(i);
            for (int j = 0; j < cols; j++) {
                map[i][j] = line.charAt(j) - '0';
            }
        }

        // 记录最大矿堆价值
        int maxVal = 0;

        // 遍历地图矩阵
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                // 如果点(i,j)没有被访问过，且点(i,j)上有矿，则进入深搜
                if (map[i][j] > 0) {
                    LinkedList<int[]> stack = new LinkedList<>();
                    stack.add(new int[]{i, j});

                    int sum = 0;

                    while (!stack.isEmpty()) {
```

```java
                        int[] pos = stack.removeLast();
                        int x = pos[0], y = pos[1];

                        sum += map[x][y];
                        map[x][y] = 0;

                        // 遍历四个方向
                        for (int[] offset : offsets) {
                            int newX = x + offset[0];
                            int newY = y + offset[1];

                            // 如果新位置在地图范围内，且有矿，则加入栈中
                            if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && map[newX][newY] > 0) {
                                stack.add(new int[]{newX, newY});
                            }
                        }
                    }

                    // 更新最大矿堆价值
                    maxVal = Math.max(maxVal, sum);
                }
            }
        }

        System.out.println(maxVal);
    }
}
```

完整用例

用例1

```Plain Text
22220
00000
00000
01111
```

用例2

```
22220
00020
00010
01111
```

用例3

```
20000
00020
00000
00111
```

用例4

```
00000
00000
00000
00000
```

用例5

```
22222
22222
22222
22222
```

用例6

```
2222222222
2222222222
2222222222
2222222222
2222222222
2222222222
2222222222
2222222222
2222222222
2222222222
```

用例7

```
2000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
```

用例8

```
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
```

用例9

```
2222222222
1111111111
2222222222
1111111111
2222222222
1111111111
2222222222
1111111111
2222222222
1111111111
```

用例10

```
2222222222
1111111111
2222200000
1111100000
2222200000
1111100000
2222200000
1111100000
2222222222
1111111111
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 用例1
- 用例2
- 用例3
- 解题思路
    - 题意分析
    - 输入与输出
    - 示例解释
- C++
- Java

- javaScript
- Python
- C语言
- 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4
  - 用例5
  - 用例6
  - 用例7
  - 用例8
  - 用例9
  - 用例10

> 来自：华为OD机考2025B卷 - 寻找最大价值的矿堆（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 观看文艺汇演问题（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

为了庆祝中国共产党成立100周年，某公园将举行多场文艺表演，很多演出都是同时进行，一个人只能同时观看一场演出，且不能迟到早退，由于演出分布在不同的演出场地，所以连续观看的演出最少有15分钟的时间间隔，
小明是一个狂热的文艺迷，想观看尽可能多的演出， 现给出演出时间表，请帮小明计算他最多能观看几场演出。

输入描述

第一行为一个数 N，表示演出场数

- 1 ≤ N ≤ 1000

接下来 N 行，每行有被空格分割的整数，第一个整数 T 表示演出的开始时间，第二个整数 L 表示演出的持续时间

- T 和 L 的单位为分钟
- 0 ≤ T ≤ 1440
- 0 < L ≤ 180

输出描述

输出最多能观看的演出场数。

示例1

输入

```Plain Text
2
720 120
840 120
```

输出

```
1
```

说明

示例2

输入

```
2
0 60
90 60
```

输出

```
2
```

说明

解题思路

题目理解：

这道题目要求我们帮小明计算他最多可以观看的文艺演出场数，给定多个演出时间表，每个演出有开始时间和持续时间，但小明有两个限制条件：

1. 小明不能迟到或早退，意味着他必须完整观看演出。
2. 连续观看的演出之间至少需要有 15 分钟的间隔，因为演出场地分布在不同地方，前往下一个演出场地需要时间。

这道题本质上是一个「区间调度问题」，需要我们在多个重叠的区间中，尽可能选择最多的不冲突区间。

示例解析：

示例1：

```
输入:
2
720 120
840 120
```

两场演出：

- 第一场：720 分钟开始 ，持续 120 分钟，到 840 分钟结束 。
- 第二场：840 分钟开始 ），持续 120 分钟，到 960 分钟结束 。

问题是：小明观看完第一场演出是 2:00 PM，但第二场演出也是 2:00 PM 开始，所以没有足够的时间移动到下一个场地（需要至少 15 分钟的间隔）。因此，小明只能观看 1 场演出。

示例2：

```
输入:
2
0 60
90 60
```

两场演出：

- 第一场：0 分钟开始（即 0:00 AM），持续 60 分钟，到 60 分钟结束（即 1:00 AM）。
- 第二场：90 分钟开始（即 1:30 AM），持续 60 分钟，到 150 分钟结束（即 2:30 AM）。

小明可以观看第一场演出结束后（1:00 AM），在 1:30 AM 前往下一个场地（刚好有 30 分钟的间隔）。因此，小明可以观看两场演出。

Java

```java
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        // 输入演出场数
        int n = in.nextInt();

        // 创建一个列表来存储演出时间表
        List<List<Integer>> schedule = new ArrayList<>();

        // 循环读取每个演出的开始时间和持续时间，并将其添加到演出时间表中
        for (int i = 0; i < n; i++) {
            int startTime = in.nextInt();
            int endTime = startTime + in.nextInt();
            schedule.add(List.of(startTime, endTime));
        }

        // 将演出时间表按照结束时间进行排序
        schedule.sort(Comparator.comparingInt(a -> a.get(1)));

        // 获取第一个演出的结束时间和初始化观看的演出场数
        int firstEndTime = schedule.get(0).get(1);
        int numShows = 1;

        // 遍历演出时间表中的每个演出时间段
        for (List<Integer> interval : schedule) {
            int startTime = interval.get(0);
            int endTime = interval.get(1);

            // 如果当前演出的开始时间与前一个演出的结束时间间隔大于等于15分钟，则可以观看该演出
            if (startTime - firstEndTime >= 15) {
                numShows++;
                firstEndTime = endTime;
            }
        }

        // 输出最多能观看的演出场数
        System.out.println(numShows);
    }
```

```
}
```

完整用例

用例1

Plain Text

```
2
720 120
840 120
```

用例2

Plain Text

```
2
0 60
90 60
```

用例3

Plain Text

```
3
600 60
660 60
720 60
```

用例4

Plain Text

```
4
600 60
660 60
720 60
780 60
```

用例5

```
5
600 60
660 60
720 60
780 60
840 60
```

用例6

```
10
0 60
75 60
150 60
225 60
300 60
375 60
450 60
525 60
600 60
675 60
```

用例7

```
10
0 60
75 60
150 60
225 60
300 60
375 60
450 60
525 60
600 60
675 30
```

用例8

```
10
0 60
75 60
150 60
225 60
300 60
375 60
450 60
525 60
600 60
660 60
```

用例9

```
1
720 120
```

用例10

```
2
0 60
90 60
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- 示例2
- 解题思路
    - 题目理解：
    - 示例解析：
        - 示例1：
        - 示例2：

- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4
  - 用例5
  - 用例6
  - 用例7
  - 用例8
  - 用例9
  - 用例10

> 来自：华为OD机考2025B卷 - 观看文艺汇演问题（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 人气最高的店铺（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看

题目描述

某购物城有m个商铺，现决定举办一场活动选出人气最高店铺。
活动共有n位市民参与，每位市民只能投一票，但1号店铺如果给该市民发放 q 元的购物补贴，该市民会改为投1号店铺。
请计算1号店铺需要最少发放多少元购物补贴才能成为人气最高店铺（即获得的票数要大于其他店铺），如果1号店铺本身就是票数最高店铺，返回0。

输入描述

第一行为小写逗号分割的两个整数n，m，其中：

- 第一个整数n表示参与的市民总数
- 第二个整数m代表店铺总数
- 1 ≤ n,m ≤ 3000

第2到n+1行，每行为小写逗号分割的两个整数p，q，表示市民的意向投票情况，其中每行的：

- 第一个整数p表示该市民意向投票给p号店铺
- 第二个整数q表示其改投1号店铺所需给予的q元购物补贴
- 1 ≤ p ≤ m
- 1 ≤ g ≤ 10^9

不考虑输入的格式问题

输出描述

1号店铺需要最少发放购物补贴金额

示例1

输入

```
5,5
2,10
3,20
4,30
5,40
5,90
```

输出

```
50
```

说明

> 说明
> 有5个人参与，共5个店铺。
> 如果选择发放 10元+20元+30元=60元 的补贴来抢2,3.4号店铺的票，总共发放了60元补贴（5号店铺有2票，1号店铺要3票才能胜出）
> 如果选择发放 10元+40元=50元 的补贴来抢2,5号店铺的票，总共发放了50元补贴（抢了5号店铺的票后，现在1号店铺只要2票就能胜出）
> 所以最少发放50元补贴

示例2

输入

```
5,5
2,10
3,20
4,30
5,80
5,90
```

输出

```
60
```

说明

> 说明
> 有5个人参与，共5个店铺。
> 如果选择发放 10元+20元+30元=60元 的补贴来抢2,3,4号店铺的票，总共发放了60元补贴(5号店铺有2票，1号店铺要3票才能胜出)
> 如果选择发放 10元+80元=90元 的补贴来抢2,5号店铺的票，总共发放了90元补贴(抢了5号店铺的票后，现在1号店铺只要2票就能胜出)
> 所以最少发放60元补贴

解题思路

Java

```java
import java.util.Scanner;
import java.util.*;
import java.util.HashMap;

public class Main {
    public static int minSubsidy = Integer.MAX_VALUE; // 最小补贴金额

    public static void main(String[] args) {
        // 输入
        Scanner in = new Scanner(System.in);
        int[] params = Arrays.stream(in.nextLine().split(",")).mapToInt(Integer::parseInt).toArray();
        int n = params[0]; // 参与的市民总数
        int m = params[1]; // 店铺总数

        // 初始化商店和投票人
        List<int[]> voters = new ArrayList<>(); // 市民投票信息列表
        Map<Integer, Integer> shopVotes = new HashMap<>(); // 商店票数映射
        shopVotes.put(1, 0); // 初始化1号店铺票数为0

        // 读取市民投票信息
        for (int i = 0; i < n; i++) {
            int[] nums = Arrays.stream(in.nextLine().split(",")).mapToInt(Integer::parseInt).toArray();
            if (nums[0] != 1) {
                voters.add(new int[]{nums[0], nums[1]});
            }
            shopVotes.put(nums[0], shopVotes.getOrDefault(nums[0], 0) + 1);
        }

        // 计算最小补贴金额
        calculateSubsidy(voters, 0, 0, shopVotes);
        System.out.println(minSubsidy);
    }

    // 计算最小补贴金额的递归函数
    public static void calculateSubsidy(List<int[]> voters, int index, int currentSubsidy, Map<Integer, Integer> shopVotes) {
        if (index == voters.size()) { // 边界条件：所有市民都已处理
            if (isTop1(shopVotes) && minSubsidy > currentSubsidy) {
                minSubsidy = currentSubsidy;
            }
            return;
        }
```

```java

        int[] currentVoter = voters.get(index); // 当前处理的市民
        int shopId = currentVoter[0]; // 市民投票的店铺ID
        int subsidy = currentVoter[1]; // 改投1号店铺所需补贴

        // 尝试不发放补贴
        calculateSubsidy(voters, index + 1, currentSubsidy, shopVotes);

        // 发放补贴
        shopVotes.put(shopId, shopVotes.get(shopId) - 1); // 原店铺票数减1
        shopVotes.put(1, shopVotes.get(1) + 1); // 1号店铺票数加1
        calculateSubsidy(voters, index + 1, currentSubsidy + subsidy, shopVotes);

        // 回溯
        shopVotes.put(shopId, shopVotes.get(shopId) + 1); // 原店铺票数加1
        shopVotes.put(1, shopVotes.get(1) - 1); // 1号店铺票数减1
    }

    // 判断1号店铺是否是票数最高的店铺
    public static boolean isTop1(Map<Integer, Integer> shopVotes) {
        int maxVotes = 0;
        for (Map.Entry<Integer, Integer> entry : shopVotes.entrySet()) {
            if (entry.getKey() != 1 && entry.getValue() >= maxVotes) {
                maxVotes = entry.getValue();
            }
        }
        return shopVotes.get(1) > maxVotes;
    }
}
```

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

// 定义投票者结构
typedef struct {
    int shopId;
    int subsidy;
} Voter;

// 定义商店结构
typedef struct {
    int shopId;
    int votes;
} Shop;

// 全局变量
int minSubsidy = INT_MAX; // 最小补贴金额

// 判断1号店铺是否票数最高
int isTop1(Shop* shops, int shopCount) {
    int maxVotes = 0;
    for (int i = 0; i < shopCount; i++) {
        if (shops[i].shopId != 1 && shops[i].votes >= maxVotes) {
            maxVotes = shops[i].votes;
        }
    }
    return shops[0].votes > maxVotes; // 假设shops[0]是1号店铺
}

// 根据商店ID查找商店在数组中的索引
int findShopIndex(Shop* shops, int shopCount, int shopId) {
    for (int i = 0; i < shopCount; i++) {
        if (shops[i].shopId == shopId) {
            return i;
        }
    }
    return -1;
}

// 计算最小补贴金额的递归函数
void calculateSubsidy(Voter* voters, int voterCount, int index, int currentSubsidy, Shop* shops, int shopCount) {
    if (index == voterCount) { // 边界条件：所有市民都已处理
```

```
        if (isTop1(shops, shopCount) && minSubsidy > currentSubsidy) {
            minSubsidy = currentSubsidy;
        }
        return;
    }

    Voter currentVoter = voters[index]; // 当前处理的市民
    int shopIndex = findShopIndex(shops, shopCount, currentVoter.shopId);
    int shop1Index = findShopIndex(shops, shopCount, 1);

    // 尝试不发放补贴
    calculateSubsidy(voters, voterCount, index + 1, currentSubsidy, shops, shopCount);

    // 发放补贴
    shops[shopIndex].votes--; // 原店铺票数减1
    shops[shop1Index].votes++; // 1号店铺票数加1
    calculateSubsidy(voters, voterCount, index + 1, currentSubsidy + currentVoter.subsidy, shops, shopCount);

    // 回溯
    shops[shopIndex].votes++; // 原店铺票数加1
    shops[shop1Index].votes--; // 1号店铺票数减1
}

// 分割字符串并转换为整数数组
int* splitAndParseInts(char* str, const char* delim, int* count) {
    int capacity = 10;
    int* result = (int*)malloc(capacity * sizeof(int));
    *count = 0;

    char* token = strtok(str, delim);
    while (token != NULL) {
        if (*count >= capacity) {
            capacity *= 2;
            result = (int*)realloc(result, capacity * sizeof(int));
        }
        result[(*count)++] = atoi(token);
        token = strtok(NULL, delim);
    }

    return result;
}

int main() {
    char line[1024];

    // 读取参数行
```

```
    fgets(line, sizeof(line), stdin);
    line[strcspn(line, "\n")] = 0; // 去除换行符

    int paramCount;
    int* params = splitAndParseInts(line, ",", &paramCount);
    int n = params[0]; // 参与的市民总数
    int m = params[1]; // 店铺总数
    free(params);

    // 初始化商店和投票者数组
    Voter* voters = (Voter*)malloc(n * sizeof(Voter));
    int voterCount = 0;

    Shop* shops = (Shop*)malloc((m + 1) * sizeof(Shop));
    int shopCount = 1; // 至少有1号店铺

    // 初始化1号店铺
    shops[0].shopId = 1;
    shops[0].votes = 0;

    // 读取市民投票信息
    for (int i = 0; i < n; i++) {
        fgets(line, sizeof(line), stdin);
        line[strcspn(line, "\n")] = 0; // 去除换行符

        int numCount;
        int* nums = splitAndParseInts(line, ",", &numCount);

        // 查找或添加商店
        int shopIndex = findShopIndex(shops, shopCount, nums[0]);
        if (shopIndex == -1) {
            shops[shopCount].shopId = nums[0];
            shops[shopCount].votes = 0;
            shopIndex = shopCount;
            shopCount++;
        }

        // 增加商店票数
        shops[shopIndex].votes++;

        // 如果不是投给1号店铺，添加到voters数组
        if (nums[0] != 1) {
            voters[voterCount].shopId = nums[0];
            voters[voterCount].subsidy = nums[1];
            voterCount++;
        }

        free(nums);
```

```
    }

    // 计算最小补贴金额
    calculateSubsidy(voters, voterCount, 0, 0, shops, shopCount);

    // 输出结果
    printf("%d\n", minSubsidy);

    // 释放内存
    free(voters);
    free(shops);

    return 0;
}
```

完整用例

用例1

Plain Text

```
3,3
2,10
3,20
3,30
```

用例2

Plain Text

```
4,4
1,0
2,10
3,20
4,30
```

用例3

```
5,3
1,0
2,10
2,20
3,30
3,40
```

用例4

```
6,4
1,10
2,10
2,20
3,30
3,40
4,50
```

用例5

```
5,5
2,10
3,20
4,30
5,40
5,90
```

用例6

```
5,5
2,10
3,20
4,30
5,80
5,90
```

用例7

```
6,6
2,5
3,10
4,15
5,20
6,25
6,30
```

用例8

```
4,3
2,15
3,10
3,15
4,15
```

用例9

```
5,4
2,10
3,20
4,30
4,40
4,50
```

用例10

```
7,6
1,1
2,5
3,10
4,15
5,20
6,25
6,30
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- 示例2
- 解题思路
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4
  - 用例5
  - 用例6
  - 用例7
  - 用例8
  - 用例9
  - 用例10

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- 示例2
- 解题思路
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4
  - 用例5
  - 用例6
  - 用例7
  - 用例8
  - 用例9
  - 用例10

> 来自: 华为OD机考2025B卷 - 人气最高的店铺（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 书籍叠放（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

书籍的长、宽都是整数对应(l,w)。如果书A的长宽度都比B长宽大时，则允许将B排列放在A上面。
现在有一组规格的书籍，书籍叠放时要求书籍不能做旋转，请计算最多能有多少个规格书籍能叠放在一起。

输入描述

输入：books = [[20,16],[15,11],[10,10],[9,10]]
说明：总共4本书籍，第一本长度为20宽度为16；第二本书长度为15宽度为11，依次类推，最后一本书长度为9宽度为10.

输出描述

输出：3
说明: 最多3个规格的书籍可以叠放到一起, 从下到上依次为: [20,16],[15,11],[10,10]

示例1

输入

Plain Text

```
[[20,16],[15,11],[10,10],[9,10]]
```

输出

Plain Text

```
3
```

说明

Java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);


        String input = sc.nextLine();

        // 去掉字符串的首尾方括号，并按书籍分隔符 "],[" 分割，生成每本书的字符串数组
        String[] bookStrings = input.substring(1, input.length() - 1).split("(?<=]),(?=\\[)");

        // 定义二维数组存储所有书的长宽信息
        Integer[][] books = new Integer[bookStrings.length][];

        // 遍历每本书的字符串，将其转换为长度和宽度的整数数组
        for (int i = 0; i < bookStrings.length; i++) {
            // 去掉当前书的方括号，并按逗号分割
            String[] bookValues = bookStrings[i].substring(1, bookStrings[i].length() - 1).split(",");
            books[i] = new Integer[bookValues.length];

            // 将分割后的字符串转换为整数并存入书籍数组
            for (int j = 0; j < bookValues.length; j++) {
                books[i][j] = Integer.parseInt(bookValues[j]);
            }
        }

        // 按照书的长度从小到大排序，如果长度相同，则按宽度从大到小排序
        Arrays.sort(books, Comparator.comparing((Integer[] arr) -> arr[0])
                    .thenComparing((Integer[] arr) -> -arr[1]));

        // 提取所有书籍的宽度，准备计算最长不下降子序列
        int[] widths = Arrays.stream(books).mapToInt(book -> book[1]).toArray();

        // 定义一个数组用于保存最长不下降子序列（LIS）
        int[] lis = new int[widths.length];
        int len = 0; // 用于记录 LIS 的长度

        // 遍历所有书籍的宽度，使用二分查找优化 LIS 计算
        for (int i = 0; i < widths.length; i++) {
            // 使用二分查找确定宽度插入 LIS 的位置
```

```java
            int idx = Arrays.binarySearch(lis, 0, len, widths[i]);

            // 如果找不到合适位置，binarySearch 返回负数，需转换为插入位置
            if (idx < 0) {
                idx = -(idx + 1); // 二分查找返回负数时转换为插入位置
            }

            // 更新 LIS，若插入位置是当前长度 len 位置，表示找到了新的子序列元素，增加 len
            lis[idx] = widths[i];
            if (idx == len) {
                len++; // 子序列长度增加
            }
        }

        // 输出最长不下降子序列的长度，即最大可以叠放的书籍数量
        System.out.println(len);
    }
}
```

Python

Plain Text

```
import sys
import bisect

# 读取用户输入的书籍数据，假设输入形如 [[20,16],[21,15],[22,14]]
books = eval(input())  # eval 将输入的字符串转化为实际的 Python 数据结构（二维列表）

# 对书籍列表按照书的长度从小到大排序，如果长度相同则按照宽度从大到小排序
books.sort(key=lambda x: (x[0], -x[1]))
# lambda 函数 x[0] 表示按长度排序，x[1] 前面的负号表示宽度按照降序排列

# 提取所有书籍的宽度，生成一个列表
widths = list(map(lambda x: x[1], books))  # 使用 map 提取每本书的宽度并转换为列表

# 初始化最长不下降子序列（LIS），首先将第一个宽度值放入 LIS 列表
lis = [widths[0]]

# 遍历所有书籍的宽度，从第二本书开始
for i in range(1, len(widths)):
    # 如果当前书的宽度大于 LIS 列表中的最后一个元素，则将其追加到 LIS 中
    if widths[i] > lis[-1]:
        lis.append(widths[i])
    # 如果当前书的宽度小于 LIS 列表中的第一个元素，替换第一个元素
    elif widths[i] < lis[0]:
        lis[0] = widths[i]
    # 否则，使用二分查找找到第一个大于或等于当前宽度的位置，并替换它
    else:
        idx = bisect.bisect_left(lis, widths[i])
        lis[idx] = widths[i]

# 输出 LIS 的长度，即可以叠放的最多书籍数量
print(len(lis))
```

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 用于书籍存储的结构体，每本书包含长度和宽度
typedef struct {
    int length;
    int width;
} Book;

// 比较函数，用于 qsort 排序书籍
// 先按长度升序，如果长度相同则按宽度降序
int compare(const void* a, const void* b) {
    Book* bookA = (Book*)a;
    Book* bookB = (Book*)b;
    if (bookA->length == bookB->length) {
        return bookB->width - bookA->width; // 宽度降序
    }
    return bookA->length - bookB->length; // 长度升序
}

// 二分查找优化，寻找插入点
int lower_bound(int* lis, int len, int target) {
    int left = 0, right = len;
    while (left < right) {
        int mid = (left + right) / 2;
        if (lis[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

int main() {
    // 获取用户输入的一行数据
    char input[1000]; // 假设输入不会超过1000字符
    fgets(input, sizeof(input), stdin);

    Book books[100]; // 假设最多有100本书
    int bookCount = 0; // 记录书的数量

    char* pos = input + 1; // 跳过输入的第一个字符 '['
    // 解析输入字符串，提取每本书的长和宽
```

```
    while (*pos != ']') { // 当遇到输入的倒数第二个字符停止
        int length, width;
        sscanf(pos, "[%d,%d]", &length, &width); // 解析出一本书的长和宽
        books[bookCount].length = length;
        books[bookCount].width = width;
        bookCount++;

        // 移动到下一本书的位置，跳过当前的 "],"
        pos = strstr(pos, "],");
        if (pos) {
            pos += 2; // 跳过 "],"
        } else {
            break; // 到达最后一本书
        }
    }

    // 对书籍进行排序，先按长度升序，如果长度相同则按宽度降序
    qsort(books, bookCount, sizeof(Book), compare);

    // 提取所有书的宽度，方便之后寻找最长不下降子序列
    int widths[100]; // 宽度数组
    for (int i = 0; i < bookCount; i++) {
        widths[i] = books[i].width;
    }

    // 寻找宽度的最长不下降子序列(LIS)，使用二分查找优化
    int lis[100]; // LIS 数组
    int len = 0; // 记录最长子序列的长度

    for (int i = 0; i < bookCount; i++) {
        // 找到当前宽度应该插入的位置，保持宽度不下降
        int idx = lower_bound(lis, len, widths[i]);
        if (idx == len) {
            // 如果插入位置等于当前 LIS 的长度，说明找到了一个新的更大的宽度
            lis[len++] = widths[i];
        } else {
            // 否则，用当前宽度替换掉对应位置的值，以保持 LIS 的最小值
            lis[idx] = widths[i];
        }
    }

    // 输出最长不下降子序列的长度，也就是最多可以叠放的书籍数量
    printf("%d\n", len);

    return 0;
}
```

完整用例

用例1

```
[[1,1],[2,2],[3,3],[4,4]]
```

用例2

```
[[1,1],[2,2],[2,2],[3,3]]
```

用例3

```
[[1,1],[2,2],[2,2],[2,2]]
```

用例4

```
[[1,1],[2,2],[3,3],[1,2]]
```

用例5

```
[[1,1],[2,2],[3,3],[2,1]]
```

用例6

```
[[20,16],[15,11],[10,10],[9,10],[8,8],[7,7],[6,6],[5,5],[4,4],[3,3],[2,2],[1,1]]
```

用例7

```
[[20,16],[15,11],[10,10],[9,10],[8,8],[7,7],[6,6],[5,5],[4,4],[3,3],[2,2],[2,2]]
```

用例8

```
[[20,16],[15,11],[10,10],[9,10],[8,8],[7,7],[6,6],[5,5],[4,4],[4,4],[4,4],[4,4]]
```

用例9

```
[[20,16],[15,11],[10,10],[9,10],[8,8],[7,7],[6,6],[5,5],[5,5],[5,5],[5,5],[5,5]]
```

用例10

```
[[20,16],[15,11],[10,10],[9,10],[8,8],[7,7],[6,6],[6,6],[6,6],[6,6],[6,6],[6,6]]
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
  - 用例1
  - 用例2

- 用例3
- 用例4
- 用例5
- 用例6
- 用例7
- 用例8
- 用例9
- 用例10

> 来自：华为OD机考2025B卷 - 书籍叠放（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 矩阵匹配（Java & Python& JS & C++ & C ）_华为od机试 2025b卷 - 矩阵匹配-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看

题目描述

从一个 N * M（N ≤ M）的矩阵中选出 N 个数，任意两个数字不能在同一行或同一列，求选出来的 N 个数中第 K 大的数字的最小值是多少。

输入描述

输入矩阵要求：1 ≤ K ≤ N ≤ M ≤ 150
输入格式：

> N M K
> N*M矩阵

输出描述

N*M 的矩阵中可以选出 M! / N! 种组合数组，每个组合数组种第 K 大的数中的最小值。无需考虑重复数字，直接取字典排序结果即可。
注意：结果是第 K 大的数字的最小值

用例

输入

```plain text
3 4 2
1 5 6 6
8 3 4 3
6 8 6 3
```

输出

```
3
```

> N*M的矩阵中可以选出 M！/ N！种组合数组，每个组合数组种第 K 大的数中的最小值；
> 上述输入中选出数组组合为：
> 1,3,6;
> 1,3,3;
> 1,4,8;
> 1,4,3;
> ...
> 上述输入样例中选出的组合数组有24种，最小数组为1,3,3，则第2大的最小值为3

C++

Plain Text

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

int n, m, k;  // n、m、k 分别表示矩阵的行数、列数和要求的第K大数
vector<vector<int>> matrix;  // matrix 用于存储输入的矩阵
vector<int> match;  // match 数组用于存储匹配信息，match[j] = i 表示第j列与第i行匹配
vector<bool> vis;  // vis 数组用于标记每一列在当前增广路中是否被访问过

// 深度优先搜索寻找增广路径
bool dfs(int i, int currentVal) {
    for (int j = 0; j < m; j++) {
        // 检查列j是否未被访问过且第i行第j列的值小于等于currentVal
        if (!vis[j] && matrix[i][j] <= currentVal) {
            vis[j] = true;  // 标记列j为已访问
            // 如果列j未匹配或者列j的匹配行可以匹配到其他列
            if (match[j] == -1 || dfs(match[j], currentVal)) {
                match[j] = i;  // 将列j与行i匹配
                return true;  // 找到增广路径
            }
        }
    }
    return false;  // 没有找到增广路径
}

// 检查当前值是否满足条件
bool check(int currentVal) {
    match.assign(m, -1);  // 初始化匹配数组
    vis.assign(m, false);  // 初始化访问标记数组
    int smallerCount = 0;  // 统计满足条件的数量
    for (int i = 0; i < n; i++) {
        fill(vis.begin(), vis.end(), false);  // 每次搜索前重置访问标记
        if (dfs(i, currentVal)) {
            smallerCount++;  // 如果找到增广路径，则计数增加
        }
    }
    return smallerCount >= n - k + 1;  // 检查是否有足够的小于等于currentVal的数
}

int main() {
    cin >> n >> m >> k;  // 读取行数、列数和k值
```

```
    int min = 1, maxT = INT_MIN;  // 初始化二分查找的上下界
    matrix.assign(n, vector<int>(m));  // 初始化矩阵
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> matrix[i][j];  // 读取矩阵元素
            maxT = max(maxT, matrix[i][j]);  // 更新矩阵元素的最大值，作为二分查找的上界
        }
    }

    // 二分查找确定第K大的数的最小可能值
    while (min <= maxT) {
        int mid = (min + maxT) / 2;  // 取中间值
        if (check(mid)) {
            maxT = mid - 1;  // 如果当前中间值满足条件，则尝试寻找更小的值
        } else {
            min = mid + 1;  // 如果不满足条件，则尝试寻找更大的值
        }
    }
    cout << min << endl;  // 输出最终结果
    return 0;
}
```

Java

Java

```java
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    static int n, m, k;  // n、m、k 分别表示矩阵的行数、列数和要求的第K大数
    static int[][] matrix;  // matrix 用于存储输入的矩阵
    static int[] match;  // match 数组用于存储匹配信息，match[j] = i 表示第j列与第i行匹配
    static boolean[] vis;  // vis 数组用于标记每一列在当前增广路中是否被访问过

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();  // 读取行数
        m = sc.nextInt();  // 读取列数
        k = sc.nextInt();  // 读取k值

        int min = 1, max = Integer.MIN_VALUE;  // 初始化二分查找的上下界
        matrix = new int[n][m];  // 初始化矩阵
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                matrix[i][j] = sc.nextInt();  // 读取矩阵元素
                max = Math.max(max, matrix[i][j]);  // 更新矩阵元素的最大值，作为二分查找的上界
            }
        }

        // 二分查找确定第K大的数的最小可能值
        while (min <= max) {
            int mid = (min + max) / 2;  // 取中间值
            if (check(mid)) {
                max = mid - 1;  // 如果当前中间值满足条件，则尝试寻找更小的值
            } else {
                min = mid + 1;  // 如果不满足条件，则尝试寻找更大的值
            }
        }
        System.out.println(min);  // 输出最终结果
    }

    // 检查当前值是否满足条件
    public static boolean check(int currentVal) {
        match = new int[m];  // 初始化匹配数组
        Arrays.fill(match, -1);  // 将所有列初始化为未匹配状态
        vis = new boolean[m];  // 初始化访问标记数组
        int smallerCount = 0;  // 统计满足条件的数量
        for (int i = 0; i < n; i++) {
```

```java
            Arrays.fill(vis, false);  // 每次搜索前重置访问标记
            if (dfs(i, currentVal)) {
                smallerCount++;  // 如果找到增广路径，则计数增加
            }
        }
        return smallerCount >= n - k + 1;  // 检查是否有足够的小于等于currentVal的数
    }

    // 深度优先搜索寻找增广路径
    public static boolean dfs(int i, int currentVal) {
        for (int j = 0; j < m; j++) {
            // 检查列j是否未被访问过且第i行第j列的值小于等于currentVal
            if (!vis[j] && matrix[i][j] <= currentVal) {
                vis[j] = true;  // 标记列j为已访问
                // 如果列j未匹配或者列j的匹配行可以匹配到其他列
                if (match[j] == -1 || dfs(match[j], currentVal)) {
                    match[j] = i;  // 将列j与行i匹配
                    return true;  // 找到增广路径
                }
            }
        }
        return false;  // 没有找到增广路径
    }
}
```

javaScript

Plain Text

```
const readline = require('readline');

// 创建 readline 接口实例
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

let n, m, k;
let matrix = [];
let match, vis;

// 深度优先搜索寻找增广路径
const dfs = (i, currentVal) => {
    for (let j = 0; j < m; j++) {
        if (!vis[j] && matrix[i][j] <= currentVal) {
            vis[j] = true;  // 标记第j列为已访问
            if (match[j] === -1 || dfs(match[j], currentVal)) {
                match[j] = i;  // 将第j列与第i行匹配
                return true;
            }
        }
    }
    return false;
};

// 检查当前值是否满足条件
const check = (currentVal) => {
    match = Array(m).fill(-1);  // 初始化匹配数组
    let smallerCount = 0;  // 统计满足条件的数量

    for (let i = 0; i < n; i++) {
        vis = Array(m).fill(false);  // 每次搜索前重置访问标记
        if (dfs(i, currentVal)) {
            smallerCount++;  // 如果找到增广路径，则计数增加
        }
    }

    return smallerCount >= n - k + 1;
};

// 从标准输入读取数据
rl.on('line', (line) => {
    if (!n) {
        [n, m, k] = line.split(' ').map(Number);
```

```
    } else {
        matrix.push(line.split(' ').map(Number));
        if (matrix.length === n) {
            rl.close();
        }
    }
}).on('close', () => {
    // 初始化二分查找的上下界
    let minVal = 1, maxVal = -Number.MAX_SAFE_INTEGER;

    // 更新矩阵元素的最大值
    matrix.forEach(row => {
        maxVal = Math.max(maxVal, ...row);
    });

    // 二分查找确定第K大的数的最小可能值
    while (minVal <= maxVal) {
        let mid = Math.floor((minVal + maxVal) / 2);
        if (check(mid)) {
            maxVal = mid - 1;
        } else {
            minVal = mid + 1;
        }
    }

    // 输出最终结果
    console.log(minVal);
});
```

Python

Plain Text

```python
import sys

def dfs(i, current_val):
    """
    深度优先搜索寻找增广路径
    :param i: 当前正在处理的行索引
    :param current_val: 当前考虑的值
    :return: 如果找到增广路径，返回True；否则返回False
    """
    for j in range(m):
        # 检查第j列是否未被访问过且第i行第j列的值小于等于current_val
        if not vis[j] and matrix[i][j] <= current_val:
            vis[j] = True  # 标记第j列为已访问
            # 如果第j列未匹配或其匹配的行可以找到其他匹配列
            if match[j] == -1 or dfs(match[j], current_val):
                match[j] = i  # 将第j列与第i行匹配
                return True
    return False

def check(current_val):
    """
    检查当前值是否满足条件
    :param current_val: 当前考虑的值
    :return: 如果满足条件，返回True；否则返回False
    """
    global match, vis
    match = [-1] * m  # 初始化匹配数组，所有列都标记为未匹配
    vis = [False] * m  # 初始化访问标记数组
    smaller_count = 0  # 统计满足条件的数量

    for i in range(n):
        vis = [False] * m  # 每次搜索前重置访问标记
        if dfs(i, current_val):
            smaller_count += 1  # 如果找到增广路径，则计数增加

    return smaller_count >= n - k + 1  # 检查是否有足够的小于等于current_val的数

# 读取输入
n, m, k = map(int, input().split())  # 读取行数、列数和k值

# 初始化矩阵
matrix = []
for _ in range(n):
    matrix.append(list(map(int, input().split())))
```

```python

# 初始化二分查找的上下界
min_val, max_val = 1, -sys.maxsize

# 更新矩阵元素的最大值，作为二分查找的上界
for row in matrix:
    max_val = max(max_val, max(row))

# 二分查找确定第K大的数的最小可能值
while min_val <= max_val:
    mid = (min_val + max_val) // 2
    if check(mid):
        max_val = mid - 1
    else:
        min_val = mid + 1

# 输出最终结果
print(min_val)
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define MAX_N 200
#define MAX_M 200

int n, m, k;  // n、m、k 分别表示矩阵的行数、列数和要求的第K大数
int matrix[MAX_N][MAX_M];  // matrix 用于存储输入的矩阵
int match[MAX_M];  // match 数组用于存储匹配信息
bool vis[MAX_M];  // vis 数组用于标记每一列在当前增广路中是否被访问过

bool dfs(int i, int current_val) {
    for (int j = 0; j < m; j++) {
        if (!vis[j] && matrix[i][j] <= current_val) {
            vis[j] = true;
            if (match[j] == -1 || dfs(match[j], current_val)) {
                match[j] = i;
                return true;
            }
        }
    }
    return false;
}

bool check(int current_val) {
    for (int j = 0; j < m; j++) {
        match[j] = -1;  // 初始化匹配数组
    }

    int smaller_count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            vis[j] = false;  // 初始化访问标记数组
        }
        if (dfs(i, current_val)) {
            smaller_count++;
        }
    }
    return smaller_count >= n - k + 1;
}

int main() {
    scanf("%d %d %d", &n, &m, &k);  // 读取行数、列数和k值

```

```
    // 初始化矩阵
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    // 初始化二分查找的上下界
    int min_val = 1, max_val = INT_MIN;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (matrix[i][j] > max_val) {
                max_val = matrix[i][j];  // 更新矩阵元素的最大值
            }
        }
    }

    // 二分查找确定第K大的数的最小可能值
    while (min_val <= max_val) {
        int mid = (min_val + max_val) / 2;
        if (check(mid)) {
            max_val = mid - 1;
        } else {
            min_val = mid + 1;
        }
    }

    // 输出最终结果
    printf("%d\n", min_val);

    return 0;
}
```

完整用例

用例1

Plain Text

```
3 3 1
1 2 3
4 5 6
7 8 9
```

用例2

```
4 5 2
10 10 10 10 10
20 20 20 20 20
30 30 30 30 30
40 40 40 40 40
```

用例3

```
10 12 10
100 101 102 103 104 105 106 107 108 109 110 111
200 201 202 203 204 205 206 207 208 209 210 211
300 301 302 303 304 305 306 307 308 309 310 311
400 401 402 403 404 405 406 407 408 409 410 411
500 501 502 503 504 505 506 507 508 509 510 511
600 601 602 603 604 605 606 607 608 609 610 611
700 701 702 703 704 705 706 707 708 709 710 711
800 801 802 803 804 805 806 807 808 809 810 811
900 901 902 903 904 905 906 907 908 909 910 911
1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011
```

用例4

```
9 11 9
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 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88
89 90 91 92 93 94 95 96 97 98 99
```

用例5

```
5 7 3
1 1 1 1 1 1 1
2 2 2 2 2 2 2
3 3 3 3 3 3 3
4 4 4 4 4 4 4
5 5 5 5 5 5 5
```

用例6

```
2 5 1
1 3 5 7 9
2 4 6 8 10
```

用例7

```
8 10 8
100 90 80 70 60 50 40 30 20 10
110 100 90 80 70 60 50 40 30 20
120 110 100 90 80 70 60 50 40 30
130 120 110 100 90 80 70 60 50 40
140 130 120 110 100 90 80 70 60 50
150 140 130 120 110 100 90 80 70 60
160 150 140 130 120 110 100 90 80 70
170 160 150 140 130 120 110 100 90 80
```

用例8

```
5 8 2
5 7 5 4 3 2 9 4
8 4 7 6 4 9 9 9
6 1 1 5 6 1 3 4
2 7 8 7 9 2 5 6
6 7 2 5 2 2 3 7
```

用例9

```
7 8 3
6 7 6 7 3 6 7 5
5 4 3 3 3 9 9 4
5 3 4 7 1 7 6 6
3 3 9 1 8 2 2 5
8 4 2 2 1 9 8 6
6 9 4 2 5 6 1 7
5 5 7 7 1 1 8 1
```

用例10

```
4 6 1
2 8 1 7 8 9
2 2 6 8 3 7
9 4 4 6 9 5
1 6 5 1 1 6
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 用例
- C++
- Java
- javaScript
- Python
- C语言
- 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4
  - 用例5
  - 用例6
  - 用例7
  - 用例8

- 用例9
- 用例10

> 来自：华为OD机考2025B卷 - 矩阵匹配（Java & Python& JS & C++ & C ）_华为od机试 2025b卷 - 矩阵匹配-CSDN博客

华为OD机考2025B卷 - 士兵过河（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看

题目描述

一支N个士兵的军队正在趁夜色逃亡，途中遇到一条湍急的大河。
敌军在T的时长后到达河面，没到过对岸的士兵都会被消灭。
现在军队只找到了1只小船，这船最多能同时坐上2个士兵。

1. 当1个士兵划船过河，用时为 a[i]；0 <= i < N
2. 当2个士兵坐船同时划船过河时，用时为max(a[j],a[i])两士兵中用时最长的。
3. 当2个士兵坐船1个士兵划船时，用时为 a[i]*10；a[i]为划船士兵用时。
4. 如果士兵下河游泳，则会被湍急水流直接带走，算作死亡。

请帮忙给出一种解决方案，保证存活的士兵最多，且过河用时最短。

输入描述

第一行：N 表示士兵数(0<N<1,000,000)
第二行：T 表示敌军到达时长(0 < T < 100,000,000)
第三行：a[0] a[1] ... a[i]... a[N- 1]
a[i]表示每个士兵的过河时长。
(10 < a[i]< 100; 0<= i< N)

备注

1）两个士兵的同时划船时，如果划速不同则会导致船原地转圈圈；所以为保持两个士兵划速相同，则需要向划的慢的士兵看齐。
2）两个士兵坐船时，重量增加吃水加深，水的阻力增大；同样的力量划船速度会变慢；
3）由于河水湍急大量的力用来抵消水流的阻力，所以2）中过河用时不是a[i] *2，
而是a[i] * 10。

输出描述

第一行：”最多存活士兵数”“最短用时”

示例1

输入

```
5
43
12 13 15 20 50
```

输出

```
3 40
```

说明

> 可以达到或小于43的一种方案：
> 第一步：a[0] a[1] 过河用时：13
> 第二步：a[0] 返回用时：12
> 第三步：a[0] a[2] 过河用时：15

示例2

输入

```
5
130
50 12 13 15 20
```

输出

```
5 128
```

说明

> 可以达到或小于130的一种方案：
> 第一步：a[1] a[2] 过河用时：13
> 第二步：a[1] 返回用时：12
> 第三步：a[0] a[4] 过河用时：50
> 第四步：a[2] 返回用时：13
> 第五步：a[1] a[2] 过河用时：13

> 第六步：a[1] 返回用时：12
> 第七步：a[1] a[3] 过河用时：15
> 所以输出为：
> `5 128`

示例3

输入

```
7
171
25 12 13 15 20 35 20
```

输出

```
7 171
```

说明

> 可以达到或小于171的一种方案：
> 第一步：a[1] a[2] 过桥用时：13
> 第二步：a[1] 带火把返回用时：12
> 第三步：a[0] a[5] 过桥用时：35
> 第四步：a[2] 带火把返回用时：13
> 第五步：a[1] a[2] 过桥用时：13
> 第六步：a[1] 带火把返回用时：12
> 第七步：a[4] a[6] 过桥用时：20
> 第八步：a[2] 带火把返回用时：13
> 第九步：a[1] a[3] 过桥用时：15
> 第十步：a[1] 带火把返回用时：12
> 第十一步：a[1] a[2] 过桥用时：13
> 所以输出为：

```
7 171
```

解题思路

这道题目是一个关于优化的组合问题，目标是最大化过河的士兵数量，同时确保这些士兵在敌军到达之前安全地渡过河。

2. 处理特定情况

- 如果只有一名士兵，直接比较其过河时间与 `T` 的大小，进行输出。
- 对于两名士兵，可以直接输出两者中较大的划船时间，但仍需确保其不超过 `T` 。

3. 排序

- 将士兵的过河时间升序排序，这有助于在后续步骤中更有效地安排过河策略。

4. 动态规划策略

- 使用动态规划来计算每种情况下的最小时间来确保士兵过河的安全。在这个过程中我们使用一个数组 `dp` 来存储每个状态的最小过河时间。
- 对于 `dp[i]` ，我们可以通过不同方式（1人过、2人过、1人返回等）更新当前的最小过河时间。
- 具体的转移方程为：
  - 一人过河加上返回时间
  - 两人同时过河加上返回时间
- 每次更新 `dp[i]` 时，如果发现超过 `T` 就可以输出当前存活的士兵数量和其过完河的时间。

Java

```java
import java.util.Scanner;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        // 创建Scanner对象
        Scanner sc = new Scanner(System.in);
        // 读取输入的n和limit
        int n = sc.nextInt();
        int limit = sc.nextInt();
        // 创建长度为n的整型数组time
        int[] time = new int[n];
        // 循环读取n个时间并存入数组time中
        for (int i = 0; i < n; i++) {
            time[i] = sc.nextInt();
        }
        // 对数组time进行排序
        Arrays.sort(time);
        // 创建长度为n的整型数组dp
        int[] dp = new int[n];
        // 循环计算dp数组的值
        for (int i = 0; i < n; i++) {
            if (i == 0) {
                // 如果i等于0，将dp[0]赋值为time[0]
                dp[0] = time[0];
                // 如果dp[0]大于limit，输出"0 0"并结束程序
                if (dp[0] > limit) {
                    System.out.println("0 0");
                    return;
                }
            } else if (i == 1) {
                // 如果i等于1，将dp[1]赋值为time[1]
                dp[1] = time[1];
            } else {
                // 如果i大于1，计算t1和t2的值并将较小值赋给dp[i]
                int t1 = dp[i - 1] + time[i] + time[0];
                int t2 = dp[i - 2] + time[0] + time[i] + time[1] + time[1];
                dp[i] = Math.min(t1, t2);
            }
            // 如果dp[i]大于limit，输出i和dp[i-1]的值并结束程序
            if (dp[i] > limit) {
                System.out.println(i + " " + dp[i - 1]);
                return;
            }
```

```java
        }
        // 输出n和dp[n-1]的值
        System.out.println(n + " " + dp[n - 1]);
    }
}
```

Python

```python
n = int(input())
limit = int(input())
time = list(map(int, input().split()))

# 对时间列表进行排序
time.sort()

# 初始化dp数组
dp = [0] * n

# 动态规划
for i in range(n):
    if i == 0: # 当i=0时，dp[0]为time[0]
        dp[0] = time[0]
        if dp[0] > limit: # 如果dp[0]已经超过了限制，则输出0 0并退出程序
            print("0 0")
            exit()
    elif i == 1: # 当i=1时，dp[1]为time[1]
        dp[1] = time[1]
    else: # 当i>=2时，dp[i]为t1和t2中的最小值
        t1 = dp[i - 1] + time[i] + time[0]
        t2 = dp[i - 2] + time[0] + time[i] + time[1] + time[1]
        dp[i] = min(t1, t2)
    if dp[i] > limit: # 如果dp[i]已经超过了限制，则输出i和dp[i-1]并退出程序
        print(i, dp[i - 1])
        exit()

# 输出n和dp[n-1]
print(n, dp[n - 1])
```

JavaScript

Plain Text

```
const readline = require('readline');
// 创建readline实例
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

// 定义变量
let n, limit, time, dp;

// 监听输入事件
rl.on('line', (line) => {
  // 如果n没有值，将输入的值赋给n
  if (!n) {
    n = parseInt(line.trim());
  }
  // 如果limit没有值，将输入的值赋给limit
  else if (!limit) {
    limit = parseInt(line.trim());
  }
  // 如果time没有值，将输入的值赋给time
  else {
    time = line.trim().split(' ').map(Number);
    // 将time数组从小到大排序
    time.sort();
    // 初始化dp数组
    dp = new Array(n).fill(0);
    // 遍历time数组
    for (let i = 0; i < n; i++) {
      // 如果是第一个元素，将dp[0]赋值为time[0]
      if (i === 0) {
        dp[0] = time[0];
        // 如果dp[0]大于limit，输出0 0并退出程序
        if (dp[0] > limit) {
          console.log('0 0');
          return ;
        }
      }
      // 如果是第二个元素，将dp[1]赋值为time[1]
      else if (i === 1) {
        dp[1] = time[1];
      }
      // 如果是后面的元素，根据状态转移方程计算dp[i]
      else {
        const t1 = dp[i - 1] + time[i] + time[0];
```

```
        const t2 = dp[i - 2] + time[0] + time[i] + time[1] + time[1];

        dp[i] = Math.min(t1, t2);
      }
      // 如果dp[i]大于limit，输出i和dp[i-1]并退出程序
      if (dp[i] > limit) {
        console.log(`${i} ${dp[i - 1]}`);
          return ;
      }
    }
    // 输出n和dp[n-1]
    console.log(`${n} ${dp[n - 1]}`);
          return ;
  }
});
```

C++

Plain Text

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int n, limit;
    // 输入n和限制时间
    cin >> n >> limit;
    // 定义一个长度为n的数组time
    int time[n];
    // 输入n个时间值
    for (int i = 0; i < n; i++) {
        cin >> time[i];
    }
    // 对时间数组进行排序
    sort(time, time + n);
    // 定义一个长度为n的数组dp
    int dp[n];
    // 遍历时间数组
    for (int i = 0; i < n; i++) {
        // 如果是第一个时间值
        if (i == 0) {
            // dp[0]等于第一个时间值
            dp[0] = time[0];
            // 如果dp[0]大于限制时间，输出0 0并结束程序
            if (dp[0] > limit) {
                cout << "0 0" << endl;
                return 0;
            }
        } else if (i == 1) {
            // 如果是第二个时间值，dp[1]等于第二个时间值
            dp[1] = time[1];
        } else {
            // 如果是第三个及以后的时间值
            // t1表示只有一个士兵划船的时间，t2表示有两个士兵划船的时间
            int t1 = dp[i - 1] + time[i] + time[0]; // 1个士兵划船
            int t2 = dp[i - 2] + time[0] + time[i] + time[1] + time[1]; // 2个士兵划船
            // dp[i]等于t1和t2中的最小值
            dp[i] = min(t1, t2);
        }
        // 如果dp[i]大于限制时间，输出i和dp[i-1]并结束程序
        if (dp[i] > limit) {
            cout << i << " " << dp[i - 1] << endl;
            return 0;
```

```
            }
        }
        // 输出n和dp[n-1]
        cout << n << " " << dp[n - 1] << endl;
        return 0;
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>

int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

int main() {
    // 读取输入的n和limit
    int n, limit;
    scanf("%d %d", &n, &limit);

    // 创建长度为n的整型数组time
    int *time = (int *)malloc(n * sizeof(int));

    // 循环读取n个时间并存入数组time中
    for (int i = 0; i < n; i++) {
        scanf("%d", &time[i]);
    }

    // 对数组time进行排序
    qsort(time, n, sizeof(int), compare);

    // 创建长度为n的整型数组dp
    int *dp = (int *)malloc(n * sizeof(int));

    // 循环计算dp数组的值
    for (int i = 0; i < n; i++) {
        if (i == 0) {
            // 如果i等于0，将dp[0]赋值为time[0]
            dp[0] = time[0];
            // 如果dp[0]大于limit，输出"0 0"并结束程序
            if (dp[0] > limit) {
                printf("0 0\n");
                free(time);
                free(dp);
                return 0;
            }
        } else if (i == 1) {
            // 如果i等于1，将dp[1]赋值为time[1]
            dp[1] = time[1];
        } else {
            // 如果i大于1，计算t1和t2的值并将较小值赋给dp[i]
            int t1 = dp[i - 1] + time[i] + time[0];
            int t2 = dp[i - 2] + time[0] + time[i] + time[1] + time[1];
```

```
                dp[i] = (t1 < t2) ? t1 : t2;
            }
            // 如果dp[i]大于limit，输出i和dp[i-1]的值并结束程序
            if (dp[i] > limit) {
                printf("%d %d\n", i, dp[i - 1]);
                free(time);
                free(dp);
                return 0;
            }
        }
        // 输出n和dp[n-1]的值
        printf("%d %d\n", n, dp[n - 1]);

        // 释放内存
        free(time);
        free(dp);

        return 0;
    }
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
    - 备注
- 输出描述
- 示例1
- 示例2
- 示例3
- 解题思路
    - 2. 处理特定情况
    - 3. 排序
    - 4. 动态规划策略
- Java
- Python
- JavaScript
- C++
- C语言

> 来自：华为OD机考2025B卷 - 士兵过河（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 启动多任务排序（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看

题目描述

一个应用启动时，会有多个初始化任务需要执行，并且任务之间有依赖关系，例如A任务依赖B任务，那么必须在B任务执行完成之后，才能开始执行A任务。
现在给出多条任务依赖关系的规则，请输入任务的顺序执行序列，规则采用贪婪策略，即一个任务如果没有依赖的任务，则立刻开始执行，如果同时有多个任务要执行，则根据任务名称字母顺序排序。
例如：B任务依赖A任务，C任务依赖A任务，D任务依赖B任务和C任务，同时，D任务还依赖E任务。那么执行任务的顺序由先到后是：

```
A任务，E任务，B任务，C任务，D任务
```

这里A和E任务都是没有依赖的，立即执行。

输入描述

输入参数每个元素都表示任意两个任务之间的依赖关系，输入参数中符号"->"表示依赖方向，例如：

> A->B：表示A依赖B

多个依赖之间用单个空格分隔

输出描述

输出排序后的启动任务列表，多个任务之间用单个空格分隔

用例

输入

```
A->B C->B
```

输出

```
B A C
```

解题思路

1. 构建图的邻接表和计算每个任务的入度。邻接表 `graph` 用于表示任务间的依赖关系，而入度表 `inDegree` 用于记录每个任务依赖的任务数量。
2. 遍历输入的依赖关系，更新邻接表和入度表。对于每个依赖 `A->B` ，将 `B` 添加到 `A` 的邻接列表中，并将 `B` 的入度加1。同时，确保每个任务至少在入度表中出现一次，如果任务没有依赖，则其入度为0。
3. 使用一个队列（或列表）来收集入度为0的任务，即可以立即执行的任务。
4. 按照字母顺序执行入度为0的任务，并在执行过程中更新其它任务的入度。对于每个执行的任务，检查它的所有后继任务，将每个后继任务的入度减1。如果后继任务的入度减为0，则将其加入到下一轮执行的队列中。
5. 重复步骤4，直到没有任务可以执行。

用例 `A->B C->B` 模拟计算过程：

- 输入： `A->B C->B`
- 分割输入，得到依赖关系： `A` 依赖于 `B` ， `C` 依赖于 `B` 。
- 构建邻接表和入度表：
  - 邻接表： `B -> [A, C]` （ `B` 完成后，可以执行 `A` 和 `C` ）
  - 入度表： `A: 1, C: 1, B: 0` （ `A` 和 `C` 各有1个依赖， `B` 没有依赖）
- 由于 `B` 的入度为0，它可以立即执行。执行 `B` ，更新队列。
- 执行 `B` 后，更新 `A` 和 `C` 的入度，它们的入度都变为0，可以执行。将 `A` 和 `C` 加入执行队列。
- 根据字母顺序，先执行 `A` ，然后执行 `C` 。
- 最终输出的执行顺序为： `B A C` 。

Plain Text

```
#include <iostream>
#include <string>
#include <sstream>
#include <map>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;
// 使用map和queue实现拓扑排序

string findOrder(const vector<string>& dependencies) {
    map<string, vector<string>> graph; // 任务图，映射每个任务到它依赖的任务列表
    map<string, int> inDegree; // 存储每个任务的入度（依赖它的任务数量）

    // 构建图和计算入度
    for (const auto& dep : dependencies) {
        auto pos = dep.find("->");
        string from = dep.substr(pos + 2), to = dep.substr(0, pos);
        graph[from].push_back(to); // 添加边
        inDegree[to]++;
        if (inDegree.find(from) == inDegree.end()) inDegree[from] = 0; // 确保每个任务都在入度表中
    }

    // 使用queue替代ArrayList存储可以立即执行的任务（入度为0）
    queue<string> queue;
    for (const auto& pair : inDegree) {
        if (pair.second == 0) queue.push(pair.first);
    }

    vector<string> order; // 存储任务执行顺序
    while (!queue.empty()) {
        string current = queue.front();
        queue.pop();
        order.push_back(current);

        // 遍历当前任务的所有后继任务，并更新它们的入度
        for (const auto& next : graph[current]) {
            if (--inDegree[next] == 0) {
                queue.push(next);
            }
        }
    }

```

```
    // 使用ostringstream组装返回的字符串
    ostringstream joinedOrder;
    for (const auto& task : order) {
        if (&task != &order[0]) joinedOrder << " ";
        joinedOrder << task;
    }

    return joinedOrder.str();
}

int main() {
    string input;
    getline(cin, input); // 读取一行输入
    istringstream iss(input);
    vector<string> dependencies;
    string dep;

    // 根据空格分割字符串
    while (getline(iss, dep, ' ')) {
        dependencies.push_back(dep);
    }

    // 打印任务执行的顺序
    cout << findOrder(dependencies) << endl;

    return 0;
}
```

Java

Java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取一行输入并按空格分割，每个部分代表一个依赖关系
        String input = scanner.nextLine();
        String[] dependencies = input.split("\\s+"); // 根据空格分割输入字符串
        System.out.println(findOrder(dependencies)); // 打印任务执行的顺序
    }

    private static String findOrder(String[] dependencies) {
        // 用于存储图的邻接表表示，键是任务，值是依赖于键的任务列表
        Map<String, List<String>> graph = new HashMap<>();
        // 存储每个任务的入度（依赖于它的其他任务的数量）
        Map<String, Integer> inDegree = new HashMap<>();
        for (String dep : dependencies) {
            // 分割每个依赖关系为两部分，parts[1] -> parts[0]
            String[] parts = dep.split("->");
            String from = parts[1], to = parts[0]; // 注意：这里与通常的表示相反，表示to依赖于from
            // 如果from不存在于图中，则添加from并初始化其任务列表
            graph.putIfAbsent(from, new ArrayList<>());
            // 在from的任务列表中添加to，表示from完成后，可以进行to
            graph.get(from).add(to);
            // 增加to的入度
            inDegree.put(to, inDegree.getOrDefault(to, 0) + 1);
            // 确保每个任务都在入度表中，如果没有其他任务依赖于它，则入度为0
            inDegree.putIfAbsent(from, 0);
        }

        // 使用ArrayList代替优先队列，以便在每次循环中排序
        ArrayList<String> queue = new ArrayList<>();
        for (String task : inDegree.keySet()) {
            // 收集入度为0的任务，即可以立即执行的任务
            if (inDegree.get(task) == 0) {
                queue.add(task);
            }
        }

        // 使用StringJoiner构建最终的任务执行顺序字符串
        StringJoiner order = new StringJoiner(" ");
        while (!queue.isEmpty()) {
```

```java
            // 准备新队列，用于存储下一轮入度变为0的任务
            ArrayList<String> newQueue = new ArrayList<>();

            for (String current : queue) {
                // 添加当前任务到执行顺序中
                order.add(current);
                // 遍历当前任务的所有后继任务
                for (String next : graph.getOrDefault(current, new ArrayList<>())) {
                    // 减少后继任务的入度
                    inDegree.put(next, inDegree.get(next) - 1);
                    // 如果后继任务的入度为0，则添加到新队列中
                    if (inDegree.get(next) == 0) {
                        newQueue.add(next);
                    }
                }
            }

            // 更新队列为新队列，并对其排序以保证字母顺序
            queue = newQueue;
            Collections.sort(queue);
        }

        // 返回构建的任务执行顺序字符串
        return order.toString();
    }
}
```

javaScript

Plain Text

```
const readline = require('readline');

// 创建 readline 接口实例
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});


rl.on("line", (input) => {
  // 根据空格分割输入字符串，得到依赖关系数组
  const dependencies = input.split(/\s+/);
  console.log(findOrder(dependencies)); // 打印任务执行的顺序
  rl.close(); // 关闭 readline 接口
});

function findOrder(dependencies) {
  // 用于存储图的邻接表表示，键是任务，值是依赖于该任务的任务列表
  const graph = {};
  // 存储每个任务的入度（依赖于它的其他任务的数量）
  const inDegree = {};

  dependencies.forEach(dep => {
    // 分割每个依赖关系为两部分，parts[1] -> parts[0]
    const parts = dep.split('->');
    const from = parts[1], to = parts[0]; // 注意：这里与通常的表示相反，表示 to 依赖于 from
    // 如果 from 不存在于图中，则添加 from 并初始化其任务列表
    if (!graph[from]) graph[from] = [];
    // 在 from 的任务列表中添加 to，表示 from 完成后，可以进行 to
    graph[from].push(to);
    // 增加 to 的入度
    inDegree[to] = (inDegree[to] || 0) + 1;
    // 确保每个任务都在入度表中，如果没有其他任务依赖于它，则入度为 0
    if (inDegree[from] === undefined) inDegree[from] = 0;
  });

  // 收集入度为 0 的任务，即可以立即执行的任务
  let queue = Object.keys(inDegree).filter(task => inDegree[task] === 0);

  let order = []; // 存储最终的任务执行顺序

  while (queue.length > 0) {
    // 对当前可执行的任务按字典序排序
    queue.sort();
```

```
    const newQueue = [];

    queue.forEach(current => {
      order.push(current);
      // 遍历当前任务的所有后继任务，减少它们的入度
      (graph[current] || []).forEach(next => {
        inDegree[next] -= 1;
        // 如果后继任务的入度为 0，则加入新队列
        if (inDegree[next] === 0) {
          newQueue.push(next);
        }
      });
    });

    // 更新队列为新队列
    queue = newQueue;
  }

  // 返回构建的任务执行顺序字符串
  return order.join(' ');
}
```

Python

Plain Text

```
def find_order(dependencies):
    """
    根据依赖关系计算任务执行的顺序。

    参数:
    dependencies - 依赖关系列表，每个元素形式为 "A->B" 表示 A 依赖于 B。

    返回:
    任务执行的顺序，以空格分隔的字符串形式。
    """
    # 用于存储图的邻接表表示，键是任务，值是依赖于键的任务列表
    graph = {}
    # 存储每个任务的入度（依赖于它的其他任务的数量）
    in_degree = {}

    # 构建图和计算入度
    for dep in dependencies:
        # 分割每个依赖关系为两部分
        parts = dep.split("->")
        from_task, to_task = parts[1], parts[0]  # 注意：这里与通常的表示相反
        # 如果 from_task 不存在于图中，则添加 from_task 并初始化其任务列表
        if from_task not in graph:
            graph[from_task] = []
        # 在 from_task 的任务列表中添加 to_task
        graph[from_task].append(to_task)
        # 增加 to_task 的入度
        in_degree[to_task] = in_degree.get(to_task, 0) + 1
        # 确保每个任务都在入度表中，如果没有其他任务依赖于它，则入度为 0
        in_degree[from_task] = in_degree.get(from_task, 0)

    # 收集入度为 0 的任务，即可以立即执行的任务
    queue = [task for task, degree in in_degree.items() if degree == 0]

    order = []  # 存储最终的任务执行顺序

    while queue:
        # 对当前可执行的任务按字典序排序
        queue.sort()
        new_queue = []
        for current in queue:
            order.append(current)
            # 遍历当前任务的所有后继任务，减少它们的入度
            for next_task in graph.get(current, []):
                in_degree[next_task] -= 1
                # 如果后继任务的入度为 0，则加入新队列
```

```
                    if in_degree[next_task] == 0:
                        new_queue.append(next_task)
            queue = new_queue

    # 返回构建的任务执行顺序字符串
    return " ".join(order)

# 主函数，读取输入并处理
if __name__ == "__main__":
    input_str = input()
    dependencies = input_str.split()
    print(find_order(dependencies))
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_TASKS 100
#define MAX_DEPENDENCIES 200
#define MAX_LEN 10

// 任务结构体，包括任务名称和依赖此任务的后继任务列表
typedef struct {
    char name[MAX_LEN];
    int inDegree; // 入度，即依赖此任务的其他任务数量
    int nextTasks[MAX_DEPENDENCIES]; // 存储后继任务的索引
    int nextCount; // 后继任务的数量
} Task;

Task tasks[MAX_TASKS]; // 所有任务的数组
int taskCount = 0; // 任务总数

// 查找或添加任务到任务数组中，并返回任务的索引
int findOrAddTask(char* name) {
    for (int i = 0; i < taskCount; i++) {
        if (strcmp(tasks[i].name, name) == 0) return i;
    }
    strcpy(tasks[taskCount].name, name);
    tasks[taskCount].inDegree = 0;
    tasks[taskCount].nextCount = 0;
    return taskCount++;
}

// 添加依赖关系
void addDependency(char* from, char* to) {
    int fromIndex = findOrAddTask(from);
    int toIndex = findOrAddTask(to);
    tasks[fromIndex].nextTasks[tasks[fromIndex].nextCount++] = toIndex;
    tasks[toIndex].inDegree++;
}

// 执行拓扑排序，并打印任务执行顺序
void findOrder() {
    int queue[MAX_TASKS], start = 0, end = 0;

    // 找出所有入度为0的任务
    for (int i = 0; i < taskCount; i++) {
        if (tasks[i].inDegree == 0) {
```

```
            queue[end++] = i;
        }
    }

    // 拓扑排序主循环
    while (start < end) {
        int current = queue[start++];
        printf("%s ", tasks[current].name);

        // 遍历当前任务的所有后继任务
        for (int i = 0; i < tasks[current].nextCount; i++) {
            int next = tasks[current].nextTasks[i];
            // 减少后继任务的入度，并在入度变为0时加入队列
            if (--tasks[next].inDegree == 0) {
                queue[end++] = next;
            }
        }
    }
}

int main() {
    char input[1000];
    fgets(input, sizeof(input), stdin); // 读取一行输入

    char* token = strtok(input, " ");
    while (token != NULL) {
        char from[MAX_LEN], to[MAX_LEN];
        sscanf(token, "%[^-]->%s", to, from); // 分割每个依赖关系
        addDependency(from, to); // 添加依赖关系
        token = strtok(NULL, " ");
    }

    findOrder(); // 执行拓扑排序并打印结果

    return 0;
}
```

完整用例

用例1

Plain Text

```
A->B B->C C->D D->E
```

用例2

```
A->B
```

用例3

```
A->B B->C C->D
```

用例4

```
A->B C->B D->A D->C
```

用例5

```
A->B C->D E->F
```

用例6

```
A->B B->C C->D D->E E->F F->G
```

用例7

```
B->A C->B D->C E->C F->A
```

用例8

```
B->A C->A E->A C->D E->D F->D
```

用例9

```
B->A B->D D->C A->D
```

用例10

```
P->Q P->R Q->S R->S S->T
```

> 来自: 华为OD机考2025B卷 – 启动多任务排序（Java & Python& JS & C++ & C ）–CSDN博客

华为OD机考2025B卷 - 恢复数字序列（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

对于一个连续正整数组成的序列，可以将其拼接成一个字符串，再将字符串里的部分字符打乱顺序。如序列8 9 10 11 12，拼接成的字符串为89101112，打乱一部分字符后得到90811211，原来的正整数10就被拆成了0和1。
现给定一个按如上规则得到的打乱字符的字符串，请将其还原成连续正整数序列，并输出序列中最小的数字。

输入描述

输入一行，为打乱字符的字符串和正整数序列的长度，两者间用空格分隔，字符串长度不超过200，正整数不超过1000，保证输入可以还原成唯一序列。

输出描述

输出一个数字，为序列中最小的数字。

示例1

输入

```
19801211 5
```

输出

```
8
```

说明

示例2

输入

```
432111111111 4
```

输出

```
111
```

说明

解题思路

题目要求从一个打乱了部分字符的字符串中还原出一个连续的正整数序列，找出其中最小的数字。

题目流程和解释：

1. **输入内容**：
   - **字符串部分**：表示将某个连续正整数序列拼接后、打乱部分字符得到的字符串。
   - **序列长度部分**：表示原来的连续正整数序列中有多少个数字。
2. **任务**：
   - 根据打乱后的字符串和给定的序列长度，找出原来连续正整数序列的最小数字。
   - **注意**：输入字符串是由原来的一些连续正整数组成的，但某些数字可能被拆分成了单个字符，且这些字符被打乱后可能不再连贯。因此，需要将这些打乱的字符重新组合成连续的正整数。
3. **关键点**：
   - **字符打乱**：原本连续的数字可能在字符串中被拆分并打乱顺序，因此需要将它们重新组合成完整的数字。
   - **唯一还原性**：题目保证每一个输入都能还原成唯一的连续正整数序列，这意味着不存在多个可能的解。

示例解析

示例 1：

```
输入：19801211 5
输出：8
```

解释：

- 打乱后的字符串是 `19801211`，序列长度是 `5`。
- 还原成的连续正整数序列为 `8 9 10 11 12`，最小的数字是 `8`。

示例 2：

```
输入：432111111111 4
输出：111
```

解释：

- 打乱后的字符串是 `432111111111`，序列长度是 `4`。
- 还原成的连续正整数序列为 `111 112 113 114`，最小的数字是 `111`。

总结

题目的核心是通过排列组合字符找到能组成连续正整数序列的方案，并返回其中最小的数字。 统计打乱字符的字符串中各字符的数量。然后遍历所有可能的连续整数序列，对于每个序列，检查序列中的字符数量是否与打乱字符的字符串中各字符数量一致。如果找到一个匹配的序列，输出序列中最小的数字并结束循环。

Java

```java
import java.util.HashMap;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // 创建一个Scanner对象，用于读取输入
        Scanner sc = new Scanner(System.in);

        // 读取输入的打乱字符的字符串
        String s = sc.next();
        // 读取输入的正整数序列的长度
        int k = sc.nextInt();

        // 创建一个HashMap，用于统计打乱字符的字符串中各字符的数量
        HashMap<Character, Integer> base = new HashMap<>();
        // 遍历打乱字符的字符串
        for (int i = 0; i < s.length(); i++) {
            // 获取字符串中的字符
            char c = s.charAt(i);
            // 将字符及其数量存入HashMap
            base.put(c, base.getOrDefault(c, 0) + 1);
        }

        // 初始化滑动窗口的起始位置
        int i = 1;
        // 当滑动窗口的起始位置小于等于1000减去序列长度加1时，继续循环
        while (i <= 1000 - k + 1) {
            // 创建一个HashMap，用于计算滑动窗口内各字符的数量
            HashMap<Character, Integer> count = new HashMap<>();
            // 遍历滑动窗口内的正整数
            for (int j = i; j < i + k; j++) {
                // 将正整数转换为字符串
                String num = String.valueOf(j);
                // 遍历正整数字符串中的字符
                for (int m = 0; m < num.length(); m++) {
                    // 获取正整数字符串中的字符
                    char c = num.charAt(m);
                    // 将字符及其数量存入HashMap
                    count.put(c, count.getOrDefault(c, 0) + 1);
                }
            }

            // 初始化一个布尔变量，用于判断滑动窗口内各字符数量是否与打乱字符的字符串中各字符数量一致
            boolean isMatch = true;
```

```java
            // 遍历打乱字符的字符串中的字符
            for (Character c : base.keySet()) {
                // 如果滑动窗口内的字符数量与打乱字符的字符串中的字符数量不一致，将isMatch设为false并跳出循环
                if (!count.containsKey(c) || count.get(c) - base.get(c) != 0) {
                    isMatch = false;
                    break;
                }
            }

            // 如果滑动窗口内各字符数量与打乱字符的字符串中各字符数量一致，则输出滑动窗口的起始位置并返回
            if (isMatch) {
                System.out.println(i);
                return;
            }

            // 更新滑动窗口的起始位置
            i++;
        }
    }
}
```

Python

Plain Text

```
import sys
from collections import defaultdict

# 读取输入的打乱字符的字符串和正整数序列的长度
s, k = input().strip().split()
k = int(k)

# 创建一个字典，用于统计打乱字符的字符串中各字符的数量
base = defaultdict(int)
for c in s:
    base[c] += 1

# 初始化滑动窗口的起始位置
i = 1
# 当滑动窗口的起始位置小于等于1000减去序列长度加1时，继续循环
while i <= 1000 - k + 1:
    # 创建一个字典，用于计算滑动窗口内各字符的数量
    count = defaultdict(int)
    # 遍历滑动窗口内的正整数
    for j in range(i, i + k):
        # 将正整数转换为字符串
        num = str(j)
        # 遍历正整数字符串中的字符
        for c in num:
            # 将字符及其数量存入字典
            count[c] += 1

    # 初始化一个布尔变量，用于判断滑动窗口内各字符数量是否与打乱字符的字符串中各字符数量一致
    is_match = True
    # 遍历打乱字符的字符串中的字符
    for c in base:
        # 如果滑动窗口内的字符数量与打乱字符的字符串中的字符数量不一致，将is_match设为False并跳出循环
        if count[c] != base[c]:
            is_match = False
            break

    # 如果滑动窗口内各字符数量与打乱字符的字符串中各字符数量一致，则输出滑动窗口的起始位置并退出循环
    if is_match:
        print(i)
        break

    # 更新滑动窗口的起始位置
```

```
        i += 1
```

JavaScript

Plain Text

```
const readline = require('readline');

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

rl.on('line', (input) => {
  // 输入打乱字符的字符串和正整数序列的长度
  const s = input.split(' ')[0];
  const k = parseInt(input.split(' ')[1]);

  // 创建一个Map，用于统计打乱字符的字符串中各字符的数量
  const base = new Map();
  for (const c of s) {
    base.set(c, (base.get(c) || 0) + 1);
  }

  // 初始化滑动窗口的起始位置
  let i = 1;
  // 当滑动窗口的起始位置小于等于1000减去序列长度加1时，继续循环
  while (i <= 1000 - k + 1) {
    // 创建一个Map，用于计算滑动窗口内各字符的数量
    const count = new Map();
    // 遍历滑动窗口内的正整数
    for (let j = i; j < i + k; j++) {
      // 将正整数转换为字符串
      const num = String(j);
      // 遍历正整数字符串中的字符
      for (const c of num) {
        // 将字符及其数量存入Map
        count.set(c, (count.get(c) || 0) + 1);
      }
    }

    // 初始化一个布尔变量，用于判断滑动窗口内各字符数量是否与打乱字符的字符串中各字符数量一致
    let isMatch = true;
    // 遍历打乱字符的字符串中的字符
    for (const c of base.keys()) {
      // 如果滑动窗口内的字符数量与打乱字符的字符串中的字符数量不一致，将isMatch设为false并跳出循环
      if (!count.has(c) || count.get(c) - base.get(c) !== 0) {
        isMatch = false;
        break;
```

```
      }
    }

    // 如果滑动窗口内各字符数量与打乱字符的字符串中各字符数量一致，则输出滑动窗口的起始位置并返回
    if (isMatch) {
      console.log(i);
      return;
    }

    // 更新滑动窗口的起始位置
    i++;
  }
});
```

C++

Plain Text

```
#include <iostream>
#include <string>
#include <unordered_map>

int main() {
    // 创建一个字符串变量，用于读取输入的打乱字符的字符串
    std::string s;
    // 创建一个整数变量，用于读取输入的正整数序列的长度
    int k;

    // 读取输入
    std::cin >> s >> k;

    // 创建一个unordered_map，用于统计打乱字符的字符串中各字符的数量
    std::unordered_map<char, int> base;
    // 遍历打乱字符的字符串
    for (char c : s) {
        // 将字符及其数量存入unordered_map
        base[c]++;
    }

    // 初始化滑动窗口的起始位置
    int i = 1;
    // 当滑动窗口的起始位置小于等于1000减去序列长度加1时，继续循环
    while (i <= 1000 - k + 1) {
        // 创建一个unordered_map，用于计算滑动窗口内各字符的数量
        std::unordered_map<char, int> count;
        // 遍历滑动窗口内的正整数
        for (int j = i; j < i + k; j++) {
            // 将正整数转换为字符串
            std::string num = std::to_string(j);
            // 遍历正整数字符串中的字符
            for (char c : num) {
                // 将字符及其数量存入unordered_map
                count[c]++;
            }
        }

        // 初始化一个布尔变量，用于判断滑动窗口内各字符数量是否与打乱字符的字符串中各字符数量一致
        bool isMatch = true;
        // 遍历打乱字符的字符串中的字符
        for (const auto& p : base) {
            char c = p.first;

```

```
            // 如果滑动窗口内的字符数量与打乱字符的字符串中的字符数量不一致，将isMatch设为false并跳出循环
            if (count[c] != base[c]) {
                isMatch = false;
                break;
            }
        }

        // 如果滑动窗口内各字符数量与打乱字符的字符串中各字符数量一致，则输出滑动窗口的起始位置并返回
        if (isMatch) {
            std::cout << i << std::endl;
            return 0;
        }

        // 更新滑动窗口的起始位置
        i++;
    }

    return 0;
}
```

C语言

Plain Text

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

int main() {
    char s[201]; // 输入的打乱字符的字符串，长度不超过200
    int k;       // 正整数序列的长度

    // 读取输入
    scanf("%s %d", s, &k);

    int base[10] = {0}; // 用于统计打乱字符的字符串中各数字字符的数量

    // 遍历打乱字符的字符串，统计各字符的数量
    for (int i = 0; i < strlen(s); i++) {
        base[s[i] - '0']++; // 将字符数字转为整型进行统计
    }

    // 初始化滑动窗口的起始位置
    int i = 1;

    // 当滑动窗口的起始位置小于等于1000减去序列长度加1时，继续循环
    while (i <= 1000 - k + 1) {
        int count[10] = {0}; // 用于计算滑动窗口内各字符的数量

        // 遍历滑动窗口内的正整数
        for (int j = i; j < i + k; j++) {
            char num[6];
            sprintf(num, "%d", j); // 将正整数转换为字符串

            // 遍历正整数字符串中的字符，统计各字符数量
            for (int x = 0; x < strlen(num); x++) {
                count[num[x] - '0']++; // 将字符数字转为整型进行统计
            }
        }

        // 判断滑动窗口内各字符数量是否与打乱字符的字符串中各字符数量一致
        bool isMatch = true;
        for (int c = 0; c < 10; c++) {
            if (count[c] != base[c]) {
                isMatch = false;
                break;
            }
        }

```

```
        // 如果滑动窗口内各字符数量一致，输出滑动窗口的起始位置并退出程序
        if (isMatch) {
            printf("%d\n", i);
            return 0;
        }

        // 更新滑动窗口的起始位置
        i++;
    }

    return 0;
}
```

完整用例

用例1

Plain Text

```
19801211 5
```

用例2

Plain Text

```
432111111111 4
```

用例3

Plain Text

```
123 3
```

用例4

Plain Text

```
2431 4
```

用例5

```
910111213 5
```

用例6

```
123456666666 6
```

用例7

```
65777877 4
```

用例8

```
87901 4
```

用例9

```
98011121 5
```

用例10

```
1410111213 5
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1

- 示例2
- 解题思路
    - 题目流程和解释：
    - 示例解析
    - 总结
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
    - 用例1
    - 用例2
    - 用例3
    - 用例4
    - 用例5
    - 用例6
    - 用例7
    - 用例8
    - 用例9
    - 用例10

> 来自：华为OD机考2025B卷 - 恢复数字序列（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 查找单入口空闲区域（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

给定一个 m x n 的矩阵，由若干字符‘X’和‘O’构成，’X’表示该处已被占据，’O’表示该处空闲，请找到最大的单入口空闲区域。
解释：
空闲区域是由连通的’O’组成的区域，位于边界的’O’可以构成入口，
单入口空闲区域即有且只有一个位于边界的’O’作为入口的由连通的’O’组成的区域。
如果两个元素在水平或垂直方向相邻，则称它们是“连通”的。

输入描述

第一行输入为两个数字，第一个数字为行数m，第二个数字为列数n，两个数字以空格分隔，
1<=m,n<=200。
剩余各行为矩阵各行元素，元素为‘X’或‘O’，各元素间以空格分隔。

输出描述

若有唯一符合要求的最大单入口空闲区域，输出三个数字

- 第一个数字为入口行坐标（0~m-1）
- 第二个数字为入口列坐标（0~n-1）
- 第三个数字为区域大小

三个数字以空格分隔；
若有多个符合要求，则输出区域大小最大的，若多个符合要求的单入口区域的区域大小相同，则此时只需要输出区域大小，不需要输出入口坐标。
若没有，输出NULL。

示例1

输入

```
4 4
X X X X
X 0 0 X
X 0 0 X
X 0 X X
```

输出

```
3 1 5
```

说明

> 存在最大单入口区域，入口坐标(3,1)，区域大小5

示例2

输入

```
4 5
X X X X X
0 0 0 0 X
X 0 0 0 X
X 0 X X 0
```

输出

```
3 4 1
```

说明

> 存在最大单入口区域，入口坐标（3,4），区域大小1

示例3

输入

```
5 4
X X X X
X 0 0 0
X 0 0 0
X 0 0 X
X X X X
```

输出

```
NULL
```

说明

> NULL

示例4

输入

```
5 4
X X X X
X 0 0 0
X X X X
X 0 0 0
X X X X
```

输出

```
3
```

说明

> 存在两个大小为3的最大单入口区域，两个入口坐标分别为(1,3)、(3,3)

解题思路

题目解析

本题的目标是找到一个矩阵中由字符 `'0'` 连通组成的最大的 **单入口空闲区域**，并根据具体情况输出不同的信息。

核心概念

1. **空闲区域**：
   - 由 `'0'` 连通组成的区域。
   - 连通：水平或垂直方向相邻的 `'0'` 视为连通。
2. **入口**：
   - 位于矩阵边界的 `'0'` 元素可作为入口。
3. **单入口空闲区域**：
   - 一个空闲区域中仅有 **一个** `'0'` 位于边界。
   - 例如，若一个空闲区域有多个边界上的 `'0'`，则不满足单入口的要求。
4. **最大区域**：
   - 单入口区域中，所有符合要求的区域里空闲格子最多的区域。

输入描述

1. 矩阵大小：第一行输入两个数字 `m` 和 `n`，表示矩阵行列数。
2. 矩阵内容：
   - 其后每行输入矩阵的一个横行，元素为 `'X'` 或 `'0'`，用空格分隔。

输出描述

1. 若存在唯一最大单入口空闲区域：
   - 输出三个数字：入口行坐标、入口列坐标、区域大小。
2. 若存在多个符合要求的最大区域：
   - 输出区域大小，不需要输出入口坐标。
3. 若没有任何符合条件的单入口区域：
   - 输出 `NULL`。

案例解析

示例 1：

输入：

```
4 4
X X X X
X 0 0 X
X 0 0 X
X 0 X X
```

分析：

1. 入口位置候选：边界的 `'O'` 分别位于 `(3,1)`。
2. 区域连通性：
   - 入口 `(3,1)` 通过连通性包含5个 `'O'`，即 `(3,1)`，`(2,1)`，`(1,1)`，`(2,2)`，`(1,2)`。
3. 结果：区域大小为 5，唯一入口 `(3,1)`。

输出：

```
3 1 5
```

示例 2：

输入：

```
4 5
X X X X X
0 0 0 0 X
X 0 0 0 X
X 0 X X 0
```

分析：

1. 入口位置候选：边界的 `'O'` 分别位于 `(0,1)`，`(3,4)`。
2. 区域连通性：
   - `(0,1)` 等区域中包含多个入口，不满足单入口要求。
   - `(3,4)` 的区域大小为 1，只有一个入口 `(3,4)`。
3. 结果：唯一区域大小为 1，入口 `(3,4)`。

输出：

```
3 4 1
```

示例 3：

输入：

```
5 4
X X X X
X 0 0 0
X 0 0 0
X 0 0 X
X X X X
```

分析：

1. 入口位置候选：边界的 `'0'` 分别位于 `(1,3)`，`(3,3)`。
2. 区域连通性：
    - 区域中有多个入口，不符合单入口要求。
3. 结果：无符合条件的单入口区域。

输出：

```
NULL
```

示例 4：

输入：

```
5 4
X X X X
X 0 0 0
X X X X
X 0 0 0
X X X X
```

分析：

1. 存在两个大小为 3 的单入口空闲区域：
   - 区域 1：入口 `(1,3)`，区域大小 3。
   - 区域 2：入口 `(3,3)`，区域大小 3。
2. 因为多个区域大小相同：
   - 输出区域大小即可。

输出：

```
3
```

解题思路

1. **边界检查**：
   - 遍历矩阵边界，找到所有可能的入口点。
2. **深度优先搜索（DFS）或广度优先搜索（BFS）**：
   - 以每个入口为起点，查找其连通的所有 `'0'`，统计区域大小。
   - 同时检查是否有多个入口连接到该区域。
3. **单入口验证**：
   - 如果区域包含超过一个边界上的 `'0'`，则不满足单入口要求。
4. **结果判定**：
   - 记录所有单入口区域的大小和入口位置。
   - 若存在多个最大区域，输出区域大小；否则输出唯一入口坐标和大小。

关键点

- **边界检测**：区分边界 `'0'` 和内部 `'0'`。
- **区域连通性**：使用 DFS 或 BFS 遍历区域。

- **多入口验证**：区域中是否包含多个边界上的 `'0'` 。
- **结果优先级**：唯一最大区域的坐标和大小、多区域输出大小、无区域输出 `NULL` 。

Java

```java
import java.util.*;

public class Main {
    static int rows, cols; // 矩阵的行数和列数
    static String[][] matrix = new String[200][200]; // 存储矩阵
    static int[][] offset = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}}; // 上下左右四个方向的偏移量
    static boolean[][] checked = new boolean[200][200]; // 存储已访问的位置
    static int maxArea = 0; // 最大单入口区域大小
    static int[] maxEntry = null; // 最大区域的入口

    // DFS搜索连通块
    public static int dfs(int i, int j, int[] entry, int count) {
        // 如果位置非法或已经访问或不是空闲区域，则退出
        if (i < 0 || i >= rows || j < 0 || j >= cols || checked[i][j] || matrix[i][j].equals("X")) {
            return count;
        }

        checked[i][j] = true; // 标记该位置已访问
        count++; // 连通块大小加1

        // 如果是边界上的'O'，记录为可能的入口
        if ((i == 0 || i == rows - 1 || j == 0 || j == cols - 1)) {
            entry[0]++; // 增加入口计数
            entry[1] = i; // 记录入口的行号
            entry[2] = j; // 记录入口的列号
        }

        // 在上下左右四个方向继续搜索
        for (int[] dir : offset) {
            count = dfs(i + dir[0], j + dir[1], entry, count);
        }

        return count;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        rows = scanner.nextInt();
        cols = scanner.nextInt();

        // 读入矩阵
        for (int i = 0; i < rows; i++) {
```

```java
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = scanner.next();
            }
        }

        int maxSize = 0; // 最大区域大小
        int[] maxRegionEntry = null; // 最大区域的入口坐标

        // 遍历矩阵，对每个未访问的空闲位置进行DFS
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (matrix[i][j].equals("O") && !checked[i][j]) {
                    int[] entry = new int[3]; // 用于记录入口：entry[0]=入口数，entry[1]=行，entry[2]=列
                    int area = dfs(i, j, entry, 0); // 计算区域大小

                    // 如果是单入口区域，更新最大值
                    if (entry[0] == 1) {
                        if (area > maxSize) {
                            maxSize = area;
                            maxRegionEntry = new int[]{entry[1], entry[2], area};
                        } else if (area == maxSize) {
                            maxRegionEntry = null; // 存在多个最大区域
                        }
                    }
                }
            }
        }

        // 输出结果
        if (maxSize == 0) { // 没有单入口区域
            System.out.println("NULL");
        } else if (maxRegionEntry != null) { // 唯一的最大单入口区域
            System.out.println(maxRegionEntry[0] + " " + maxRegionEntry[1] + " " + maxRegionEntry[2]);
        } else { // 存在多个最大单入口区域
            System.out.println(maxSize);
        }
    }
}
```

Python

Plain Text

```python
# 导入所需模块
import sys
from typing import List

# 全局变量
rows, cols = 0, 0  # 矩阵的行数和列数
matrix = []  # 存储矩阵
checked = []  # 存储访问过的位置
offset = [(0, -1), (0, 1), (-1, 0), (1, 0)]  # 上下左右四个方向的偏移量


# 深度优先搜索（DFS）函数
def dfs(i: int, j: int, entry: List[int], count: int) -> int:
    # 如果越界或已访问或当前位置是'X'，直接返回
    if i < 0 or i >= rows or j < 0 or j >= cols or checked[i][j] or matrix[i][j] == "X":
        return count

    checked[i][j] = True  # 标记当前位置为已访问
    count += 1  # 区域大小增加

    # 如果当前位置在边界，可能是一个入口
    if i == 0 or i == rows - 1 or j == 0 or j == cols - 1:
        entry[0] += 1  # 入口计数加1
        entry[1], entry[2] = i, j  # 记录入口位置

    # 遍历上下左右四个方向
    for dx, dy in offset:
        count = dfs(i + dx, j + dy, entry, count)

    return count


def main():
    global rows, cols, matrix, checked

    # 从标准输入读取矩阵大小
    rows, cols = map(int, input().split())
    matrix = [input().split() for _ in range(rows)]  # 读取矩阵
    checked = [[False] * cols for _ in range(rows)]  # 初始化访问数组

    maxSize = 0  # 最大区域大小
    maxRegionEntry = None  # 最大区域的入口坐标

    # 遍历矩阵，寻找未访问的'O'
```

```
    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == "O" and not checked[i][j]:
                entry = [0, -1, -1]  # 初始化入口数据：[入口数, 行, 列]
                area = dfs(i, j, entry, 0)  # 计算连通区域大小

                # 如果是单入口区域，更新最大区域信息
                if entry[0] == 1:
                    if area > maxSize:
                        maxSize = area
                        maxRegionEntry = [entry[1], entry[2], area]
                    elif area == maxSize:
                        maxRegionEntry = None  # 如果有多个相同大小的最大区域

    # 输出结果
    if maxSize == 0:  # 没有单入口区域
        print("NULL")
    elif maxRegionEntry:  # 唯一最大单入口区域
        print(" ".join(map(str, maxRegionEntry)))
    else:  # 有多个相同大小的最大区域
        print(maxSize)


if __name__ == "__main__":
    main()
```

JavaScript

Plain Text

```
const readline = require("readline");

// 全局变量
let rows = 0, cols = 0; // 矩阵的行数和列数
let matrix = []; // 存储矩阵
let checked = []; // 存储访问过的位置
const offset = [[0, -1], [0, 1], [-1, 0], [1, 0]]; // 上下左右四个方向的偏移量

// 深度优先搜索（DFS）函数
function dfs(i, j, entry, count) {
    // 如果越界、已访问或当前位置是'X'，直接返回
    if (i < 0 || i >= rows || j < 0 || j >= cols || checked[i][j] || matrix[i][j] === "X") {
        return count;
    }

    checked[i][j] = true; // 标记当前位置为已访问
    count++; // 区域大小增加

    // 如果当前位置在边界，可能是一个入口
    if (i === 0 || i === rows - 1 || j === 0 || j === cols - 1) {
        entry[0]++; // 入口计数加1
        entry[1] = i; // 记录入口行号
        entry[2] = j; // 记录入口列号
    }

    // 遍历上下左右四个方向
    for (const [dx, dy] of offset) {
        count = dfs(i + dx, j + dy, entry, count);
    }

    return count;
}

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

let input = []; // 存储输入
rl.on("line", (line) => {
    input.push(line);
}).on("close", () => {
    [rows, cols] = input[0].split(" ").map(Number);
    matrix = input.slice(1).map(line => line.split(" "));
```

```
    checked = Array.from({ length: rows }, () => Array(cols).fill(false)); // 初始化访问数组

    let maxSize = 0; // 最大区域大小
    let maxRegionEntry = null; // 最大区域入口

    // 遍历矩阵
    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            if (matrix[i][j] === "O" && !checked[i][j]) {
                let entry = [0, -1, -1]; // 初始化入口数据
                const area = dfs(i, j, entry, 0); // 计算区域大小

                // 如果是单入口区域，更新最大值
                if (entry[0] === 1) {
                    if (area > maxSize) {
                        maxSize = area;
                        maxRegionEntry = [entry[1], entry[2], area];
                    } else if (area === maxSize) {
                        maxRegionEntry = null; // 有多个最大区域
                    }
                }
            }
        }
    }

    // 输出结果
    if (maxSize === 0) {
        console.log("NULL");
    } else if (maxRegionEntry) {
        console.log(maxRegionEntry.join(" "));
    } else {
        console.log(maxSize);
    }
});
```

C++

Plain Text

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

// 全局变量
int rows, cols;                      // 矩阵的行数和列数
vector<vector<string>> matrix;       // 存储矩阵
vector<vector<bool>> checked;        // 存储访问过的位置
int offset[4][2] = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}}; // 上下左右四个方向的偏移量

// 深度优先搜索（DFS）函数
int dfs(int i, int j, vector<int> &entry, int count) {
    // 如果越界、已访问或当前位置是'X'，直接返回
    if (i < 0 || i >= rows || j < 0 || j >= cols || checked[i][j] || matrix[i][j] == "X") {
        return count;
    }

    checked[i][j] = true; // 标记当前位置为已访问
    count++; // 区域大小增加

    // 如果当前位置在边界，可能是一个入口
    if (i == 0 || i == rows - 1 || j == 0 || j == cols - 1) {
        entry[0]++; // 入口计数加1
        entry[1] = i; // 记录入口行号
        entry[2] = j; // 记录入口列号
    }

    // 遍历上下左右四个方向
    for (auto &dir : offset) {
        count = dfs(i + dir[0], j + dir[1], entry, count);
    }

    return count;
}

int main() {
    cin >> rows >> cols;

    matrix.resize(rows, vector<string>(cols)); // 初始化矩阵
    checked.resize(rows, vector<bool>(cols, false)); // 初始化访问数组

```

```
    // 输入矩阵
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> matrix[i][j];
        }
    }

    int maxSize = 0; // 最大区域大小
    vector<int> maxRegionEntry; // 最大区域的入口

    // 遍历矩阵
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (matrix[i][j] == "O" && !checked[i][j]) {
                vector<int> entry(3, 0); // 初始化入口数据：[入口数, 行, 列]
                int area = dfs(i, j, entry, 0); // 计算区域大小

                // 如果是单入口区域，更新最大值
                if (entry[0] == 1) {
                    if (area > maxSize) {
                        maxSize = area;
                        maxRegionEntry = {entry[1], entry[2], area};
                    } else if (area == maxSize) {
                        maxRegionEntry.clear(); // 有多个最大区域
                    }
                }
            }
        }
    }

    // 输出结果
    if (maxSize == 0) {
        cout << "NULL" << endl;
    } else if (!maxRegionEntry.empty()) {
        cout << maxRegionEntry[0] << " " << maxRegionEntry[1] << " " << maxRegionEntry[2] << endl;
    } else {
        cout << maxSize << endl;
    }

    return 0;
}
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

int rows, cols; // 矩阵的行数和列数
char matrix[200][200]; // 存储矩阵
bool checked[200][200]; // 存储已访问的位置
int offset[4][2] = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}}; // 上下左右四个方向的偏移量
int maxArea = 0; // 最大单入口区域大小

// DFS搜索连通块
int dfs(int i, int j, int entry[3], int count) {
    // 如果位置非法或已经访问或不是空闲区域，则退出
    if (i < 0 || i >= rows || j < 0 || j >= cols || checked[i][j] || matrix[i][j] == 'X') {
        return count;
    }

    checked[i][j] = true; // 标记该位置已访问
    count++; // 连通块大小加1

    // 如果是边界上的'O'，记录为可能的入口
    if ((i == 0 || i == rows - 1 || j == 0 || j == cols - 1)) {
        entry[0]++; // 增加入口计数
        entry[1] = i; // 记录入口的行号
        entry[2] = j; // 记录入口的列号
    }

    // 在上下左右四个方向继续搜索
    for (int k = 0; k < 4; k++) {
        count = dfs(i + offset[k][0], j + offset[k][1], entry, count);
    }

    return count;
}

int main() {
    scanf("%d %d", &rows, &cols);

    // 读入矩阵
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf(" %c", &matrix[i][j]);
```

```
        }
    }

    // 初始化checked数组
    memset(checked, false, sizeof(checked));

    int maxSize = 0; // 最大区域大小
    int maxRegionEntry[3] = {-1, -1, -1}; // 最大区域的入口坐标
    bool multipleMax = false; // 是否存在多个最大区域

    // 遍历矩阵，对每个未访问的空闲位置进行DFS
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (matrix[i][j] == 'O' && !checked[i][j]) {
                int entry[3] = {0, 0, 0}; // 用于记录入口：entry[0]=入口数, entry[1]=行, entry[2]=列
                int area = dfs(i, j, entry, 0); // 计算区域大小

                // 如果是单入口区域，更新最大值
                if (entry[0] == 1) {
                    if (area > maxSize) {
                        maxSize = area;
                        maxRegionEntry[0] = entry[1];
                        maxRegionEntry[1] = entry[2];
                        maxRegionEntry[2] = area;
                        multipleMax = false;
                    } else if (area == maxSize) {
                        multipleMax = true; // 存在多个最大区域
                    }
                }
            }
        }
    }

    // 输出结果
    if (maxSize == 0) { // 没有单入口区域
        printf("NULL\n");
    } else if (!multipleMax) { // 唯一的最大单入口区域
        printf("%d %d %d\n", maxRegionEntry[0], maxRegionEntry[1], maxRegionEntry[2]);
    } else { // 存在多个最大单入口区域
        printf("%d\n", maxSize);
    }

    return 0;
}
```

> 来自: 华为OD机考2025B卷 - 查找单入口空闲区域（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 食堂供餐（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

某公司员工食堂以盒饭方式供餐。
为将员工取餐排队时间降低为0，食堂的供餐速度必须要足够快。
现在需要根据以往员工取餐的统计信息，计算出一个刚好能达成排队时间为0的最低供餐速度。即，食堂在每个单位时间内必须至少做出多少价盒饭才能满足要求。

输入描述

第1行为一个正整数N，表示食堂开餐时长。

- 1 ≤ N ≤ 1000

第2行为一个正整数M，表示开餐前食堂已经准备好的盒饭份数。

- P1 ≤ M ≤ 1000

第3行为N个正整数，用空格分隔，依次表示开餐时间内按时间顺序每个单位时间进入食堂取餐的人数Pi。

- 1 ≤ i ≤ N
- 0 ≤ Pi ≤ 100

备注

- 每人只取一份盒饭。
- 需要满足排队时间为0，必须保证取餐员工到达食堂时，食堂库存盒饭数量不少于本次来取餐的人数。
- 第一个单位时间来取餐的员工只能取开餐前食堂准备好的盒饭。
- 每个单位时间里制作的盒饭只能供应给后续单位时间来的取餐的员工。
- 食堂在每个单位时间里制作的盒饭数量是相同的。

输出描述

一个整数，能满足题目要求的最低供餐速度（每个单位时间需要做出多少份盒饭）。

示例1

输入

```
3
14
10 4 5
```

输出

```
3
```

说明

> 本样例中，总共有3批员工就餐，每批人数分别为10、4、5。
> 开餐前食堂库存14份。
> 食堂每个单位时间至少要做出3份餐饭才能达成排队时间为0的目标。具体情况如下:
> 第一个单位时间来的10位员工直接从库存取餐。取餐后库存剩余4份盒饭，加上第一个单位时间做出的3份，库存有7份
> 第二个单位时间来的4员工从库存的7份中取4份。取餐后库存剩余3份盒饭，加上第二个单位时间做出的3份，库存有6份。
> 第三个单位时间来的员工从库存的6份中取5份，库存足够。
> 如果食堂在单位时间只能做出2份餐饭，则情况如下:
> 第一个单位时间来的10位员工直接从库存取餐。取餐后库存剩余4份盒饭，加上第一个单位时间做出的2份，库存有6份
> 第二个单位时间来的4员工从库存的6份中取4份。取餐后库存剩余2份盒饭，加上第二个单位时间做出的2份，库存有4份.
> 第三个单位时间来的员工需要取5份，但库存只有4份，库存不够。

解题思路

1. **问题理解**：题目要求我们计算出一个最低的供餐速度，使得员工在取餐时不会因为盒饭不足而排队。我们需要关注的是每个时间单位进入取餐的员工人数和食堂当前的盒饭库存。
2. **输入参数**：
   - `N`：开餐的单位时间（总共多少个时间段）。
   - `M`：开餐前食堂准备好的盒饭数量。
   - `P`：一个数组，表示在每个单位时间内到达的员工人数。
3. **核心要求**：在每个时间段，食堂的库存（包括开餐前的库存和每个时间段生产的盒饭）必须不小于员工的需求。如果在某个时间段无法满足需求，则当前的供餐速度太低。
4. **二分查找**：
   - 我们可以利用二分查找来高效地找出满足条件的最低供餐速度。首先确定搜索范围：速度的下界是0，上界可以设置为员工到达量的最大值。

- 在每次查找中，计算一个中间速度，并验证该速度是否能够满足所有时间段的需求。

Java

```java
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // 食堂开餐时长
        int n = sc.nextInt();
        // 开餐前食堂已经准备好的盒饭份数
        int m = sc.nextInt();
        // 开餐时间内按时间顺序每个单位时间进入食堂取餐的人数
        int[] pi = new int[n];
        int min = 0;
        int max = 0;

        // 读取每个单位时间进入食堂取餐的人数，并找到最大值
        for (int i = 0; i < n; i++) {
            pi[i] = sc.nextInt();
            if (pi[i] > max) {
                max = pi[i];
            }
        }

        // 初始时，最低供餐速度为0
        int ans = 0;

        // 二分查找最低供餐速度
        while (min <= max) {
            int mid = (min + max) >> 1;
            if (calculate_minimum_speed(m, mid, pi)) {
                ans = mid;
                max = mid - 1;
            } else {
                min = mid + 1;
            }
        }

        // 输出能满足题目要求的最低供餐速度
        System.out.println(ans);
    }

    public static boolean calculate_minimum_speed(int m, int add, int[] pi) {
        // 开餐前食堂已经准备好的盒饭份数
        m -= pi[0];
```

```java

        // 遍历每个单位时间
        for (int i = 1; i < pi.length; i++) {
            // 计算每个单位时间需要制作的盒饭数量
            int make = Math.max(0, pi[i] - m);
            // 更新食堂库存盒饭数量
            m = Math.max(0, m - pi[i] + add);
            // 判断是否能满足要求
            if (make > add) {
                return false;
            }
        }

        return true;
    }
}
```

Python

Plain Text

```
def calculate_minimum_speed(m, add, pi):
    # 开餐前食堂已经准备好的盒饭份数
    m -= pi[0]

    # 遍历每个单位时间
    for i in range(1, len(pi)):
        # 计算每个单位时间需要制作的盒饭数量
        make = max(0, pi[i] - m)
        # 更新食堂库存盒饭数量
        m = max(0, m - pi[i] + add)
        # 判断是否能满足要求
        if make > add:
            return False

    return True

n = int(input()) # 食堂开餐时长
m = int(input()) # 开餐前食堂已经准备好的盒饭份数
pi = list(map(int, input().split())) # 开餐时间内按时间顺序每个单位时间进入食堂取餐的人数

ans = 0 # 最低供餐速度
# 二分查找
min_num = 0
max_num = max(pi)
while min_num <= max_num:
    mid = (min_num + max_num) >> 1
    if calculate_minimum_speed(m, mid, pi):
        ans = mid
        max_num = mid - 1
    else:
        min_num = mid + 1

print(ans)
```

JavaScript

Plain Text

```
const readline = require('readline');

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

let n, m, pi;

rl.on('line', (line) => {
  if (!n) {
    n = parseInt(line);
  } else if (!m) {
    m = parseInt(line);
  } else {
    pi = line.split(' ').map(num => parseInt(num));

    main(n, m, pi);
  }
});

function main(n, m, pi) {
  let min = 0;
  let max = 0;


  for (let i = 0; i < n; i++) {
    if (pi[i] > max) {
      max = pi[i];
    }
  }

  let ans = 0;


  while (min <= max) {
    let mid = (min + max) >> 1;
    if (calculate_minimum_speed(m, mid, pi)) {
      ans = mid;
      max = mid - 1;
    } else {
      min = mid + 1;
    }
  }

```

```
  console.log(ans);
}

function calculate_minimum_speed(m, add, pi) {
  m -= pi[0];

  for (let i = 1; i < pi.length; i++) {
    let make = Math.max(0, pi[i] - m);
    m = Math.max(0, m - pi[i] + add);
    if (make > add) {
      return false;
    }
  }

  return true;
}
```

C++

Plain Text

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool calculate_minimum_speed(int m, int add, vector<int>& pi) {
    m -= pi[0];

    for (int i = 1; i < pi.size(); i++) {
        int make = max(0, pi[i] - m);
        m = max(0, m - pi[i] + add);
        if (make > add) {
            return false;
        }
    }

    return true;
}

int main() {
    int n, m;
    cin >> n >> m;

    vector<int> pi(n);
    int min = 0;
    int max = 0;

    for (int i = 0; i < n; i++) {
        cin >> pi[i];
        if (pi[i] > max) {
            max = pi[i];
        }
    }

    int ans = 0;

    while (min <= max) {
        int mid = (min + max) >> 1;
        if (calculate_minimum_speed(m, mid, pi)) {
            ans = mid;
            max = mid - 1;
        } else {
            min = mid + 1;
        }
```

```
    }

    cout << ans << endl;

    return 0;
}
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// 计算最低供餐速度的函数
bool calculate_minimum_speed(int m, int add, int* pi, int length) {
    // 开餐前食堂已经准备好的盒饭份数
    m -= pi[0];

    // 遍历每个单位时间
    for (int i = 1; i < length; i++) {
        // 计算每个单位时间需要制作的盒饭数量
        int make = (pi[i] - m) > 0 ? (pi[i] - m) : 0;
        // 更新食堂库存盒饭数量
        m = (m - pi[i] + add) > 0 ? (m - pi[i] + add) : 0;
        // 判断是否能满足要求
        if (make > add) {
            return false;
        }
    }

    return true;
}

int main() {
    // 食堂开餐时长
    int n;
    // 开餐前食堂已经准备好的盒饭份数
    int m;

    scanf("%d %d", &n, &m);

    // 开餐时间内按时间顺序每个单位时间进入食堂取餐的人数
    int* pi = (int*)malloc(n * sizeof(int));
    int min = 0;
    int max = 0;

    // 读取每个单位时间进入食堂取餐的人数，并找到最大值
    for (int i = 0; i < n; i++) {
        scanf("%d", &pi[i]);
        if (pi[i] > max) {
            max = pi[i];
        }
    }

```

```
    // 初始时，最低供餐速度为0
    int ans = 0;

    // 二分查找最低供餐速度
    while (min <= max) {
        int mid = (min + max) >> 1;
        if (calculate_minimum_speed(m, mid, pi, n)) {
            ans = mid;
            max = mid - 1;
        } else {
            min = mid + 1;
        }
    }

    // 输出能满足题目要求的最低供餐速度
    printf("%d\n", ans);

    // 释放动态分配的内存
    free(pi);

    return 0;
}
```

完整用例

用例1

Plain Text

```
3
14
10 4 5
```

用例2

Plain Text

```
5
20
8 5 10 3 6
```

用例3

```
4
10
6 4 7 3
```

用例4

```
2
5
2 3
```

用例5

```
6
30
10 5 8 12 4 6
```

用例6

```
1
5
5
```

用例7

```
8
15
6 4 7 2 5 3 8 1
```

用例8

```
10
50
10 8 5 6 7 9 4 3 2 1
```

用例9

```
7
20
5 3 6 8 4 9 2
```

用例10

```
9
25
8 5 7 4 6 3 9 2 1
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 备注
- 输出描述
- 示例1
- 解题思路
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4

- 用例5
- 用例6
- 用例7
- 用例8
- 用例9
- 用例10

> 来自: 华为OD机考2025B卷 - 食堂供餐（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 构成正方形的数量（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

输入N个互不相同的二维整数坐标，求这N个坐标可以构成的正方形数量。[内积为零的的两个向量垂直]

输入描述

第一行输入为N，N代表坐标数量，N为正整数。N <= 100
之后的 K 行输入为坐标x y以空格分隔，x，y为整数，-10<=x, y<=10

输出描述

输出可以构成的正方形数量。

示例1

输入

```plain text
3
1 3
2 4
3 1
```

输出

```plain text
0
```

说明

> （3个点不足以构成正方形）

示例2

输入

```
4
0 0
1 2
3 1
2 -1
```

输出

```
1
```

说明

内积的定义

内积（Dot Product）是向量代数中的一个重要概念。给定两个向量 a = (a 1 , a 2) \mathbf{a} = (a_1, a_2) a=(a1,a2) 和 b = (b 1 , b 2) \mathbf{b} = (b_1, b_2) b=(b1,b2)，它们的内积定义为：
a · b = a 1 × b 1 + a 2 × b 2 \mathbf{a} \cdot \mathbf{b} = a_1 \times b_1 + a_2 \times b_2 a·b=a1×b1+a2×b2
如果两个向量的内积为零，即 a · b = 0 \mathbf{a} \cdot \mathbf{b} = 0 a·b=0，那么这两个向量互相垂直（即成90度角）。

解题思路

1. **遍历所有点对，寻找可能的正方形**：
   - 对于每一对点 (A , B) (A, B) (A,B)，我们假设这两个点为正方形的对角线的两个端点。
   - 通过计算向量 A B → \overrightarrow{AB} AB 的垂直向量，可以找到另外两个点的坐标。
   - 检查这两个点是否都在集合中，如果在，则构成一个正方形。
2. **避免重复计算**：
   - 在统计正方形时，每个正方形会被计算四次（每个顶点都可能作为起始点），所以最终的正方形数量应该除以4。

用例解释

输入

```plain text
4
0 0
1 2
3 1
2 -1
```

坐标点

- A (0, 0)
- B (1, 2)
- C (3, 1)
- D (2, -1)

计算过程

1. **遍历点对**：
   - 遍历所有的点对，检查是否能构成正方形。
   - 共有 6 对点组合：`(A, B)`，`(A, C)`，`(A, D)`，`(B, C)`，`(B, D)`，`(C, D)`。
2. **判断正方形**：
   - 以点对 `(A, B)` 为例：
     - 计算可能的两个正方形对角点：
       1. `(x3, y3) = (-2, 1)` 和 `(x4, y4) = (3, 1)`。
       2. `(x5, y5) = (-2, -1)` 和 `(x6, y6) = (-1, 1)`。
     - 检查这些点是否存在于输入的坐标集合中：
       - `(x3, y3)` 不存在，但 `(x4, y4)` 即 C 存在。
       - `(x5, y5)` 即 D 存在，但 `(x6, y6)` 不存在。
     - 因此，通过 `(A, B)` 这一对，找到了一个正方形 ABCD。
3. **结果**：
   - 通过遍历所有点对，只找到一个正方形 ABCD，最终正方形数量为 1。

Java

```java
import java.util.ArrayList;
import java.util.Scanner;



public class Main {
    static class Point {
        int x, y;

        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        // 判断两个点是否相等
        boolean equals(Point p) {
            return this.x == p.x && this.y == p.y;
        }
    }
    // 检查点是否存在于点列表中
    static boolean pointExists(ArrayList<Point> points, Point p) {
        for (Point point : points) {
            if (point.equals(p)) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        ArrayList<Point> points = new ArrayList<>();

        // 读取所有点的坐标
        for (int i = 0; i < n; i++) {
            int x = scanner.nextInt();
            int y = scanner.nextInt();
            points.add(new Point(x, y));
        }

        int squareCount = 0;

        // 遍历所有点对，检查是否能构成正方形
        for (int i = 0; i < n; i++) {
```

```java
            Point p1 = points.get(i);

            for (int j = i + 1; j < n; j++) {
                Point p2 = points.get(j);

                // 计算两个可能的对角点
                Point p3 = new Point(p1.x - (p1.y - p2.y), p1.y + (p1.x - p2.x));
                Point p4 = new Point(p2.x - (p1.y - p2.y), p2.y + (p1.x - p2.x));

                if (pointExists(points, p3) && pointExists(points, p4)) {
                    squareCount++;
                }

                // 计算另外两个可能的对角点
                Point p5 = new Point(p1.x + (p1.y - p2.y), p1.y - (p1.x - p2.x));
                Point p6 = new Point(p2.x + (p1.y - p2.y), p2.y - (p1.x - p2.x));

                if (pointExists(points, p5) && pointExists(points, p6)) {
                    squareCount++;
                }
            }
        }

        // 每个正方形被计算了4次，因此结果需要除以4
        System.out.println(squareCount / 4);

        scanner.close();
    }
}
// 代码2
import java.util.Scanner;
import java.util.Vector;
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        // 创建Scanner对象，用于从控制台读取输入
        Scanner scanner = new Scanner(System.in);

        // 读取第一个输入的整数，表示坐标数量
        int n = scanner.nextInt();
        // 读取整行输入并忽略换行符
        scanner.nextLine();
```

```java
        // 创建一个Vector来存储坐标点的字符串形式
        Vector<String> coordinates = new Vector<>(n);
        for (int i = 0; i < n; i++) {
            // 逐行读取坐标点，并存储到Vector中
            coordinates.add(scanner.nextLine());
        }

        // 初始化正方形计数器
        int squareCount = 0;
        // 使用HashSet存储坐标点，便于快速查找
        HashSet<String> set = new HashSet<>(coordinates);

        // 遍历所有坐标点对
        for (int i = 0; i < n; i++) {
            // 将第一个坐标点分割为x1和y1
            String[] coordinate1 = coordinates.get(i).split(" ");
            int x1 = Integer.parseInt(coordinate1[0]);
            int y1 = Integer.parseInt(coordinate1[1]);

            for (int j = i + 1; j < n; j++) {
                // 将第二个坐标点分割为x2和y2
                String[] coordinate2 = coordinates.get(j).split(" ");
                int x2 = Integer.parseInt(coordinate2[0]);
                int y2 = Integer.parseInt(coordinate2[1]);

                // 计算两个可能的对角点
                int x3 = x1 - (y1 - y2), y3 = y1 + (x1 - x2);
                int x4 = x2 - (y1 - y2), y4 = y2 + (x1 - x2);

                // 检查这两个对角点是否存在于坐标集合中
                if (set.contains(x3 + " " + y3) && set.contains(x4 + " " + y4)) {
                    squareCount++;
                }

                // 计算另外两个可能的对角点
                int x5 = x1 + (y1 - y2), y5 = y1 - (x1 - x2);
                int x6 = x2 + (y1 - y2), y6 = y2 - (x1 - x2);

                // 检查这两个对角点是否存在于坐标集合中
                if (set.contains(x5 + " " + y5) && set.contains(x6 + " " + y6)) {
                    squareCount++;
                }
            }
        }

        // 每个正方形被计算了4次，因此结果需要除以4
```

```
        System.out.println(squareCount / 4);

        // 关闭Scanner对象，释放资源
        scanner.close();
    }
}
```

Python

Plain Text

```
# 定义一个函数来判断两个点是否相等
def are_points_equal(p1, p2):
    return p1[0] == p2[0] and p1[1] == p2[1]

# 定义一个函数来检查一个点是否存在于点列表中
def point_exists(points, p):
    for point in points:
        if are_points_equal(point, p):
            return True
    return False

# 读取坐标数量
n = int(input())
coordinates = []

# 读取坐标并存入列表
for _ in range(n):
    x, y = map(int, input().split())
    coordinates.append((x, y))

square_count = 0

# 遍历所有点对，检查是否能构成正方形
for i in range(n):
    x1, y1 = coordinates[i]

    for j in range(i + 1, n):
        x2, y2 = coordinates[j]

        # 计算两个可能的对角点
        x3, y3 = x1 - (y1 - y2), y1 + (x1 - x2)
        x4, y4 = x2 - (y1 - y2), y2 + (x1 - x2)
        p3 = (x3, y3)
        p4 = (x4, y4)

        if point_exists(coordinates, p3) and point_exists(coordinates, p4):
            square_count += 1

        # 计算另外两个可能的对角点
        x5, y5 = x1 + (y1 - y2), y1 - (x1 - x2)
        x6, y6 = x2 + (y1 - y2), y2 - (x1 - x2)
        p5 = (x5, y5)
        p6 = (x6, y6)

```

```
        if point_exists(coordinates, p5) and point_exists(coordinates, p6):
            square_count += 1

# 每个正方形被计算了4次，因此结果需要除以4
print(square_count // 4)




# 代码2

import sys

# 读取第一个输入的整数，表示坐标数量
n = int(input())

# 存储坐标的字符串形式的列表
coordinates = []
for i in range(n):
    # 读取坐标点的字符串，并去掉两端的空白字符
    coordinates.append(input().strip())

# 初始化正方形计数器
squareCount = 0
# 使用集合存储坐标点，便于快速查找
coordinate_set = set(coordinates)

# 遍历所有坐标点对
for i in range(n):
    # 将第一个坐标点分割为x1和y1
    x1, y1 = map(int, coordinates[i].split())

    for j in range(i + 1, n):
        # 将第二个坐标点分割为x2和y2
        x2, y2 = map(int, coordinates[j].split())

        # 计算两个可能的对角点
        x3, y3 = x1 - (y1 - y2), y1 + (x1 - x2)
        x4, y4 = x2 - (y1 - y2), y2 + (x1 - x2)

        # 检查这两个对角点是否存在于坐标集合中
        if f"{x3} {y3}" in coordinate_set and f"{x4} {y4}" in coordinate_set:
            squareCount += 1

        # 计算另外两个可能的对角点
        x5, y5 = x1 + (y1 - y2), y1 - (x1 - x2)
        x6, y6 = x2 + (y1 - y2), y2 - (x1 - x2)
```

```

        # 检查这两个对角点是否存在于坐标集合中
        if f"{x5} {y5}" in coordinate_set and f"{x6} {y6}" in coordinate_set:
            squareCount += 1

# 每个正方形被计算了4次，因此结果需要除以4
print(squareCount // 4)
```

JavaScript

Plain Text

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

let n;
let coordinates = [];

// 判断两个点是否相等
function arePointsEqual(p1, p2) {
  return p1[0] === p2[0] && p1[1] === p2[1];
}

// 检查一个点是否存在于点数组中
function pointExists(points, p) {
  for (let point of points) {
    if (arePointsEqual(point, p)) {
      return true;
    }
  }
  return false;
}

rl.on('line', (line) => {
  if (n === undefined) {
    n = parseInt(line);
  } else {
    coordinates.push(line.split(' ').map(Number));
    if (coordinates.length === n) rl.close();
  }
});

rl.on('close', () => {
  let squareCount = 0;

  // 遍历所有点对，检查是否能构成正方形
  for (let i = 0; i < n; i++) {
    let [x1, y1] = coordinates[i];

    for (let j = i + 1; j < n; j++) {
      let [x2, y2] = coordinates[j];

      // 计算两个可能的对角点
      let x3 = x1 - (y1 - y2), y3 = y1 + (x1 - x2);
```

```
      let x4 = x2 - (y1 - y2), y4 = y2 + (x1 - x2);

      if (pointExists(coordinates, [x3, y3]) && pointExists(coordinates, [x4, y4])) {
        squareCount++;
      }

      // 计算另外两个可能的对角点
      let x5 = x1 + (y1 - y2), y5 = y1 - (x1 - x2);
      let x6 = x2 + (y1 - y2), y6 = y2 - (x1 - x2);

      if (pointExists(coordinates, [x5, y5]) && pointExists(coordinates, [x6, y6])) {
        squareCount++;
      }
    }
  }

  // 每个正方形被计算了4次，因此结果需要除以4
  console.log(Math.floor(squareCount / 4));
});

// 代码2
const readline = require('readline');

// 创建接口读取标准输入输出
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

let n; // 用于存储坐标数量
let coordinates = []; // 存储坐标的数组

// 监听每行输入
rl.on('line', (line) => {
  if (!n) {
    // 第一行输入是坐标数量
    n = parseInt(line);
    return;
  }

  // 其余的行是坐标点
  coordinates.push(line);

  // 当输入的坐标数达到n时，关闭输入流
  if (coordinates.length === n) {
    rl.close();
```

```
  }
});

// 当输入完成时
rl.on('close', () => {
  let squareCount = 0; // 记录正方形数量
  let set = new Set(coordinates); // 用集合存储坐标点，便于快速查找

  // 遍历所有坐标点对
  for (let i = 0; i < n; i++) {
    // 将第一个坐标点分割为x1和y1
    let [x1, y1] = coordinates[i].split(' ').map(Number);

    for (let j = i + 1; j < n; j++) {
      // 将第二个坐标点分割为x2和y2
      let [x2, y2] = coordinates[j].split(' ').map(Number);

      // 计算两个可能的对角点
      let x3 = x1 - (y1 - y2), y3 = y1 + (x1 - x2);
      let x4 = x2 - (y1 - y2), y4 = y2 + (x1 - x2);

      // 检查这两个对角点是否存在于坐标集合中
      if (set.has(`${x3} ${y3}`) && set.has(`${x4} ${y4}`)) {
        squareCount++;
      }

      // 计算另外两个可能的对角点
      let x5 = x1 + (y1 - y2), y5 = y1 - (x1 - x2);
      let x6 = x2 + (y1 - y2), y6 = y2 - (x1 - x2);

      // 检查这两个对角点是否存在于坐标集合中
      if (set.has(`${x5} ${y5}`) && set.has(`${x6} ${y6}`)) {
        squareCount++;
      }
    }
  }

  // 每个正方形被计算了4次，因此结果需要除以4
  console.log(Math.floor(squareCount / 4));
});
```

C++

Plain Text

```
#include <iostream>
#include <vector>
using namespace std;

// 定义一个点结构体
struct Point {
    int x, y;
};

// 判断两个点是否相等
bool arePointsEqual(Point a, Point b) {
    return a.x == b.x && a.y == b.y;
}

// 检查数组中是否存在某点
bool pointExists(vector<Point>& points, Point p) {
    for (auto& point : points) {
        if (arePointsEqual(point, p)) {
            return true;
        }
    }
    return false;
}

int main() {
    int n;
    cin >> n;

    vector<Point> points(n);

    // 读取坐标并存入数组
    for (int i = 0; i < n; i++) {
        cin >> points[i].x >> points[i].y;
    }

    int squareCount = 0;

    // 遍历所有点对，检查是否能构成正方形
    for (int i = 0; i < n; i++) {
        int x1 = points[i].x;
        int y1 = points[i].y;

        for (int j = i + 1; j < n; j++) {
            int x2 = points[j].x;
            int y2 = points[j].y;
```

```
            // 计算两个可能的对角点
            Point p3 = {x1 - (y1 - y2), y1 + (x1 - x2)};
            Point p4 = {x2 - (y1 - y2), y2 + (x1 - x2)};

            if (pointExists(points, p3) && pointExists(points, p4)) {
                squareCount++;
            }

            // 计算另外两个可能的对角点
            Point p5 = {x1 + (y1 - y2), y1 - (x1 - x2)};
            Point p6 = {x2 + (y1 - y2), y2 - (x1 - x2)};

            if (pointExists(points, p5) && pointExists(points, p6)) {
                squareCount++;
            }
        }
    }

    // 每个正方形被计算了4次，因此结果需要除以4
    cout << squareCount / 4 << endl;

    return 0;
}

// 代码2
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;

int main() {
    int n;
    // 读取坐标数量
    cin >> n;
    // 忽略换行符
    cin.ignore();

    // 创建一个二维数组来存储坐标
    vector<vector<int>> coordinates(n, vector<int>(2));
    for (int i = 0; i < n; i++) {
        // 读取坐标并存储到数组中
        cin >> coordinates[i][0] >> coordinates[i][1];
    }

    int squareCount = 0; // 记录正方形数量
    unordered_set<string> set;
    // 将坐标转换为字符串形式并存入哈希集合
```

```
    for (int i = 0; i < n; i++) {
        set.insert(to_string(coordinates[i][0]) + " " + to_string(coordinates[i][1]));
    }

    // 遍历所有坐标点对
    for (int i = 0; i < n; i++) {
        int x1 = coordinates[i][0];
        int y1 = coordinates[i][1];

        for (int j = i + 1; j < n; j++) {
            int x2 = coordinates[j][0];
            int y2 = coordinates[j][1];

            // 计算两个可能的对角点
            int x3 = x1 - (y1 - y2), y3 = y1 + (x1 - x2);
            int x4 = x2 - (y1 - y2), y4 = y2 + (x1 - x2);

            // 检查这两个对角点是否存在于坐标集合中
            if (set.count(to_string(x3) + " " + to_string(y3)) && set.count(to_string(x4) + " " + to_string(y4))) {
                squareCount++;
            }

            // 计算另外两个可能的对角点
            int x5 = x1 + (y1 - y2), y5 = y1 - (x1 - x2);
            int x6 = x2 + (y1 - y2), y6 = y2 - (x1 - x2);

            // 检查这两个对角点是否存在于坐标集合中
            if (set.count(to_string(x5) + " " + to_string(y5)) && set.count(to_string(x6) + " " + to_string(y6))) {
                squareCount++;
            }
        }
    }

    // 每个正方形被计算了4次，因此结果需要除以4
    cout << squareCount / 4 << endl;

    return 0;
}
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    int x, y;
} Point;

// 判断两个点是否相等
int arePointsEqual(Point a, Point b) {
    return a.x == b.x && a.y == b.y;
}

// 检查数组中是否存在某点
int pointExists(Point points[], int n, Point p) {
    for (int i = 0; i < n; i++) {
        if (arePointsEqual(points[i], p)) {
            return 1;
        }
    }
    return 0;
}

int main() {
    int n;
    scanf("%d", &n);

    Point points[n];

    // 读取坐标并存入数组
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &points[i].x, &points[i].y);
    }

    int squareCount = 0;

    // 遍历所有点对，检查是否能构成正方形
    for (int i = 0; i < n; i++) {
        int x1 = points[i].x;
        int y1 = points[i].y;

        for (int j = i + 1; j < n; j++) {
            int x2 = points[j].x;
            int y2 = points[j].y;

```

```
                // 计算两个可能的对角点
                Point p3 = {x1 - (y1 - y2), y1 + (x1 - x2)};
                Point p4 = {x2 - (y1 - y2), y2 + (x1 - x2)};

                if (pointExists(points, n, p3) && pointExists(points, n, p4)) {
                    squareCount++;
                }

                // 计算另外两个可能的对角点
                Point p5 = {x1 + (y1 - y2), y1 - (x1 - x2)};
                Point p6 = {x2 + (y1 - y2), y2 - (x1 - x2)};

                if (pointExists(points, n, p5) && pointExists(points, n, p6)) {
                    squareCount++;
                }
            }
        }

        // 每个正方形被计算了4次，因此结果需要除以4
        printf("%d\n", squareCount / 4);

        return 0;
    }
```

完整用例

用例1

Plain Text

```
3
1 1
2 2
3 3
```

用例2

```
4
0 0
1 1
1 0
0 1
```

用例3

```
5
-1 0
0 1
1 0
0 -1
1 2
```

用例4

```
4
0 0
1 0
1 1
0 1
```

用例5

```
6
0 0
1 0
1 1
0 1
2 0
2 1
```

用例6

```
8
0 0
1 1
2 2
3 3
0 1
1 2
2 3
3 0
```

用例7

```
10
-1 0
0 1
1 0
0 -1
1 2
2 1
0 2
2 0
-2 0
0 -2
```

用例8

```
6
0 0
1 1
2 2
3 3
4 4
5 5
```

用例9

```
7
0 0
1 1
2 2
3 3
0 1
1 2
2 3
```

用例10

```
9
-1 0
0 1
1 0
0 -1
1 2
2 1
0 2
2 0
-2 0
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- 示例2
- 内积的定义
- 解题思路
  - 用例解释
  - 输入
  - 坐标点
  - 计算过程
- Java
- Python
- JavaScript

- C++
- C语言
- 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4
  - 用例5
  - 用例6
  - 用例7
  - 用例8
  - 用例9
  - 用例10

> 来自: 华为OD机考2025B卷 – 构成正方形的数量（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 核酸检测（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

为了达到新冠疫情精准防控的需要，为了避免全员核酸检测带来的浪费，需要精准圈定可能被感染的人群。
现在根据传染病流调以及大数据分析，得到了每个人之间在时间、空间上是否存在轨迹的交叉。
现在给定一组确诊人员编号 (X1,X2,X3,...Xn)在所有人当中，找出哪些人需要进行核酸检测，输出需要进行核酸检测的数。
(注意:确诊病例自身不需要再做核酸检测)需要进行核酸检测的人，是病毒传播链条上的所有人员，即有可能通过确诊病例所能传播到的所有人。
例如:A是确诊病例，A和B有接触、B和C有接触 C和D有接触，D和E有接触。那么B、C、D、E都是需要进行核酸检测的人

输入描述

第一行为总人数N
第二行为确证病例人员编号（确证病例人员数量<N），用逗号隔开
接下来N行，每一行有N个数字，用逗号隔开，其中第i行的第j个数字表名编号i是否与编号j接触过。0表示没有接触，1表示有接触
备注：
人员编号从0开始
0 < N < 100 0<N<1000<N<100

输出描述

输出需要做核酸检测的人数

用例

输入

```
5
1,2
1,1,0,1,0
1,1,0,0,0
0,0,1,0,1
1,0,0,1,0
0,0,1,0,1
```

输出

```
3
```

说明

> 编号为1、2号的人员为确诊病例
> 1号与0号有接触，0号与3号有接触，2号与4号有接触。所以，需要做核酸检测的人是0号、3号、4号,总计3人要进行核酸检测。

解题思路

1. 初始化一个大小为N的布尔数组 `visited`，用来记录每个人是否已经被访问过（即是否已经确定需要进行核酸检测）。
2. 初始化一个大小为N×N的布尔矩阵 `contacts`，用来表示人与人之间的接触情况。如果 `contacts[i][j]` 为 `true`，则表示编号为i的人与编号为j的人有接触。
3. 从输入中读取确诊病例的编号，并对每个确诊病例执行深度优先搜索（DFS）：
   - 在DFS中，首先将当前节点（即当前人员编号）标记为已访问。
   - 然后遍历该节点的所有邻接节点（即与当前人员有接触的所有人），如果邻接节点未被访问，则递归地对邻接节点执行DFS。
4. 完成DFS后，遍历 `visited` 数组，统计除确诊病例外的已访问节点的数量，即为需要进行核酸检测的人数。

用例模拟计算过程：

1. 初始化 `visited` 数组为 `[false, false, false, false, false]`。
2. 构建 `contacts` 矩阵如下：

```Plain Text
[true,  true,  false, true,  false]
[true,  true,  false, false, false]
[false, false, true,  false, true ]
[true,  false, false, true,  false]
[false, false, true,  false, true ]
```

3. 对于确诊病例1和2，执行DFS：
   - DFS(1): 标记 `visited[1]` 为 `true` ，检查与1有接触的人，发现0和3，递归DFS(0)和DFS(3)。
     - DFS(0): 标记 `visited[0]` 为 `true` ，检查与0有接触的人，发现3，但3已在DFS(1)中被访问，所以不再递归。
     - DFS(3): 标记 `visited[3]` 为 `true` ，检查与3有接触的人，发现0，但0已在DFS(0)中被访问，所以不再递归。
   - DFS(2): 标记 `visited[2]` 为 `true` ，检查与2有接触的人，发现4，递归DFS(4)。
     - DFS(4): 标记 `visited[4]` 为 `true` ，检查与4有接触的人，发现2，但2已在DFS(2)中被访问，所以不再递归。
4. DFS执行完毕后， `visited` 数组为 `[true, true, true, true, true]` 。
5. 统计除确诊病例外的已访问节点数量，即 `visited` 中为 `true` 的元素数量减去确诊病例的数量。在这个用例中，所有人都被访问过，但需要排除确诊病例1和2，所以需要进行核酸检测的人数为5 − 2 = 3。

Plain Text

```
#include <iostream>
#include <vector>
#include <sstream>
#include <string>
#include <set>

using namespace std;
// 深度优先搜索（DFS）算法
void dfs(const vector<vector<bool>>& contacts, vector<bool>& visited, int start) {
    visited[start] = true; // 标记当前节点为已访问
    for (size_t i = 0; i < contacts.size(); ++i) {
        // 如果当前节点与其他节点有接触，并且该节点未被访问过
        if (contacts[start][i] && !visited[i]) {
            dfs(contacts, visited, i); // 递归访问该节点
        }
    }
}

int main() {
    int N;
    cin >> N;
    cin.ignore(); // 忽略换行符

    string line;
    getline(cin, line);
    istringstream iss(line);
    string caseIndex;
    set<int> confirmedCases;

    // 读取确诊病例人员编号
    while (getline(iss, caseIndex, ',')) {
        confirmedCases.insert(stoi(caseIndex));
    }

    vector<vector<bool>> contacts(N, vector<bool>(N));
    vector<bool> visited(N, false);

    // 构建接触矩阵
    for (int i = 0; i < N; ++i) {
        getline(cin, line);
        istringstream rowStream(line);
        string contact;
        int j = 0;
        while (getline(rowStream, contact, ',')) {
```

```
                contacts[i][j++] = contact == "1";
            }
        }

        // 对每个确诊病例执行深度优先搜索
        for (int index : confirmedCases) {
            if (!visited[index]) {
                dfs(contacts, visited, index);
            }
        }

        int count = 0; // 需要进行核酸检测的人数
        // 遍历访问记录数组，统计需要进行核酸检测的人数
        for (int i = 0; i < N; ++i) {
            if (visited[i] && confirmedCases.find(i) == confirmedCases.end()) {
                count++; // 如果该人员被访问过且不是确诊病例，则计数器加一
            }
        }

        cout << count << endl; // 输出需要进行核酸检测的人数
        return 0;
    }
```

Java

Java

```java
import java.util.*;

public class Main {
    // 深度优先搜索（DFS）算法
    public static void dfs(boolean[][] contacts, boolean[] visited, int start) {
        visited[start] = true; // 标记当前节点为已访问
        for (int i = 0; i < contacts.length; i++) {
            // 如果当前节点与其他节点有接触，并且该节点未被访问过
            if (contacts[start][i] == true && !visited[i]) {
                dfs(contacts, visited, i); // 递归访问该节点
            }
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int N = Integer.parseInt(scanner.nextLine()); // 读取总人数
        String[] confirmedCases = scanner.nextLine().split(","); // 读取确诊病例人员编号
        boolean[][] contacts = new boolean[N][N]; // 创建接触矩阵
        boolean[] visited = new boolean[N]; // 创建访问记录数组

        // 构建接触矩阵
        for (int i = 0; i < N; i++) {
            String[] row = scanner.nextLine().split(",");
            for (int j = 0; j < N; j++) {
                contacts[i][j] = "1".equals(row[j]); // 将接触情况转换为布尔值存储
            }
        }

        // 对每个确诊病例执行深度优先搜索
        for (String caseIndex : confirmedCases) {
            int index = Integer.parseInt(caseIndex);
            dfs(contacts, visited, index);
        }

        int count = 0; // 需要进行核酸检测的人数
        // 遍历访问记录数组，统计需要进行核酸检测的人数
        for (int i = 0; i < N; i++) {
            if (visited[i]) { // 如果该人员被访问过
                // 检查该人员是否为确诊病例
                boolean isConfirmedCase = Arrays.asList(confirmedCases).contains(String.valueOf(i));
```

```
                if (!isConfirmedCase) { // 如果不是确诊病例，则计数器加一
                    count++;
                }
            }
        }

        System.out.println(count); // 输出需要进行核酸检测的人数
        scanner.close(); // 关闭扫描器
    }
}
```

javaScript

Plain Text

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

// 深度优先搜索（DFS）算法
function dfs(contacts, visited, start) {
  visited[start] = true; // 标记当前节点为已访问
  for (let i = 0; i < contacts.length; i++) {
    // 如果当前节点与其他节点有接触，并且该节点未被访问过
    if (contacts[start][i] && !visited[i]) {
      dfs(contacts, visited, i); // 递归访问该节点
    }
  }
}

// 读取输入数据
let lineCount = 0;
let N = 0;
let confirmedCases = [];
let contacts = [];
let visited = [];

rl.on('line', (line) => {
  if (lineCount === 0) {
    N = parseInt(line); // 读取总人数
    visited = new Array(N).fill(false); // 初始化访问记录数组
  } else if (lineCount === 1) {
    confirmedCases = line.split(',').map(Number); // 读取确诊病例人员编号
    contacts = new Array(N).fill(null).map(() => new Array(N).fill(false)); // 创建接触矩阵
  } else {
    let row = line.split(',').map(Number);
    contacts[lineCount - 2] = row.map(value => value === 1); // 构建接触矩阵
    if (lineCount - 2 === N - 1) {
      rl.close(); // 如果已读取完所有输入数据，则关闭读取接口
    }
  }
  lineCount++;
});

rl.on('close', () => {
  // 对每个确诊病例执行深度优先搜索
  confirmedCases.forEach((caseIndex) => {
```

```
      dfs(contacts, visited, caseIndex);
    });

    let count = 0; // 需要进行核酸检测的人数
    // 遍历访问记录数组，统计需要进行核酸检测的人数
    visited.forEach((hasVisited, i) => {
      if (hasVisited && !confirmedCases.includes(i)) {
        count++; // 如果该人员被访问过且不是确诊病例，则计数器加一
      }
    });

    console.log(count); // 输出需要进行核酸检测的人数
  });
```

Python

Plain Text

```
import sys

# 深度优先搜索（DFS）算法
def dfs(contacts, visited, start):
    visited[start] = True  # 标记当前节点为已访问
    for i in range(len(contacts)):
        # 如果当前节点与其他节点有接触，并且该节点未被访问过
        if contacts[start][i] and not visited[i]:
            dfs(contacts, visited, i)  # 递归访问该节点

# 读取输入数据
N = int(input())  # 读取总人数
confirmed_cases = list(map(int, input().split(',')))  # 读取确诊病例人员编号
contacts = []  # 创建接触矩阵
visited = [False] * N  # 创建访问记录数组

# 构建接触矩阵
for _ in range(N):
    row = list(map(int, input().split(',')))
    contacts.append([bool(x) for x in row])

# 对每个确诊病例执行深度优先搜索
for case_index in confirmed_cases:
    dfs(contacts, visited, case_index)

count = 0  # 需要进行核酸检测的人数
# 遍历访问记录数组，统计需要进行核酸检测的人数
for i, has_visited in enumerate(visited):
    if has_visited and i not in confirmed_cases:
        count += 1  # 如果该人员被访问过且不是确诊病例，则计数器加一

print(count)  # 输出需要进行核酸检测的人数
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 深度优先搜索（DFS）算法
void dfs(int contacts[][100], int visited[], int start, int N) {
    visited[start] = 1; // 标记当前节点为已访问
    for (int i = 0; i < N; i++) {
        // 如果当前节点与其他节点有接触，并且该节点未被访问过
        if (contacts[start][i] == 1 && visited[i] == 0) {
            dfs(contacts, visited, i, N); // 递归访问该节点
        }
    }
}

int main() {
    int N; // 总人数
    scanf("%d", &N);
    getchar(); // 读取换行符

    int confirmed_cases[100]; // 确诊病例人员编号数组
    char input[300]; // 用于接收输入的确诊病例字符串
    fgets(input, 300, stdin); // 读取确诊病例字符串

    // 分割确诊病例字符串，转换为整数，并存储到确诊病例数组中
    int case_count = 0; // 确诊病例的数量
    char *token = strtok(input, ",");
    while (token != NULL) {
        confirmed_cases[case_count++] = atoi(token);
        token = strtok(NULL, ",");
    }

    int contacts[100][100]; // 接触矩阵
    int visited[100] = {0}; // 访问记录数组

    // 构建接触矩阵
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            scanf("%d", &contacts[i][j]);
            if (j < N - 1) getchar(); // 读取逗号
        }
        getchar(); // 读取换行符
    }

    // 对每个确诊病例执行深度优先搜索
```

```
    for (int i = 0; i < case_count; i++) {
        dfs(contacts, visited, confirmed_cases[i], N);
    }

    int count = 0; // 需要进行核酸检测的人数
    // 遍历访问记录数组，统计需要进行核酸检测的人数
    for (int i = 0; i < N; i++) {
        if (visited[i] == 1) {
            int is_confirmed_case = 0;
            for (int j = 0; j < case_count; j++) {
                if (confirmed_cases[j] == i) {
                    is_confirmed_case = 1;
                    break;
                }
            }
            if (!is_confirmed_case) {
                count++; // 如果该人员被访问过且不是确诊病例，则计数器加一
            }
        }
    }

    printf("%d\n", count); // 输出需要进行核酸检测的人数
    return 0;
}
```

完整用例

用例1

Plain Text

```
4
0
1,0,0,0
0,1,0,0
0,0,1,0
0,0,0,1
```

用例2

```
3
0,1
1,1,0
1,1,1
0,1,1
```

用例3

```
5
1,3
0,1,0,0,0
1,1,1,1,0
0,1,1,0,0
0,1,0,1,1
0,0,0,1,1
```

用例4

```
6
2,4
0,0,1,0,0,0
0,0,0,1,0,0
1,0,1,1,1,0
0,1,1,1,0,0
0,0,1,0,1,1
0,0,0,0,1,1
```

用例5

```
4
0,2
1,1,1,0
1,1,0,1
1,0,1,1
0,1,1,1
```

用例6

```
7
1,3,5
0,1,0,0,0,0,0
1,1,1,0,0,0,0
0,1,1,1,0,0,0
0,0,1,1,1,0,0
0,0,0,1,1,1,0
0,0,0,0,1,1,1
0,0,0,0,0,1,1
```

用例7

```
3
0
1,1,0
1,1,1
0,1,1
```

用例8

```
5
0,1,2
1,1,1,0,0
1,1,1,1,0
1,1,1,0,1
0,1,0,1,0
0,0,1,0,1
```

用例9

```
8
4,7
0,0,0,0,1,0,0,0
0,0,0,0,0,1,0,0
0,0,0,0,0,0,1,0
0,0,0,0,0,0,0,1
1,0,0,0,1,1,0,0
0,1,0,0,1,1,1,0
0,0,1,0,0,1,1,1
0,0,0,1,0,0,1,1
```

用例10

```
5
0,4
1,0,0,1,1
0,1,1,0,0
0,1,1,0,0
1,0,0,1,0
1,0,0,0,1
```

来自: 华为OD机考2025B卷 – 核酸检测（Java & Python& JS & C++ & C ）–CSDN博客

华为OD机考2025B卷 - 字符串分割转换（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

给定一个非空字符串S，其被N个‘-’分隔成N+1的子串，给定正整数K，要求除第一个子串外，其余的子串每K个字符组成新的子串，并用‘-’分隔。
对于新组成的每一个子串，如果它含有的小写字母比大写字母多，则将这个子串的所有大写字母转换为小写字母；
反之，如果它含有的大写字母比小写字母多，则将这个子串的所有小写字母转换为大写字母；大小写字母的数量相等时，不做转换。

输入描述

输入为两行，第一行为参数K，第二行为字符串S。

输出描述

输出转换后的字符串。

示例1

输入

```
3
12abc-abCABc-4aB@
```

输出

```
12abc-abc-ABC-4aB-@
```

说明

> 子串为12abc、abCABc、4aB@，第一个子串保留，
> 后面的子串每3个字符一组为abC、ABc、4aB、@，
> abC中小写字母较多，转换为abc，
> ABc中大写字母较多，转换为ABC，
> 4aB中大小写字母都为1个，不做转换，
> @中没有字母，连起来即12abc-abc-ABC-4aB-@

示例2

输入

```
12
12abc-abCABc-4aB@
```

输出

```
12abc-abCABc4aB@
```

说明

> 子串为12abc、abCABc、4aB@，第一个子串保留，
> 后面的子串每12个字符一组为abCABc4aB@，
> 这个子串中大小写字母都为4个，不做转换，
> 连起来即12abc-abCABc4aB@

解题思路

这个题目的目标是对字符串进行重新分组和大小写转换。题目的关键点和步骤如下：

1. **字符串分组**：
   - 给定一个字符串 `S`，这个字符串用 `'-'` 分隔成了多个子串。假设有 `N` 个 `'-'`，那么 `S` 被分隔成 `N+1` 个子串。
   - 给定一个正整数 `K`，要求除第一个子串外，其余的子串按照每 `K` 个字符重新分组。
2. **大小写转换**：
   - 对于新分组后的每个子串，如果小写字母的数量比大写字母多，就将这个子串的所有大写字母转换为小写字母；
   - 如果大写字母的数量比小写字母多，就将这个子串的所有小写字母转换为大写字母；
   - 如果大小写字母数量相等，则保持不变。
3. **输出要求**：
   - 第一个子串保持不变。

- 后面的子串根据重新分组和大小写转换后的结果，再次用 `'-'` 分隔，形成最终的输出字符串。

示例1解析：

- 输入：`K=3`，字符串为 `12abc-abCABc-4aB@`
- 初始子串分割结果：`12abc`、`abCABc`、`4aB@`
- 分组处理：
  - 第一个子串 `12abc` 保持不变。
  - 第二个子串 `abCABc` 按照每 `3` 个字符重新分组：`abC`、`ABc`。分别统计大小写字母数量，`abC` 中小写字母较多，因此将 `abC` 转换为 `abc`；`ABc` 中大写字母较多，因此将 `ABc` 转换为 `ABC`。
  - 第三个子串 `4aB@` 按照每 `3` 个字符重新分组：`4aB`、`@`。`4aB` 中大小写字母数量相等，因此保持不变。
- 最终输出：`12abc-abc-ABC-4aB-@`

示例2解析：

- 输入：`K=12`，字符串为 `12abc-abCABc-4aB@`
- 初始子串分割结果：`12abc`、`abCABc`、`4aB@`
- 分组处理：
  - 第一个子串 `12abc` 保持不变。
  - 第二个子串 `abCABc` 和 `4aB@` 加起来不超过 `12` 个字符，所以它们合并为一个子串：`abCABc4aB@`，大小写字母数量相等，因此保持不变。
- 最终输出：`12abc-abCABc4aB@`

Java

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int k = Integer.parseInt(scanner.nextLine()); // 输入正整数K
        String s = scanner.nextLine(); // 输入字符串S
        String[] sArr = s.split("-"); // 将S按照“-”分隔成N+1个子串
        String prefix = sArr[0]; // 第一个子串
        StringBuilder postfixSb = new StringBuilder(); // 除第一个子串外的所有子串
        for (int i = 1; i < sArr.length; i++) {
            postfixSb.append(sArr[i]);
        }
        char[] postfixChars = postfixSb.toString().toCharArray(); // 将除第一个子串外的所有子串拼接成字符数组
        StringBuilder newSb = new StringBuilder();
        for (int i = 0; i < postfixChars.length; i++) { // 将除第一个子串外的所有子串每K个字符组成新的子串，并用‘-’分隔
            if ((i + 1) % k == 0 && i + 1 != postfixChars.length) {
                newSb.append(postfixChars[i]).append("-");
            } else {
                newSb.append(postfixChars[i]);
            }
        }
        String[] newSArr = newSb.toString().split("-"); // 将新组成的每一个子串按照“-”分隔
        StringBuilder resultSb = new StringBuilder();
        for (String str : newSArr) { // 对于新组成的每一个子串，如果它含有的小写字母比大写字母多，则将这个子串的所有大写字母转换为小写字母；反之，如果它含有的大写字母比小写字母多，则将这个子串的所有小写字母转换为大写字母；大小写字母的数量相等时，不做转换

            long upperCase = str.chars().filter(Character::isUpperCase).count();

            long lowerCase = str.chars().filter(Character::isLowerCase).count();
            if (upperCase > lowerCase) {
                resultSb.append(str.toUpperCase()).append("-");
            } else if (lowerCase > upperCase) {
                resultSb.append(str.toLowerCase()).append("-");
            } else {
                resultSb.append(str).append("-");
            }
        }
```

```java
        String postfix = resultSb.toString().substring(0, resultSb.length() - 1); // 将处理后的新组成的每一个子串按照“-”拼接成字符串
        System.out.println(prefix + "-" + postfix); // 输出转换后的字符串
    }
}
```

Python

Plain Text

```python
k = int(input()) # 输入正整数K
s = input() # 输入字符串S
sArr = s.split("-") # 将S按照“-”分隔成N+1个子串
prefix = sArr[0] # 第一个子串
postfixSb = "" # 除第一个子串外的所有子串
for i in range(1, len(sArr)):
    postfixSb += sArr[i]
postfixChars = list(postfixSb) # 将除第一个子串外的所有子串拼接成字符数组
newSb = ""
for i in range(len(postfixChars)): # 将除第一个子串外的所有子串每K个字符组成新的子串，并用‘-’分隔
    if (i + 1) % k == 0 and i + 1 != len(postfixChars):
        newSb += postfixChars[i] + "-"
    else:
        newSb += postfixChars[i]
newSArr = newSb.split("-") # 将新组成的每一个子串按照“-”分隔
resultSb = ""
for str in newSArr: # 对于新组成的每一个子串，如果它含有的小写字母比大写字母多，则将这个子串的所有大写字母转换为小写字母；反之，如果它含有的大写字母比小写字母多，则将这个子串的所有小写字母转换为大写字母；大小写字母的数量相等时，不做转换
    upperCase = sum(1 for c in str if c.isupper())
    lowerCase = sum(1 for c in str if c.islower())
    if upperCase > lowerCase:
        resultSb += str.upper() + "-"
    elif lowerCase > upperCase:
        resultSb += str.lower() + "-"
    else:
        resultSb += str + "-"
postfix = resultSb[:-1] # 将处理后的新组成的每一个子串按照“-”拼接成字符串
print(prefix + "-" + postfix) # 输出转换后的字符串
```

JavaScript

Plain Text

```
const readline = require("readline");

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

const lines = [];
rl.on("line", (line) => {
  lines.push(line);

  if (lines.length === 2) {
    const k = parseInt(lines[0]);
    const s = lines[1];
     const arr = s.split("-");
  const prefix = arr.shift();
  const postfix = arr
    .join("")
    .match(new RegExp(`.{1,${k}}`, "g"))
    .map((str) => {
      let upperCase = 0;
      let lowerCase = 0;
      [...str].forEach((char) => {
        if (/[a-z]/.test(char)) lowerCase++;
        if (/[A-Z]/.test(char)) upperCase++;
      });
      if (upperCase > lowerCase) {
        return str.toUpperCase();
      }
      if (lowerCase > upperCase) {
        return str.toLowerCase();
      }
      return str;
    })
    .join("-");


    console.log(prefix + "-" + postfix);
  }
});
```

C++

Plain Text

```
#include <iostream>
#include <sstream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int k; // 用于存储输入的正整数K
    string s; // 用于存储输入的字符串S
    getline(cin, s); // 读取第一行输入，即正整数K
    k = stoi(s); // 将字符串转换为整数K
    getline(cin, s); // 读取第二行输入，即字符串S

    vector<string> sArr; // 用于存储分割后的子串
    size_t pos = 0;
    while ((pos = s.find("-")) != string::npos) { // 将字符串S按照“-”分隔成多个子串
        string token = s.substr(0, pos); // 获取当前子串
        sArr.push_back(token); // 将子串添加到数组中
        s.erase(0, pos + 1); // 从原字符串中移除已处理的部分
    }
    sArr.push_back(s); // 添加最后一个子串（没有“-”的部分）

    string prefix = sArr[0]; // 第一个子串，按照题目要求保留
    string postfix; // 用于存储除第一个子串外的所有子串

    for (int i = 1; i < sArr.size(); i++) {
        postfix += sArr[i]; // 将所有其他子串拼接成一个字符串
    }

    vector<char> postfixChars(postfix.begin(), postfix.end()); // 将拼接后的字符串转换为字符数组

    string newS = ""; // 用于存储重新分组后的字符串
    for (int i = 0; i < postfixChars.size(); i++) {
        if ((i + 1) % k == 0 && i + 1 != postfixChars.size()) {
            newS += postfixChars[i]; // 添加当前字符
            newS += '-'; // 在每K个字符后添加一个“-”作为分隔符
        } else {
            newS += postfixChars[i]; // 否则只添加当前字符
        }
    }

    vector<string> newSArr; // 用于存储重新分组后的子串
```

```
    pos = 0;
    while ((pos = newS.find("-")) != string::npos) { // 将重新分组后的字符串按照“-”分隔
        string token = newS.substr(0, pos); // 获取每个新子串
        newSArr.push_back(token); // 将子串添加到数组中
        newS.erase(0, pos + 1); // 从原字符串中移除已处理的部分
    }
    newSArr.push_back(newS); // 添加最后一个子串

    string result = ""; // 用于存储最终的结果字符串
    for (string str : newSArr) {
        int upperCase = count_if(str.begin(), str.end(), [](char c) { return isupper(c); }); // 统计大写字母的数量
        int lowerCase = count_if(str.begin(), str.end(), [](char c) { return islower(c); }); // 统计小写字母的数量

        // 根据大写字母和小写字母的数量决定是否进行大小写转换
        if (upperCase > lowerCase) {
            transform(str.begin(), str.end(), str.begin(), ::toupper); // 大写字母多，全部转换为大写
            result += str; // 将转换后的子串添加到结果字符串中
            result += '-'; // 添加“-”作为分隔符
        } else if (lowerCase > upperCase) {
            transform(str.begin(), str.end(), str.begin(), ::tolower); // 小写字母多，全部转换为小写
            result += str; // 将转换后的子串添加到结果字符串中
            result += '-'; // 添加“-”作为分隔符
        } else {
            result += str; // 大小写字母数量相等，不做转换，直接添加子串
            result += '-'; // 添加“-”作为分隔符
        }
    }

    result = result.substr(0, result.length() - 1); // 移除最后一个多余的“-”

    cout << prefix << "-" << result << endl; // 输出最终的结果字符串

    return 0;
}
```

C语言

Plain Text

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_LEN 500001 // 假设字符串的最大长度不超过500000

// 函数：将子串中的字母根据大小写字母数量转换为全大写或全小写
void convert(char *str) {
    int upper = 0, lower = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        if (isupper(str[i])) upper++;
        if (islower(str[i])) lower++;
    }
    for (int i = 0; str[i] != '\0'; i++) {
        if (upper > lower) str[i] = toupper(str[i]);
        if (lower > upper) str[i] = tolower(str[i]);
    }
}

int main() {
    int k; // 输入正整数K
    scanf("%d\n", &k);

    char s[MAX_LEN]; // 输入字符串S
    fgets(s, MAX_LEN, stdin);
    s[strcspn(s, "\n")] = 0; // 去除换行符

    char *token = strtok(s, "-"); // 第一个子串
    printf("%s", token); // 输出第一个子串

    char postfix[MAX_LEN] = ""; // 除第一个子串外的所有子串
    while ((token = strtok(NULL, "-")) != NULL) {
        strcat(postfix, token);
    }

    char newStr[MAX_LEN] = ""; // 新组成的字符串
    int len = strlen(postfix);
    for (int i = 0; i < len; i++) {
        strncat(newStr, &postfix[i], 1); // 逐个字符拼接到新字符串中
        if ((i + 1) % k == 0 && i + 1 != len) {
            strcat(newStr, "-"); // 每K个字符后添加分隔符
        }
    }

    char *newToken = strtok(newStr, "-"); // 按照“-”分隔新组成的字符串
```

```
    while (newToken != NULL) {
        convert(newToken); // 转换子串中的字母
        printf("-%s", newToken); // 输出转换后的子串
        newToken = strtok(NULL, "-"); // 继续获取下一个子串
    }

    return 0;
}
```

完整用例

用例1

Plain Text

```
3
12abc-abCABc-4aB@
```

用例2

Plain Text

```
2
aBcDe-FgHiJ-KlMnO
```

用例3

Plain Text

```
4
aBcDe-FgHiJ-KlMnO
```

用例4

Plain Text

```
5
aBcDe-FgHiJ-KlMnO
```

用例5

```
1
aBcDe-FgHiJ-KlMnO
```

用例6

```
6
aBcDe-FgHiJ-KlMnO
```

用例7

```
3
aBcDe-FgHiJ-KlMnO
```

用例8

```
2
aBcDe-FgHiJ-KlMnO
```

用例9

```
4
aBcDe-FgHiJ-KlMnO
```

用例10

```
5
aBcDe-FgHiJ-KlMnO
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- 示例2
- 解题思路
    - 示例1解析：
    - 示例2解析：
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
    - 用例1
    - 用例2
    - 用例3
    - 用例4
    - 用例5
    - 用例6
    - 用例7
    - 用例8
    - 用例9
    - 用例10

> 来自: 华为OD机考2025B卷 - 字符串分割转换（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 停车场最大距离/找车位（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

停车场有一横排车位，0代表没有停车，1代表有车。至少停了一辆车在车位上，也至少有一个空位没有停车。
为了防剐蹭，需为停车人找到一个车位，使得距停车人的车最近的车辆的距离是最大的，返回此时的**最大距离**。

输入描述

1. 一个用半角逗号分割的停车标识字符串，停车标识为0或1，0为空位，1为已停车。
2. 停车位最多100个。

输出描述

输出一个整数记录最大距离。

示例1

输入

```
1,0,0,0,0,1,0,0,1,0,1
```

输出

```
2
```

说明

> 当车停在第3个位置上时，离其最近的的车距离为2（1到3）。

> 当车停在第4个位置上时，离其最近的的车距离为2（4到6）。
> 其他位置距离为1。
> 因此最大距离为2

解题思路

1. **计算距离**：
    - 为了确保车辆停放时与最近的车之间的距离最大，考虑以下几种情况：
        i. 如果空位段位于开头（在第一个'1'之前），那么最大距离就是空位段的长度。
        ii. 如果空位段位于结尾（在最后一个'1'之后），同样，最大距离就是这个段的长度。
        iii. 如果空位段位于中间（在两个'1'之间），最大的距离就等于这个段长度的一半（向下取整），因为车辆放置在中间时，可以与两侧的车辆保持最大距离。
1. **获取最大距离**：在计算所有独立空位段的最大距离后，返回这个最大值作为最终结果。

Java

```java
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String inputNum = scanner.nextLine();
        List<String> inputNumList = new ArrayList<>();
        String temp = "";
        for (int i = 0; i < inputNum.length(); i++) {
            if (inputNum.charAt(i) == ',') {
                inputNumList.add(temp);
                temp = "";
            } else {
                temp += inputNum.charAt(i);
            }
        }
        inputNumList.add(temp);

        String lineNum = "";
        for (int i = 0; i < inputNumList.size(); i++) {
            lineNum += inputNumList.get(i);
        }
        List<String> lineNumList = new ArrayList<>();
        temp = "";
        for (int i = 0; i < lineNum.length(); i++) {
            if (lineNum.charAt(i) == '1') {
                lineNumList.add(temp);
                temp = "";
            } else {
                temp += lineNum.charAt(i);
            }
        }
        lineNumList.add(temp);

        List<Integer> res = new ArrayList<>();
        res.add(lineNumList.get(0).length());
        res.add(lineNumList.get(lineNumList.size() - 1).length());
        int maxLen = 0;
        for (int i = 0; i < lineNumList.size(); i++) {
            maxLen = Math.max(maxLen, lineNumList.get(i).length());
        }
        res.add((maxLen + 1) / 2);
```

```
        int num = Collections.max(res);

        System.out.println(num);
    }
}
```

Python

```
inputNum = input().split(",")
lineNum = "".join(inputNum).split("1")

# 定义一个列表res，其中包含三个元素：
# res[0]表示第一个空位之前的距离，即lineNum[0]的长度；
# res[1]表示最后一个空位之后的距离，即lineNum[-1]的长度；
# res[2]表示最大距离，即lineNum中最长的空位长度加1再除以2。
res = [len(lineNum[0]), len(lineNum[-1]), (max(len(i) for i in lineNum) + 1) // 2]

# 找出res中的最大值，并将其赋给变量num
num = max(res)

# 输出变量num，即最大距离
print(num)
```

JavaScript

Plain Text

```
const readline = require('readline');

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

rl.on('line', (input) => {
  const inputNum = input.split(",");
  const lineNum = inputNum.join("").split("1");

  const res = [lineNum[0].length, lineNum[lineNum.length - 1].length, (Math.max(...lineNum.map(i => i.length)) + 1) / 2];

  const num = Math.max(...res);

  console.log(Math.floor(num));

  rl.close();
});
```

C++

Plain Text

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    string inputNum;
    getline(cin, inputNum);
    vector<string> inputNumList;
    string temp = "";
    for (int i = 0; i < inputNum.length(); i++) {
        if (inputNum[i] == ',') {
            inputNumList.push_back(temp);
            temp = "";
        }
        else {
            temp += inputNum[i];
        }
    }
    inputNumList.push_back(temp);

    string lineNum = "";
    for (int i = 0; i < inputNumList.size(); i++) {
        lineNum += inputNumList[i];
    }
    vector<string> lineNumList;
    temp = "";
    for (int i = 0; i < lineNum.length(); i++) {
        if (lineNum[i] == '1') {
            lineNumList.push_back(temp);
            temp = "";
        }
        else {
            temp += lineNum[i];
        }
    }
    lineNumList.push_back(temp);

    vector<int> res;
    res.push_back(lineNumList[0].length());
    res.push_back(lineNumList[lineNumList.size() - 1].length());
    int maxLen = 0;
    for (int i = 0; i < lineNumList.size(); i++) {
        maxLen = max(maxLen, (int)lineNumList[i].length());
    }
```

```
    res.push_back((maxLen + 1) / 2);

    int num = *max_element(res.begin(), res.end());

    cout << num << endl;

    return 0;
}
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 找出两个整数中的最大值
int max(int a, int b) {
    return (a > b) ? a : b;
}

// 找出数组中的最大值
int findMax(int arr[], int size) {
    int maxVal = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > maxVal) {
            maxVal = arr[i];
        }
    }
    return maxVal;
}

int main() {
    char inputNum[1000];
    fgets(inputNum, sizeof(inputNum), stdin);

    // 移除可能的换行符
    inputNum[strcspn(inputNum, "\n")] = 0;

    // 存储分割后的字符串
    char inputNumList[1000][1000];
    int inputNumListSize = 0;

    char temp[1000] = "";
    int tempIndex = 0;

    // 按逗号分割字符串
    for (int i = 0; i < strlen(inputNum); i++) {
        if (inputNum[i] == ',') {
            temp[tempIndex] = '\0';
            strcpy(inputNumList[inputNumListSize++], temp);
            tempIndex = 0;
            temp[0] = '\0';
        } else {
            temp[tempIndex++] = inputNum[i];
        }
    }
```

```
    temp[tempIndex] = '\0';
    strcpy(inputNumList[inputNumListSize++], temp);

    // 合并所有字符串
    char lineNum[1000] = "";
    for (int i = 0; i < inputNumListSize; i++) {
        strcat(lineNum, inputNumList[i]);
    }

    // 按'1'分割字符串
    char lineNumList[1000][1000];
    int lineNumListSize = 0;

    tempIndex = 0;
    temp[0] = '\0';

    for (int i = 0; i < strlen(lineNum); i++) {
        if (lineNum[i] == '1') {
            temp[tempIndex] = '\0';
            strcpy(lineNumList[lineNumListSize++], temp);
            tempIndex = 0;
            temp[0] = '\0';
        } else {
            temp[tempIndex++] = lineNum[i];
        }
    }
    temp[tempIndex] = '\0';
    strcpy(lineNumList[lineNumListSize++], temp);

    // 计算结果
    int res[3];
    res[0] = strlen(lineNumList[0]);
    res[1] = strlen(lineNumList[lineNumListSize - 1]);

    int maxLen = 0;
    for (int i = 0; i < lineNumListSize; i++) {
        maxLen = max(maxLen, strlen(lineNumList[i]));
    }
    res[2] = (maxLen + 1) / 2;

    // 找出最大值
    int num = findMax(res, 3);

    printf("%d\n", num);

    return 0;
}
```

完整用例

用例1

```
1,0,0,0,0,1,0,0,1,0,1
```

用例2

```
0,1,0,0,0,0,0,0,0,0,1
```

用例3

```
1,1,1,1,1,1,1,1,1,1,0
```

用例4，无效用例，已删除

```

```

用例5

```
1,0,1,0,1,0,1,0,1,0,1
```

用例6

```
0,1,0,1,0,1,0,1,0,1,0
```

用例7

```
1,1,1,0,0,0,0,0,0,0,0
```

用例8

```
0,0,0,0,0,0,0,1,1,1,1
```

用例9

```
1,0,0,0,1,0,0,0,1,0,0
```

用例10

```
0,0,0,0,1,0,0,0,0,0,0
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- 解题思路
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
- 用例1
- 用例2
- 用例3
- 用例4，无效用例，已删除

- 用例5
- 用例6
- 用例7
- 用例8
- 用例9
- 用例10

> 来自: 华为OD机考2025B卷 - 停车场最大距离/找车位（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 小明减肥（Java & Python& JS & C++ & C ）_c++小明有n个可选运动,每个运动有对应卡路里,想选出其中k个运动且卡路里和为t。k,-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看

题目描述

小明有n个可选运动，每个运动有对应卡路里，想选出其中k个运动且卡路里和为t。k，t，n都是给定的。求出可行解数量

输入描述

第一行输入 n t k
第一行输入 每个运动的卡路里 按照空格进行分割

备注

- `0 < n < 10`
- `t > 0, 0 < k <= n`
- 每个运动量的卡路里 > 0

输出描述

求出可行解数量

示例1

输入

```Plain Text
4 3 2
1 1 2 3
```

输出

```
2
```

说明

> 可行解为2，选取{0, 2}, {1,2}两种方式。

解题思路

暴力回溯

这个问题的核心目标是选出 `k` 个运动，使得它们的卡路里总和恰好是 `t` 。由于运动的数量 `n` 较小（0 < n < 10），我们可以利用回溯（Backtracking）算法来穷举所有可能的组合，从中找到符合条件的组合。

回溯函数 `backtrack`：

- 函数接受当前的运动索引 `index` ，还需要选择的运动数量 `k` ，以及目标卡路里 `t` 和当前已经选取的卡路里和 `currentSum` 以及已选运动的数量 `count` 。
- **终止条件**:
  - 如果已经选出了 `k` 个运动且它们的卡路里和正好等于 `t` ，则找到一种有效组合，返回 1。
  - 如果当前运动超出数组边界，或已选运动数量达到了 `k` ，或当前的卡路里和超过了 `t` ，则返回 0，表示这一条路径不再有效。
- **递归选择**:
  - 选择当前运动并递归继续寻找其他运动，更新当前总和和已选数量。
  - 不选择当前运动，则直接递归查找下一个运动。
- 最后将两种选择的结果相加，返回总的有效组合数。

动态规划

这个问题还可以通过动态规划来解决，目标是找到选择k个运动使得卡路里总和为t的不同组合数量。思路如下：

解题思路

1. **定义状态**：
   - 设 `dp[j][m]` 表示选择 `j` 个运动，卡路里总和为 `m` 的组合数量。
2. **初始化状态**：
   - `dp[0][0] = 1` ，因为选择0个运动使得卡路里和为0的方式只有一种，即什么都不选择。
   - 其他状态 `dp[j][m]` 初始化为0，因为在未填充前没有任何组合。
3. **状态转移方程**：
   - 对于每一个运动的卡路里值 `cal` ，我们从后往前遍历选择数量 `j` （从 `k` 到1）和卡路里总和 `m` （从 `t` 到 `cal` ）。
   - 转移方程为：
     - `dp[j][m] += dp[j-1][m-cal]`

- 这个公式的意思是：如果当前选择的运动卡路里是 `cal` ，那么要使得选择 `j` 个运动且卡路里为 `m` ，我们需要在之前的组合中选择 `j-1` 个运动，其卡路里和为 `m-cal` 。

Java

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取输入：n, t, k
        String[] firstLine = scanner.nextLine().trim().split(" ");
        int n = Integer.parseInt(firstLine[0]); // 运动数量
        int t = Integer.parseInt(firstLine[1]); // 目标卡路里
        int k = Integer.parseInt(firstLine[2]); // 需要选择的运动数量

        // 读取每个运动的卡路里
        String[] secondLine = scanner.nextLine().trim().split(" ");
        int[] calories = new int[n];
        for (int i = 0; i < n; i++) {
            calories[i] = Integer.parseInt(secondLine[i]);
        }

        // 使用回溯法求解
        int result = backtrack(calories, 0, k, t, 0, 0);
        System.out.println(result);

        scanner.close();
    }

    /**
     * 回溯法计算符合条件的解决方案数量
     * @param calories 卡路里数组
     * @param index 当前考虑的运动索引
     * @param k 还需要选择的运动数量
     * @param t 目标卡路里
     * @param currentSum 当前已选运动的卡路里和
     * @param count 已选择的运动数量
     * @return 符合条件的方案数
     */
    private static int backtrack(int[] calories, int index, int k, int t, int currentSum, int count) {
        // 如果已经选择了k个运动且卡路里和为t，找到一种方案
        if (count == k && currentSum == t) {
            return 1;
        }

        // 如果运动已经考虑完，或者已经选了k个运动，或者当前和已经超过t，无法构成有效方案
```

```java
        if (index == calories.length || count == k || currentSum > t) {
            return 0;
        }

        // 选择当前运动
        int pickCurrent = backtrack(calories, index + 1, k, t, currentSum + calories[index], count + 1);

        // 不选择当前运动
        int skipCurrent = backtrack(calories, index + 1, k, t, currentSum, count);

        // 返回两种选择的总方案数
        return pickCurrent + skipCurrent;
    }
}



import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取输入: n, t, k
        String[] firstLine = scanner.nextLine().trim().split(" ");
        int n = Integer.parseInt(firstLine[0]); // 运动数量
        int t = Integer.parseInt(firstLine[1]); // 目标卡路里
        int k = Integer.parseInt(firstLine[2]); // 需要选择的运动数量

        // 读取每个运动的卡路里
        String[] secondLine = scanner.nextLine().trim().split(" ");
        int[] calories = new int[n];
        for (int i = 0; i < n; i++) {
            calories[i] = Integer.parseInt(secondLine[i]);
        }

        // 创建二维DP数组
        // dp[j][m] 表示选择j个运动, 总卡路里为m的方案数
        int[][] dp = new int[k+1][t+1];

        // 初始化: 选0个运动, 卡路里为0的方案数为1
        dp[0][0] = 1;

        // 填充DP表格
        for (int i = 0; i < n; i++) {
            int cal = calories[i]; // 当前运动的卡路里
```

```java

            // 需要从后往前遍历，避免重复计算
            for (int j = k; j >= 1; j--) {
                for (int m = t; m >= cal; m--) {
                    dp[j][m] += dp[j-1][m-cal];
                }
            }
        }

        System.out.println(dp[k][t]);

        scanner.close();
    }
}
```

Python

Plain Text

```
def backtrack(calories, index, k, t, current_sum, count):
    """
    回溯法计算符合条件的解决方案数量

    参数:
    calories -- 卡路里数组
    index -- 当前考虑的运动索引
    k -- 还需要选择的运动数量
    t -- 目标卡路里
    current_sum -- 当前已选运动的卡路里和
    count -- 已选择的运动数量

    返回:
    符合条件的方案数
    """
    # 如果已经选择了k个运动且卡路里和为t，找到一种方案
    if count == k and current_sum == t:
        return 1

    # 如果运动已经考虑完，或者已经选了k个运动，或者当前和已经超过t，无法构成有效方案
    if index == len(calories) or count == k or current_sum > t:
        return 0

    # 选择当前运动
    pick_current = backtrack(calories, index + 1, k, t, current_sum + calories[index], count + 1)

    # 不选择当前运动
    skip_current = backtrack(calories, index + 1, k, t, current_sum, count)

    # 返回两种选择的总方案数
    return pick_current + skip_current


first_line = input().strip().split()
n = int(first_line[0])  # 运动数量
t = int(first_line[1])  # 目标卡路里
k = int(first_line[2])  # 需要选择的运动数量

# 读取每个运动的卡路里
calories = list(map(int, input().strip().split()))

# 使用回溯法求解
result = backtrack(calories, 0, k, t, 0, 0)
```

```python
print(result)
```

```python
# 读取输入：n, t, k
first_line = input().strip().split()
n = int(first_line[0])  # 运动数量
t = int(first_line[1])  # 目标卡路里
k = int(first_line[2])  # 需要选择的运动数量

# 读取每个运动的卡路里
second_line = input().strip().split()
calories = [0] * n
for i in range(n):
    calories[i] = int(second_line[i])

# 创建二维DP数组
# dp[j][m] 表示选择j个运动，总卡路里为m的方案数
dp = [[0] * (t + 1) for _ in range(k + 1)]

# 初始化：选0个运动，卡路里为0的方案数为1
dp[0][0] = 1

# 填充DP表格
for i in range(n):
    cal = calories[i]  # 当前运动的卡路里

    # 需要从后往前遍历，避免重复计算
    for j in range(k, 0, -1):
        for m in range(t, cal - 1, -1):
            dp[j][m] += dp[j-1][m-cal]

print(dp[k][t])
```

JavaScript

Plain Text

```
// 引入readline模块处理标准输入
const readline = require('readline');

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

let lines = [];

// 监听行输入事件
rl.on('line', (line) => {
  lines.push(line);

  // 当收集到2行输入时，进行处理
  if (lines.length === 2) {
    // 解析第一行输入：n, t, k
    const [n, t, k] = lines[0].trim().split(' ').map(Number);

    // 解析第二行输入：每个运动的卡路里
    const calories = lines[1].trim().split(' ').map(Number);

    // 使用回溯法求解
    const result = backtrack(calories, 0, k, t, 0, 0);
    console.log(result);

    // 关闭输入流
    rl.close();
  }
});

/**
 * 回溯法计算符合条件的解决方案数量
 * @param {number[]} calories - 卡路里数组
 * @param {number} index - 当前考虑的运动索引
 * @param {number} k - 还需要选择的运动数量
 * @param {number} t - 目标卡路里
 * @param {number} currentSum - 当前已选运动的卡路里和
 * @param {number} count - 已选择的运动数量
 * @returns {number} - 符合条件的方案数
 */
function backtrack(calories, index, k, t, currentSum, count) {
  // 如果已经选择了k个运动且卡路里和为t，找到一种方案
  if (count === k && currentSum === t) {
    return 1;
```

```
  }

  // 如果运动已经考虑完，或者已经选了k个运动，或者当前和已经超过t，无法构成有效方案
  if (index === calories.length || count === k || currentSum > t) {
    return 0;
  }

  // 选择当前运动
  const pickCurrent = backtrack(calories, index + 1, k, t, currentSum + calories[index], count + 1);

  // 不选择当前运动
  const skipCurrent = backtrack(calories, index + 1, k, t, currentSum, count);

  // 返回两种选择的总方案数
  return pickCurrent + skipCurrent;
}





// 引入readline模块处理标准输入
const readline = require('readline');

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

let lines = [];

// 监听行输入事件
rl.on('line', (line) => {
  lines.push(line);

  // 当收集到2行输入时，进行处理
  if (lines.length === 2) {
    // 解析第一行输入：n, t, k
    const firstLine = lines[0].trim().split(' ');
    const n = parseInt(firstLine[0]); // 运动数量
    const t = parseInt(firstLine[1]); // 目标卡路里
    const k = parseInt(firstLine[2]); // 需要选择的运动数量

    // 解析第二行输入：每个运动的卡路里
    const secondLine = lines[1].trim().split(' ');
    const calories = new Array(n);
```

```
    for (let i = 0; i < n; i++) {
      calories[i] = parseInt(secondLine[i]);
    }

    // 创建二维DP数组
    // dp[j][m] 表示选择j个运动，总卡路里为m的方案数
    const dp = Array(k + 1).fill().map(() => Array(t + 1).fill(0));

    // 初始化：选0个运动，卡路里为0的方案数为1
    dp[0][0] = 1;

    // 填充DP表格
    for (let i = 0; i < n; i++) {
      const cal = calories[i]; // 当前运动的卡路里

      // 需要从后往前遍历，避免重复计算
      for (let j = k; j >= 1; j--) {
        for (let m = t; m >= cal; m--) {
          dp[j][m] += dp[j-1][m-cal];
        }
      }
    }

    console.log(dp[k][t]);

    // 关闭输入流
    rl.close();
  }
});
```

C++

Plain Text

```
#include <iostream>
#include <vector>
#include <sstream>

using namespace std;

// 回溯法计算符合条件的解决方案数量
int backtrack(const vector<int>& calories, int index, int k, int t, int currentSum, int count) {
    // 如果已经选择了k个运动且卡路里和为t，找到一种方案
    if (count == k && currentSum == t) {
        return 1;
    }

    // 如果运动已经考虑完完，或者已经选了k个运动，或者当前和已经超过t，无法构成有效方案
    if (index == calories.size() || count == k || currentSum > t) {
        return 0;
    }

    // 选择当前运动
    int pickCurrent = backtrack(calories, index + 1, k, t, currentSum + calories[index], count + 1);

    // 不选择当前运动
    int skipCurrent = backtrack(calories, index + 1, k, t, currentSum, count);

    // 返回两种选择的总方案数
    return pickCurrent + skipCurrent;
}

int main() {
    // 读取输入：n, t, k
    int n, t, k;
    cin >> n >> t >> k;

    // 读取每个运动的卡路里
    vector<int> calories(n);
    for (int i = 0; i < n; ++i) {
        cin >> calories[i];
    }

    // 使用回溯法求解
    int result = backtrack(calories, 0, k, t, 0, 0);
```

```
    cout << result << endl;

    return 0;
}



#include <iostream>
#include <vector>
#include <string>
#include <sstream>
using namespace std;
int main() {
    // 读取输入：n, t, k
    string firstLine;
    getline(cin, firstLine);
    istringstream iss1(firstLine);

    int n, t, k;
    iss1 >> n >> t >> k; // 分别是：运动数量、目标卡路里、需要选择的运动数量

    // 读取每个运动的卡路里
    string secondLine;
    getline(cin, secondLine);
    istringstream iss2(secondLine);

    vector<int> calories(n);
    for (int i = 0; i < n; i++) {
        iss2 >> calories[i];
    }

    // 创建二维DP数组
    // dp[j][m] 表示选择j个运动，总卡路里为m的方案数
    vector<vector<int>> dp(k + 1, vector<int>(t + 1, 0));

    // 初始化：选0个运动，卡路里为0的方案数为1
    dp[0][0] = 1;

    // 填充DP表格
    for (int i = 0; i < n; i++) {
        int cal = calories[i]; // 当前运动的卡路里

        // 需要从后往前遍历，避免重复计算
        for (int j = k; j >= 1; j--) {
```

```
                for (int m = t; m >= cal; m--) {
                    dp[j][m] += dp[j-1][m-cal];
                }
            }
        }

        // 输出结果
        cout << dp[k][t] << endl;

        return 0;
    }
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * 回溯法计算符合条件的解决方案数量
 *
 * @param calories 卡路里数组
 * @param index 当前考虑的运动索引
 * @param k 还需要选择的运动数量
 * @param t 目标卡路里
 * @param currentSum 当前已选运动的卡路里和
 * @param count 已选择的运动数量
 * @param n 运动总数
 * @return 符合条件的方案数
 */
int backtrack(int* calories, int index, int k, int t, int currentSum, int count, int n) {
    // 如果已经选择了k个运动且卡路里和为t，找到一种方案
    if (count == k && currentSum == t) {
        return 1;
    }

    // 如果运动已经考虑完，或者已经选了k个运动，或者当前和已经超过t，无法构成有效方案
    if (index == n || count == k || currentSum > t) {
        return 0;
    }

    // 选择当前运动
    int pickCurrent = backtrack(calories, index + 1, k, t, currentSum + calories[index], count + 1, n);

    // 不选择当前运动
    int skipCurrent = backtrack(calories, index + 1, k, t, currentSum, count, n);

    // 返回两种选择的总方案数
    return pickCurrent + skipCurrent;
}

int main() {
    int n, t, k;

    // 读取输入：n, t, k
    scanf("%d %d %d", &n, &t, &k);
```

```

    // 创建并读取每个运动的卡路里
    int* calories = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        scanf("%d", &calories[i]);
    }

    // 使用回溯法求解
    int result = backtrack(calories, 0, k, t, 0, 0, n);
    printf("%d\n", result);

    // 释放内存
    free(calories);

    return 0;
}


#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    int n, t, k;

    // 读取输入：n, t, k
    scanf("%d %d %d", &n, &t, &k);

    // 读取每个运动的卡路里
    int calories[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &calories[i]);
    }

    // 创建二维DP数组
    // dp[j][m] 表示选择j个运动，总卡路里为m的方案数
    int dp[k+1][t+1];

    // 初始化DP数组全部为0
    memset(dp, 0, sizeof(dp));

    // 初始化：选0个运动，卡路里为0的方案数为1
    dp[0][0] = 1;

    // 填充DP表格
    for (int i = 0; i < n; i++) {
        int cal = calories[i]; // 当前运动的卡路里
```

```
        // 需要从后往前遍历，避免重复计算
        for (int j = k; j >= 1; j--) {
            for (int m = t; m >= cal; m--) {
                dp[j][m] += dp[j-1][m-cal];
            }
        }
    }

    printf("%d\n", dp[k][t]);

    return 0;
}
```

完整用例

用例1

Plain Text

```
4 3 2
1 1 2 3
```

用例2

Plain Text

```
5 5 3
1 2 2 3 4
```

用例3

Plain Text

```
3 6 2
2 2 2
```

用例4

Plain Text

```
4 10 2
5 5 5 5
```

用例5

Plain Text

```
5 10 3
1 2 3 4 5
```

用例6

Plain Text

```
4 2 1
2 3 4 5
```

用例7

Plain Text

```
5 6 3
1 1 2 3 4
```

用例8

Plain Text

```
2 3 2
1 2
```

用例9

Plain Text

```
4 8 3
4 2 2 1
```

用例10

Plain Text

```
5 15 5
3 3 3 3 3
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
    - 备注
- 输出描述
- 示例1
- 解题思路
    - 暴力回溯
  - 动态规划
    - 解题思路
- Java
- Python
- JavaScript
- C++
- C语言
  - 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4
  - 用例5
  - 用例6
  - 用例7
  - 用例8
  - 用例9
  - 用例10

> 来自: 华为OD机考2025B卷 - 小明减肥（Java & Python& JS & C++ & C ）_c++小明有n个可选运动,每个运动有对应卡路里,想选出其中k个运动且卡路里和为t。k,-CSDN博客

华为OD机考2025B卷 - 字符串加密（Java & Python& JS & C++ & C ）_华为od2025b卷 *?字符串-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看

题目描述

给你一串未加密的字符串str，通过对字符串的每一个字母进行改变来实现加密，加密方式是在每一个字母str[i]偏移特定数组元素a[i]的量，数组a前三位已经赋值：a[0]=1,a[1]=2,a[2]=4。
当i>=3时，数组元素a[i]=a[i-1]+a[i-2]+a[i-3]。
例如：原文 abcde 加密后 bdgkr，其中偏移量分别是1,2,4,7,13。

输入描述

第一行为一个整数n（1<=n<=1000），表示有n组测试数据，每组数据包含一行，原文str（只含有小写字母，0<长度<=50）。

输出描述

每组测试数据输出一行，表示字符串的密文。

用例

输入	`1` `xy`
输出	ya
说明	第一个字符x偏移量是1，即为y，第二个字符y偏移量是2，即为a。

解题思路

这道题要求实现一个加密算法，通过对字符串中的每个字母进行特定偏移量的位移来加密。关键点在于偏移量数组的计算规则和字符的偏移处理。

1. **计算偏移量数组**：

- 数组前三个元素固定：a[0]=1, a[1]=2, a[2]=4
- 之后的元素满足递推关系：a[i] = a[i-1] + a[i-2] + a[i-3]（当i≥3时）
- 这是一个类似斐波那契的递推序列

2. **字符偏移处理**：
    - 对字符串中的每个字符，使用对应位置的偏移量进行位移
    - 由于要在小写字母范围内循环，需要进行模26运算

Plain Text

```
#include <iostream>
#include <vector>
using namespace std;

// 计算数组a
vector<long long> calculateA(int n) {
    vector<long long> a(n);
    if (n > 0) a[0] = 1;
    if (n > 1) a[1] = 2;
    if (n > 2) a[2] = 4;
    if (n > 3) {
        for (int i = 3; i < n; i++) {
            a[i] = a[i - 1] + a[i - 2] + a[i - 3];
        }
    }
    return a;
}

int main() {
    int n;
    cin >> n;

    // 读入n个字符串
    vector<string> strArr(n);
    for (int i = 0; i < n; i++) {
        cin >> strArr[i];
    }

    // 对每个字符串进行加密并输出
    for (int i = 0; i < n; i++) {
        int strLength = strArr[i].length();
        vector<long long> a = calculateA(strLength);
        string encryptedStr = "";
        for (int j = 0; j < strLength; j++) {
            encryptedStr += (char) ((a[j] + strArr[i][j] - 97) % 26 + 97);
        }
        cout << encryptedStr << endl;
    }

    return 0;
}
```

java

Java

```java
import java.util.Scanner;

public class Main {
    // 计算数组a
    public static long[] calculateA(int n) {
        long[] a = new long[n];
        if (n > 0) a[0] = 1;
        if (n > 1) a[1] = 2;
        if (n > 2) a[2] = 4;
        if (n > 3) {
            for (int i = 3; i < n; i++) {
                a[i] = a[i - 1] + a[i - 2] + a[i - 3];
            }
        }
        return a;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();

        // 读入n个字符串
        String[] strArr = new String[n];
        for (int i = 0; i < n; i++) {
            strArr[i] = scanner.next();
        }

        // 对每个字符串进行加密并输出
        for (int i = 0; i < n; i++) {
            int strLength = strArr[i].length();
            long[] a = calculateA(strLength);
            StringBuilder encryptedStr = new StringBuilder();
            for (int j = 0; j < strLength; j++) {
                encryptedStr.append((char) ((a[j] + strArr[i].charAt(j) - 97) % 26 + 97));
            }
            System.out.println(encryptedStr.toString());
        }
    }
}
```

javaScript

Plain Text

```
const readline = require('readline');

// 计算数组a
function calculateA(n) {
    const a = new Array(n);
    if (n > 0) a[0] = 1;
    if (n > 1) a[1] = 2;
    if (n > 2) a[2] = 4;
    if (n > 3) {
        for (let i = 3; i < n; i++) {
            a[i] = a[i - 1] + a[i - 2] + a[i - 3];
        }
    }
    return a;
}

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
rl.on('line', (n) => {
  const strArr = [];
  let count = 0;
  rl.on('line', (line) => {
    strArr.push(line.trim());
    count++;
    if (count === n) {
      rl.close();
    }
  });

  rl.on('close', () => {
    // 对每个字符串进行加密并输出

      // 对每个字符串进行加密并输出
        for (let i = 0; i < n; i++) {
            const strLength = strArr[i].length;
            const a = calculateA(strLength);
            let encryptedStr = '';
            for (let j = 0; j < strLength; j++) {
                encryptedStr += String.fromCharCode((a[j] + strArr[i].charCodeAt(j) - 97) % 26 + 97);
            }
            console.log(encryptedStr);
        }
```

```
    process.exit(0);
  });
});
```

python

```
def calculateA(n):
    a = [0] * n
    if n > 0:
        a[0] = 1
    if n > 1:
        a[1] = 2
    if n > 2:
        a[2] = 4
    if n > 3:
        for i in range(3, n):
            a[i] = a[i - 1] + a[i - 2] + a[i - 3]
    return a

n = int(input())

strArr = []
for i in range(n):
    strArr.append(input())

for i in range(n):
    strLength = len(strArr[i])
    a = calculateA(strLength)
    encryptedStr = ""
    for j in range(strLength):
        encryptedStr += chr((a[j] + ord(strArr[i][j]) - 97) % 26 + 97)
    print(encryptedStr)
```

C语言

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 计算数组a
long* calculateA(int n) {
    long* a = (long*)malloc(n * sizeof(long));
    if (n > 0) a[0] = 1;
    if (n > 1) a[1] = 2;
    if (n > 2) a[2] = 4;
    if (n > 3) {
        for (int i = 3; i < n; i++) {
            a[i] = a[i - 1] + a[i - 2] + a[i - 3];
        }
    }
    return a;
}

int main() {
    int n;
    scanf("%d", &n);

    // 读入n个字符串
    char strArr[50][51]; // 最多50个字符串，每个字符串最长50个字符，加1用于存储字符串结束符'\0'
    for (int i = 0; i < n; i++) {
        scanf("%s", strArr[i]);
    }

    // 对每个字符串进行加密并输出
    for (int i = 0; i < n; i++) {
        int strLength = strlen(strArr[i]);
        long* a = calculateA(strLength);

        char encryptedStr[51]; // 加密后的字符串
        for (int j = 0; j < strLength; j++) {
            encryptedStr[j] = (char)((a[j] + strArr[i][j] - 97) % 26 + 97);
        }
        encryptedStr[strLength] = '\0'; // 字符串结束符

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

        free(a); // 释放动态分配的内存
    }
```

```
    return 0;
}
```

完整用例

用例1

Plain Text

```
1
xy
```

用例2

Plain Text

```
1
abc
```

用例3

Plain Text

```
1
zzz
```

用例4

```
1
def
```

用例5

```
1
pqr
```

用例6

```
1
abcdefghijklmnopqrstuvwxyz
```

用例7

```
1
aaaaaaaabbbbbbbbcccccccc
```

用例8 极端用例，用来验证代码鲁棒性

```
1
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

> 来自：华为OD机考2025B卷 – 字符串加密（Java & Python& JS & C++ & C ）_华为od2025b卷 *?字符串-CSDN博客

华为OD机考2025B卷 - 无向图染色（Java & Python& JS & C++ & C ）_华为od机试真题 python 实现【无向图染色】-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看

题目描述

给一个无向图染色，可以填红黑两种颜色，必须保证相邻两个节点不能同时为红色，输出有多少种不同的染色方案?

输入描述

第一行输入M(图中节点数) N(边数)
后续N行格式为：V1 V2表示一个V1到V2的边。
数据范围：1 <= M <= 15,0 <= N <= M * 3，不能保证所有节点都是连通的。

说明

0 < n < 15
0 <= m <= n * 3
0 <= s, t < n
不保证图连通
保证没有重边和自环

输出描述

输出一个数字表示染色方案的个数。

示例1

输入

```
4 4
1 2
2 4
3 4
1 3
```

输出

```
7
```

说明

> 4个节点，4条边，1号节点和2号节点相连，
> 2号节点和4号节点相连，3号节点和4号节点相连，
> 1号节点和3号节点相连，
> 若想必须保证相邻两个节点不能同时为红色，总共7种方案。

示例2

输入

```
3 3
1 2
1 3
2 3
```

输出

```
4
```

说明

解题思路

要解决这个图的染色问题，我们可以利用深度优先搜索（DFS）进行递归遍历每一个节点，对于每个节点，我们可以选择将其染为红色或黑色，同时要注意相邻节点的限制条件。

步骤分析：

1. **输入处理：**
   读取节点数 `M` 和边数 `N`。然后将接下来的 `N` 行表示为无向边存储在一个二维布尔数组中，这个数组可以表示图的邻接矩阵。
2. **DFS 递归函数设计：**
   - 定义一个递归函数 `dfs`，它接受当前节点索引、上次染红的节点（通过位运算表示），以及图的邻接矩阵作为参数。
   - 如果当前节点的索引超过 `M`，说明染色完成，返回1（表示成功的一种方案）。
3. **染色决策：**
   - **染黑色**: 在每次调用 DFS 时，首先尝试将当前节点染色为黑色，不需要进行任何检查，直接继续到下一个节点。
   - **染红色**: 对于染红色的情况，需要检查当前节点的邻接节点是否已经被染为红色。通过位运算判断当前节点的邻接节点的状态。
     - 通过检查邻接矩阵和红色节点的状态，确定当前节点是否可以染红。
     - 如果可以染红，将其状态更新，调用 recursively 继续下一节点。
4. **累积方案数**:
   - 每次成功染完所有节点后，返回 1；每种染色方案会被累积到结果 `count` 中。
5. **输出结果**:
   - 最后输出全部染色方案的数量。

特别注意：

- 利用位运算 `redCount | (1 << index)` 来记录当前节点染红之后的红色节点状态，这种方法在处理小规模问题时时效性更高。
- 由于图可能不连通，所以通过 `index` 更加灵活地遍历每一个节点，即使某些节点未被关联的情况。

Java

Java

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int nodeNum = sc.nextInt();
        int edgeNum = sc.nextInt();

        boolean[][] edges = new boolean[nodeNum + 1][nodeNum + 1];
        for (int i = 0; i < edgeNum; i++) {
            int node1 = sc.nextInt();
            int node2 = sc.nextInt();
            edges[node1][node2] = true;
            edges[node2][node1] = true;
        }

        System.out.println(dfs(edges, nodeNum, 1, 0));
    }

    public static int dfs(boolean[][] edges, int nodeNum, int index, int redCount) {
        if (index > nodeNum) {
            return 1;
        }

        int count = 0;

        // 当前节点染黑色
        count += dfs(edges, nodeNum, index + 1, redCount);

        // 检查当前节点是否可以染红色
        boolean canBeRed = true;
        for (int i = 1; i < index; i++) {
            if (edges[index][i] && (redCount & (1 << i)) != 0) {
                canBeRed = false;
                break;
            }
        }

        // 当前节点染红色
        if (canBeRed) {
            count += dfs(edges, nodeNum, index + 1, redCount | (1 << index));
        }
```

```
44
45
            return count;
46      }
47  }
```

Plain Text

```
#include <stdio.h>
#include <stdbool.h>

int dfs(bool edges[][101], int nodeNum, int index, int redCount);

int main() {
    int nodeNum, edgeNum;
    scanf("%d %d", &nodeNum, &edgeNum);

    bool edges[101][101] = {false};

    for (int i = 0; i < edgeNum; i++) {
        int node1, node2;
        scanf("%d %d", &node1, &node2);
        edges[node1][node2] = true;
        edges[node2][node1] = true;
    }

    printf("%d\n", dfs(edges, nodeNum, 1, 0));

    return 0;
}

int dfs(bool edges[][101], int nodeNum, int index, int redCount) {
    if (index > nodeNum) {
        return 1;
    }

    int count = 0;

    // 当前节点染黑色
    count += dfs(edges, nodeNum, index + 1, redCount);

    // 检查当前节点是否可以染红色
    bool canBeRed = true;
    for (int i = 1; i < index; i++) {
        if (edges[index][i] && (redCount & (1 << i)) != 0) {
            canBeRed = false;
            break;
        }
    }

    // 当前节点染红色
    if (canBeRed) {
        count += dfs(edges, nodeNum, index + 1, redCount | (1 << index));
```

```
        }

        return count;
    }
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
  - 说明
- 输出描述
- 示例1
- 示例2
- 解题思路
  - 步骤分析:
  - 特别注意:
- Java
- Python
- JavaScript
- C++
- C语言

> 来自: 华为OD机考2025B卷 - 无向图染色（Java & Python& JS & C++ & C ）_华为od机试真题 python 实现【无向图染色】-CSDN博客

华为OD机考2025B卷 - 差值数组不同的字符串（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

给你一个字符串数组 `words` ，每一个字符串长度都相同，令所有字符串的长度都为 `n` 。
每个字符串 `words[i]` 可以被转化为一个长度为 `n - 1` 的 **差值整数数组** `difference[i]` ，其中对于 `0 <= j <= n - 2` 有 `difference[i][j] = words[i][j+1] - words[i][j]` 。
注意两个字母的差值定义为它们在字母表中 **位置** 之差，也就是说 `'a'` 的位置是 `0` ，`'b'` 的位置是 `1` ，`'z'` 的位置是 `25` 。

- 比方说，字符串 `"acb"` 的差值整数数组是 `[2 - 0, 1 - 2] = [2, -1]` 。

`words` 中所有字符串 **除了一个字符串以外** ，其他字符串的差值整数数组都相同。你需要找到那个不同的字符串。
输出 `words` 中 **差值整数数组** 不同的字符串。

输入描述

第一行输入一个字符串，包含两个数字，空格分隔，数组长度m和数字n，n表示字符串的长度
第二行输入一个字符串，m个长度为n的字符串，空格分隔

备注

- `3 <= n <= 100`
- `2 <= m <= 20`
- `words[i]` 只含有小写英文字母。

输出描述

请返回 words中 差值整数数组不同的字符串。

示例1

输入

```
3 4
abcd bcde cdeg
```

输出

```
cdeg
```

说明

> bcde 和 cdef 的差值数组为 [1, 1, 1]，abcf 的为 [1, 1, 3]。

示例2

输入

```
4 2
ab cd ef ac
```

输出

```
ac
```

说明

> ab, cd, ef 的差值数组为 [1]，ac 的为 [2]。

解题思路

1. **理解题目需求**：题目要求从给定的字符串数组中找出一个特殊字符串，其差值整数数组（字符之间的差值序列）与其他字符串不同。通过字符的ASCII码差值计算生成的数组便是该字母的差值整数数组。
2. **差值数组的计算**：
   - 对于每个字符串，通过遍历相邻的字符，计算其在字母表中的差值并存储在一个差值数组中。具体来说，可以通过ASCII码相减来获取字符的差值。例如字符 `'b'` 的ASCII码是98，`'a'` 的ASCII码是97，因此 `'b' - 'a'` 的差值为1。
3. **使用映射来记录类型**：
   - 由于多个字符串可能产生相同的差值数组，可以用一个哈希表（`Map`）来存储每种差值数组及其

出现的字符串索引。这个哈希表的键是差值数组的字符串表示，值是对应的字符串索引列表。这样可以方便地查找与其他字符串不同的字符串。

4. **处理和查找不同的字符串**：
    - 在所有字符串都处理完后，通过遍历这个哈希表，查找那些只出现一次的差值模式，若找到，则可以确定这就是那个不同的字符串。

Java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取输入参数: m-字符串数量, n-每个字符串的长度
        String[] firstLine = scanner.nextLine().split(" ");
        int m = Integer.parseInt(firstLine[0]);
        int n = Integer.parseInt(firstLine[1]);

        // 读取所有字符串
        String[] words = scanner.nextLine().split(" ");

        // 存储差值模式 -> 对应字符串索引列表的映射
        // 关键思想: 相同类型的字符串会有相同的字符间差值模式
        Map<String, List<Integer>> diffPatternCount = new HashMap<>();

        // 遍历每个字符串, 计算其差值模式
        for (int i = 0; i < m; i++) {
            String word = words[i];
            int[] diffArray = new int[n - 1];

            // 计算相邻字符的ASCII差值
            for (int j = 0; j < n - 1; j++) {
                diffArray[j] = word.charAt(j + 1) - word.charAt(j);
            }

            // 将差值数组转换为字符串形式作为模式标识
            String diffPattern = Arrays.toString(diffArray);

            // 将当前字符串索引添加到对应模式的列表中
            diffPatternCount.putIfAbsent(diffPattern, new ArrayList<>());
            diffPatternCount.get(diffPattern).add(i);
        }

        // 找出只出现一次的差值模式对应的字符串(即异常字符串)
        String result = "";
        for (Map.Entry<String, List<Integer>> entry : diffPatternCount.entrySet()) {
            if (entry.getValue().size() == 1) {
                result = words[entry.getValue().get(0)];
                break;
            }
        }
```

```
45
46
            // 输出结果并关闭资源
47          System.out.println(result);
48          scanner.close();
49      }
50  }
```

Plain Text

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_WORDS 100
#define MAX_LENGTH 100

// 用于存储差值模式和对应的字符串索引
typedef struct {
    char diffPattern[MAX_LENGTH];
    int indices[MAX_WORDS];
    int count;
} DiffPattern;

int main() {
    int m, n;
    char words[MAX_WORDS][MAX_LENGTH];

    // 读取输入参数: m-字符串数量, n-每个字符串的长度
    if (scanf("%d %d", &m, &n) != 2) {
        printf("请正确输入代码片段。\n");
        return 1;
    }

    // 读取所有字符串
    for (int i = 0; i < m; i++) {
        scanf("%s", words[i]);
    }

    DiffPattern diffPatternCount[MAX_WORDS];
    int diffPatternCountSize = 0;

    // 遍历每个字符串，计算其差值模式
    for (int i = 0; i < m; i++) {
        int diffArray[MAX_LENGTH];

        // 计算相邻字符的ASCII差值
        for (int j = 0; j < n - 1; j++) {
            diffArray[j] = words[i][j + 1] - words[i][j];
        }

        // 将差值数组转换为字符串形式作为模式标识
        char diffPattern[MAX_LENGTH];
        char temp[10];
        strcpy(diffPattern, "[");
```

```
        for (int j = 0; j < n - 1; j++) {
            sprintf(temp, "%d", diffArray[j]);
            strcat(diffPattern, temp);
            if (j < n - 2) {
                strcat(diffPattern, ", ");
            }
        }
        strcat(diffPattern, "]");

        // 在模式计数中查找或添加当前模式
        int found = 0;
        for (int k = 0; k < diffPatternCountSize; k++) {
            if (strcmp(diffPatternCount[k].diffPattern, diffPattern) == 0) {
                diffPatternCount[k].indices[diffPatternCount[k].count++] = i;
                found = 1;
                break;
            }
        }
        if (!found) {
            strcpy(diffPatternCount[diffPatternCountSize].diffPattern, diffPattern);
            diffPatternCount[diffPatternCountSize].indices[0] = i;
            diffPatternCount[diffPatternCountSize].count = 1;
            diffPatternCountSize++;
        }
    }

    // 找出只出现一次的差值模式对应的字符串(即异常字符串)
    char result[MAX_LENGTH] = "";
    for (int k = 0; k < diffPatternCountSize; k++) {
        if (diffPatternCount[k].count == 1) {
            strcpy(result, words[diffPatternCount[k].indices[0]]);
            break;
        }
    }

    // 输出结果
    if (strlen(result) > 0) {
        printf("%s\n", result);
    } else {
        printf("没有找到异常字符串。\n");
    }

    return 0;
}
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
    - 备注
- 输出描述
- 示例1
- 示例2
- 解题思路
- Java
- Python
- JavaScript
- C++
- C语言

> 来自：华为OD机考2025B卷 - 差值数组不同的字符串（Java & Python& JS & C++ & C ）-CSDN博客

华为OD机考2025B卷 - 货币单位换算（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看

题目描述

记账本上记录了若干条多国货币金额，需要转换成人民币分（fen），汇总后输出。
每行记录一条金额，金额带有货币单位，格式为数字+单位，可能是单独元，或者单独分，或者元与分的组合。
要求将这些货币全部换算成人民币分（fen）后进行汇总，汇总结果仅保留整数，小数部分舍弃。
元和分的换算关系都是1:100，如下：

- 1CNY=100fen（1元=100分）
- 1HKD=100cents（1港元=100港分）
- 1JPY=100sen（1日元=100仙）
- 1EUR=100eurocents（1欧元=100欧分）
- 1GBP=100pence（1英镑=100便士）

汇率表如下：
即：100CNY = 1825JPY = 123HKD = 14EUR = 12GBP

CNY	JPY	HKD	EUR	GBP
100	1825	123	14	12

输入描述

第一行输入为N，N表示记录数。0<N<100
之后N行，每行表示一条货币记录，且该行只会是一种货币。

输出描述

将每行货币转换成人民币分（fen）后汇总求和，只保留整数部分。
输出格式只有整数数字，不带小数，不带单位。

示例1

输入

```
1
100CNY
```

输出

```
10000
```

说明

> 100CNY转换后是10000fen，所以输出结果为10000

示例2

输入

```
1
3000fen
```

输出

```
3000
```

说明

> 3000fen，结果就是3000

示例3

输入

```
1
123HKD
```

输出

```
10000
```

说明

> HKD与CNY的汇率关系是123:100，所以换算后，输出结果为10000

示例4

输入

```
2
20CNY53fen
53HKD87cents
```

输出

```
6432
```

说明

> 20元53分 + 53港元87港分，换算成人民币分后汇总，为6432

解题思路

这题主要是拆解字符串，题目本身不难，按照题意写好分支判断即可。如果觉得代码复杂，可以考虑使用正则表达式来做。

Java

```java
import java.util.Scanner;

public class CurrencyConverter {

    // 方法：根据货币单位返回其转换为人民币分的汇率
    public static double exChange(String unit) {
        switch (unit) {
            case "CNY":
                return 100.0; // 人民币
            case "JPY":
                return 100.0 / 1825 * 100; // 日元
            case "HKD":
                return 100.0 / 123 * 100; // 港元
            case "EUR":
                return 100.0 / 14 * 100; // 欧元
            case "GBP":
                return 100.0 / 12 * 100; // 英镑
            case "fen":
                return 1.0; // 人民币分
            case "cents":
                return 100.0 / 123; // 港元分
            case "sen":
                return 100.0 / 1825; // 日元分
            case "eurocents":
                return 100.0 / 14; // 欧元分
            case "pence":
                return 100.0 / 12; // 英镑分
            default:
                return 0.0; // 无效单位返回0
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // 创建Scanner对象用于读取输入
        int n = Integer.parseInt(scanner.nextLine()); // 读取记录数


        double totalFen = 0; // 汇总结果

        // 处理每一条货币记录
        for (int i = 0; i < n; i++) {
            String record = scanner.nextLine(); // 读取每一行的记录
            int amount = 0; // 用于保存金额
            StringBuilder unit = new StringBuilder(); // 保存单位
```

```java

            // 遍历当前行，逐个提取金额和单位
            for (int j = 0; j < record.length(); j++) {
                char c = record.charAt(j);
                if (Character.isDigit(c)) {
                    amount = amount * 10 + (c - '0'); // 构建数字
                } else {
                    unit.append(c); // 构建货币单位
                }

                // 当遇到完整的金额+单位时进行换算
                if (j == record.length() - 1 || Character.isDigit(record.charAt(j + 1)) && unit.length() > 0) {
                    totalFen += amount * exChange(unit.toString()); // 计算并累加到总数
                    amount = 0; // 重置金额
                    unit.setLength(0); // 清空单位
                }
            }
        }

        // 输出汇总结果，只保留整数部分
        System.out.println((int) totalFen);
        scanner.close(); // 关闭Scanner对象
    }
}
```

整用例

用例1

Plain Text

```
1
100CNY
```

用例2

Plain Text

```
1
3000fen
```

用例3

```
1
123HKD
```

用例4

```
2
20CNY53fen
53HKD87cents
```

用例5

```
2
100CNY
100JPY
```

用例6

```
3
50EUR
25GBP
1000fen
```

用例7

```
4
5000sen
1000cents
200eurocents
300pence
```

用例8

```
3
100HKD
200JPY
50EUR
```

用例9

```
2
1000pence
5000eurocents
```

用例10

```
4
10CNY
20JPY
30HKD
40EUR
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- 示例2
- 示例3
- 示例4
- 解题思路
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例

- 用例1
- 用例2
- 用例3
- 用例4
- 用例5
- 用例6
- 用例7
- 用例8
- 用例9
- 用例10

> 来自: 华为OD机考2025B卷 – 货币单位换算（Java & Python& JS & C++ & C ）–CSDN博客

华为OD机考2025B卷 - 最长的顺子（Java & Python& JS & C++ & C ）-CSDN博客

最新华为OD机试

真题目录：点击查看目录
华为OD面试真题精选：点击立即查看
2025华为od 机试2025B卷-华为机考OD2025年B卷

题目描述

斗地主起源于湖北十堰房县，据说是一位叫吴修全的年轻人根据当地流行的扑克玩法“跑得快”改编的，如今已风靡整个中国，并流行于互联网上。
牌型：单顺，又称顺子，最少5张牌，最多12张牌(3...A)不能有2，也不能有大小王，不计花色。
例如： 3-4-5-6-7-8，7-8-9-10-J-Q，3-4-5-6-7-8-9-10-J-Q-K-A
可用的牌 3<4<5<6<7<8<9<10<J<Q<K<A<2<B(小王)<C(大王)，每种牌除大小王外有四种花色(共有13×4+2张牌)
输入：

1. 手上有的牌
2. 已经出过的牌(包括对手出的和自己出的牌)

输出：

- 对手可能构成的最长的顺子(如果有相同长度的顺子，输出牌面最大的那一个)，
- 如果无法构成顺子，则输出 NO-CHAIN。

输入描述

输入的第一行为当前手中的牌
输入的第二行为已经出过的牌

输出描述

示例1

输入

```plain text
3-3-3-3-4-4-5-5-6-7-8-9-10-J-Q-K-A
4-5-6-7-8-8-8
```

输出

```
9-10-J-Q-K-A
```

说明

> 无

示例2

输入

```
3-3-3-3-8-8-8-8
K-K-K-K
```

输出

```
NO-CHAIN
```

说明

> 剩余的牌无法构成顺子

解题思路

1. **问题分析**：
   - 题目要求从当前手牌和已经出过的牌中找出对手可能构成的最长顺子。顺子是由连续的 5 张及以上的牌组成，且不包括 2 和大小王。
   - 如果能够找到多个顺子，则返回最长的顺子，如果多个顺子的长度相同，则返回字面值最大的顺子。
   - 如果无法构成顺子，输出“NO-CHAIN”。
2. **牌的表示和转换**：
   - 题目中定义了 3 至 A 的顺子，牌面为 `3-4-5-6-7-8-9-10-J-Q-K-A`，需要将这些牌映射为数字，以便于处理和排序。
   - 另外，题目中还出现了 J（11）、Q（12）、K（13）、A（14）等特殊字符，需要将其映射为对应的数字。
3. **思路步骤**：
   - **步骤 1**：将输入的牌转换为数字形式，并记录每种牌的数量。
     - 构建一个数组 `cardCount`，其中每个索引对应一个牌面值，记录其剩余的牌数。

    - 通过从手牌和已出牌中减去已用掉的牌，得到当前对手剩余的牌。
  - **步骤 2**：查找对手的顺子。
    - 遍历 `cardCount`，从数字 3 到 A （对应 3 到 14）查找是否能组成顺子。
    - 如果连续的牌数超过或等于 5 张，就可以形成一个顺子。
    - 保存所有找到的顺子。
  - **步骤 3**：排序顺子。
    - 按照顺子的长度进行降序排序，如果多个顺子的长度相同，则按照字面值进行降序排序。
  - **步骤 4**：返回结果。
    - 如果找到顺子，返回字面值最大的顺子；如果没有找到，返回“NO-CHAIN”。

Java

```java
import java.util.*;

public class Main {
    // 将牌面映射为数字，方便处理
    public static String mapToNumber(String pai) {
        if (pai.equals("J")) return "11";
        if (pai.equals("11")) return "J";
        if (pai.equals("Q")) return "12";
        if (pai.equals("12")) return "Q";
        if (pai.equals("K")) return "13";
        if (pai.equals("13")) return "K";
        if (pai.equals("A")) return "14";
        if (pai.equals("14")) return "A";
        return pai;
    }

    public static String getLongestChain(List<String> myCards, List<String> usedCards) {
        // 统计3~A的每张牌的数量，对应数组的索引3~14
        int[] cardCount = {0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 0};

        // 从cardCount中去除myCards，剩下的就是对手的牌及数量
        for (String card : myCards) {
            int idx = Integer.parseInt(mapToNumber(card));
            cardCount[idx]--;
        }

        // 从cardCount中去除usedCards，剩下的就是对手的牌及数量
        for (String card : usedCards) {
            int idx = Integer.parseInt(mapToNumber(card));
            cardCount[idx]--;
        }

        // res用于保存顺子
        List<String> res = new ArrayList<>();
        // path用于保存连续的牌
        List<String> path = new ArrayList<>();

        for (int i = 3; i < cardCount.length; i++) {
            // 从牌面3开始遍历，如果牌数量大于0，则加入path
            if (cardCount[i] > 0) {
                path.add(String.valueOf(i));
            } else {

```

```java
                // 如果遇到牌数量为0的，则连续的牌被中断，若此时连续的牌数量大于等于5，则可以形成顺子，我们将顺子加入res中
                if (path.size() >= 5) {
                    StringBuilder sb = new StringBuilder();
                    for (String idx : path) {
                        String pai = mapToNumber(idx);
                        sb.append(pai).append("-");
                    }
                    sb.deleteCharAt(sb.length() - 1); // 去掉最后一个"-"
                    res.add(sb.toString());
                }
                // 清空path，为下一次连续的牌腾位置
                path.clear();
            }
        }

        // 如果没有顺子，则返回"NO-CHAIN"
        if (res.size() == 0) return "NO-CHAIN";

        // 如果有顺子，则先按照顺子长度升序，若长度相等，则按照字典序升序
        Collections.sort(res, new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                return a.length() != b.length() ? a.length() - b.length() : a.compareTo(b);
            }
        });

        return res.get(0);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<String> myCards = new ArrayList<>();
        List<String> usedCards = new ArrayList<>();
        String line = scanner.nextLine();
        String[] tokens = line.split("-");
        for (String token : tokens) {
            myCards.add(token);
        }

        line = scanner.nextLine();
        tokens = line.split("-");
        for (String token : tokens) {
            usedCards.add(token);
        }

        System.out.println(getLongestChain(myCards, usedCards));
```

```
        }
    }
```

完整用例

用例1

```
3-3-3-3-4-4-5-5-6-7-8-9-10-J-Q-K-A
4-5-6-7-8-8-8
```

用例2

```
3-3-3-3-8-8-8-8
K-K-K-K
```

用例3

```
3-3-3-3-4-4-5-5-6-7-8-9-10-J-Q-K-A
4-5-6-7-8-8-8-9-10-J-Q-K-A
```

用例4

```
3-3-3-3-4-4-5-5-6-7-8-9-10-J-Q-K-A
4-5-6-7-8-8-8-9-9-9-10-J
```

用例5

```
3-4-5-6-7-8-9-10-J-Q-K-A
2-3-4-5-6-7-8-9-10-J-Q-K-A
```

用例6

```
3-4-5-6-7-8-9-10-J-Q-K-A
2-2-2-2-3-3-3-J-J-J
```

用例7

```
3-3-3-3-9-9-9-9
K-K-K-K
```

用例8

```
3-4-5-6-7-8-9-10-J-Q-K-A
3-4-5-6-7-8-9-10-J-Q-K-A
```

用例9

```
3-4-5-6-7-8-9-10-J-Q-K-A
A-2-3-4-5-6-7-8-9-10-J-Q-K
```

用例10

```
2-3-4-5-6-7-8-9-10-J-Q-K-A
2-3-3-3-4-5-6-7-8-9-9-9-10-J-Q-K-A-A-A
```

文章目录

- 最新华为OD机试
- 题目描述
- 输入描述
- 输出描述
- 示例1
- 示例2

- 解题思路
- Java
- Python
- JavaScript
- C++
- C语言
- 完整用例
  - 用例1
  - 用例2
  - 用例3
  - 用例4
  - 用例5
  - 用例6
  - 用例7
  - 用例8
  - 用例9
  - 用例10

> 来自: 华为OD机考2025B卷 – 最长的顺子（Java & Python& JS & C++ & C ）–CSDN博客