蓝桥杯全国软件和信息技术专业人才大赛官方备赛指南

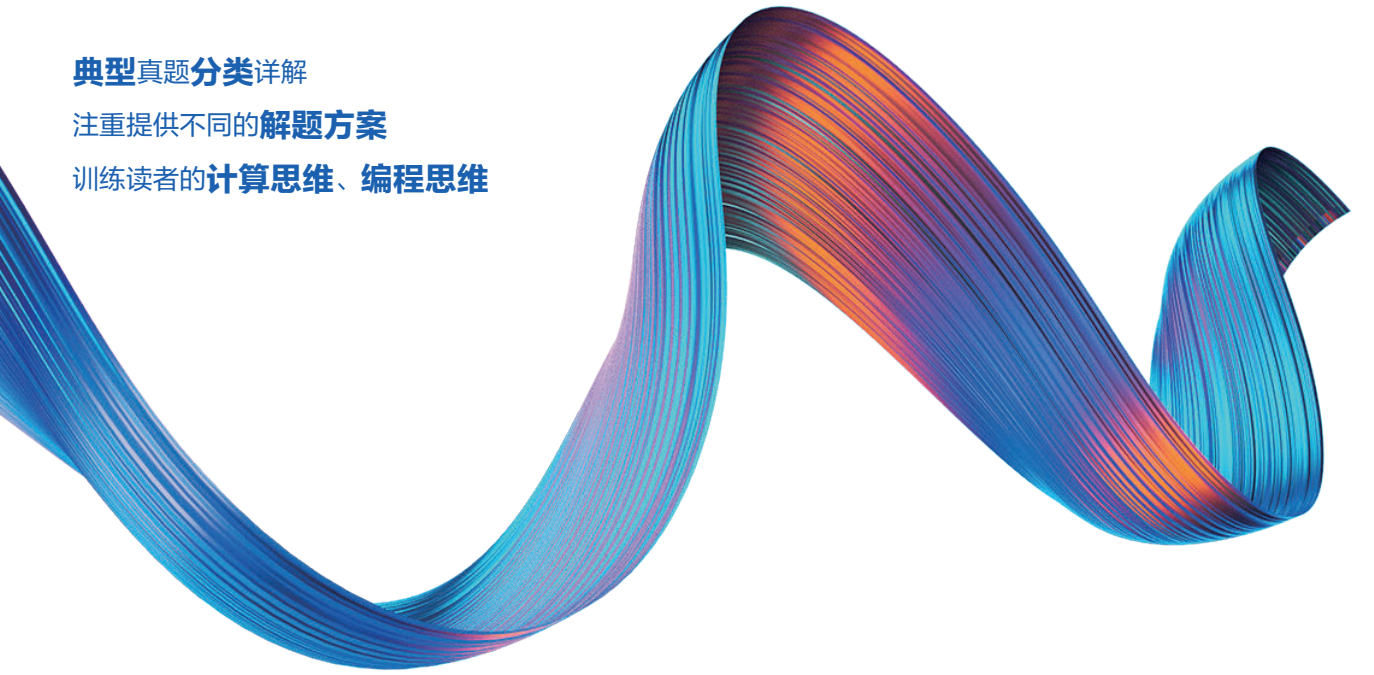
C/C++程序设计竞赛

真题实战特训教程

图解版

蓝桥杯大赛组委会◎组编
张航 唐鑫程 郑未◎编著

典型真题**分类**详解
注重提供不同的**解题方案**
训练读者的**计算思维**、**编程思维**



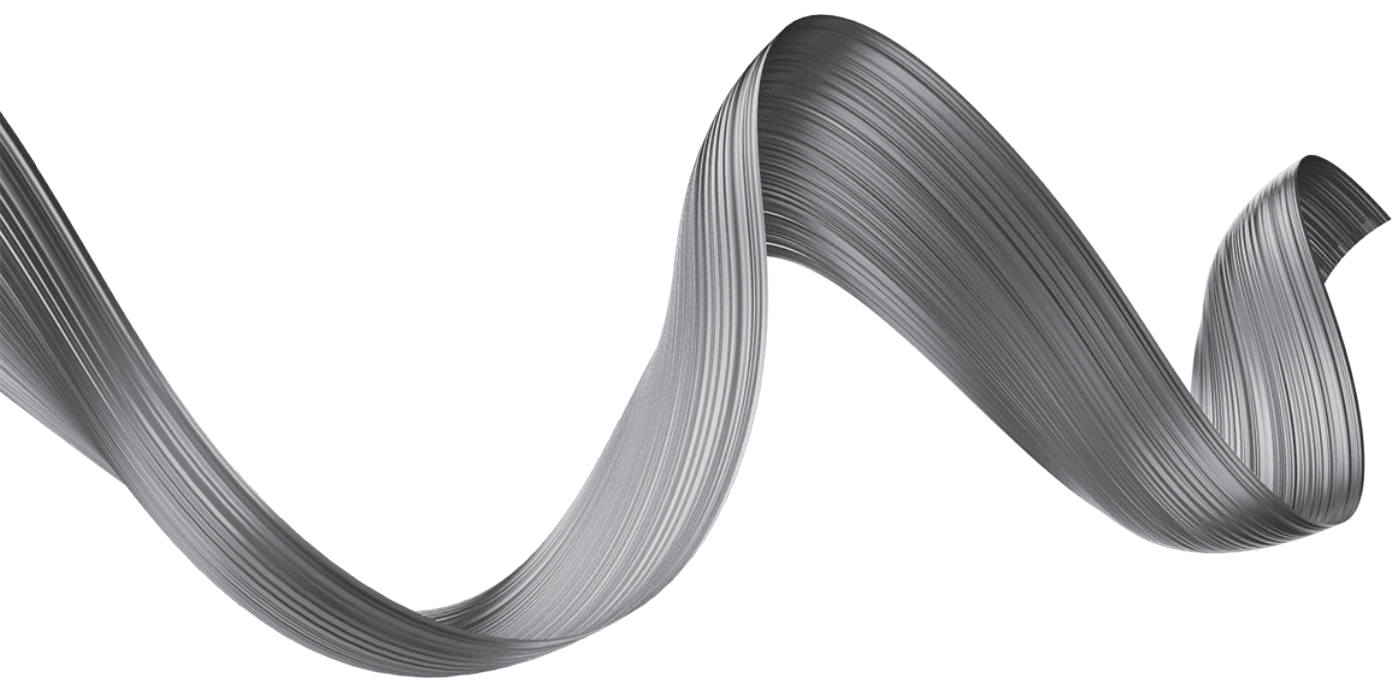
中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

C/C++ 程序设计竞赛

真题实战特训教程

图解版

蓝桥杯大赛组委会◎组编
张航 唐鑫程 郑未◎编著



人民邮电出版社
北京

图书在版编目（CIP）数据

C/C++程序设计竞赛真题实战特训教程 : 图解版 / 蓝桥杯大赛组委会组编 ; 张航, 唐鑫程, 郑未编著. -- 北京 : 人民邮电出版社, 2023.1
ISBN 978-7-115-60135-3

Ⅰ. ①C… Ⅱ. ①蓝… ②张… ③唐… ④郑… Ⅲ. ①C语言－程序设计－习题集 Ⅳ. ①TP312.8-44

中国版本图书馆CIP数据核字(2022)第243649号

内 容 提 要

本书面向蓝桥杯全国软件和信息技术专业人才大赛的软件类赛项（以下简称蓝桥杯软件类大赛）中的C/C++语言组的备赛，从数百道历年真题中精选具有代表性的题目作为例题进行分类详解。

全书共 7 章，由浅入深、由易到难地介绍了各类例题，主要包括枚举与模拟、搜索与查找、思维与贪心、简单数论、字符串算法、动态规划、数据结构等。每一类例题的讲解，不只是简单地给出解析及参考代码，而是注重通过提供不同的解题方案训练读者的计算思维、编程思维，不仅有助于读者提高解题能力和竞赛水平，还有助于形成自己的编程思想，实现“以赛促学”的学习目标。

本书不仅适合作为蓝桥杯软件类大赛 C/C++语言组的备赛用书，还适合作为本科生和研究生相关编程语言课程的教材或参考资料。

◆ 组　　编　蓝桥杯大赛组委会
编　　著　张　航　唐鑫程　郑　未
责任编辑　李　莎
责任印制　王　郁　胡　南

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：14.5　　2023 年 1 月第 1 版
字数：323 千字　　2023 年 1 月北京第 1 次印刷

定价：69.80 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字 20170147 号

写在前面的话

请坚信，你曾艰苦奋斗的日日夜夜，绝不会允许你在比赛中充当“分母”。

“或许我们可以为蓝桥杯出版一本教材！”

这是我们蓝桥杯软件类大赛教研团队在某次工作会议时的结束语，也是本书的起点。

在我们团队过去十几年的发展过程中，无论是教育培训，还是程序开发，我们都得心应手。但对于出版教材这事，我们却是“零经验”。

“出书是一件艰难的事情，因为我们要面对的是一种全新的思维模式。”团队负责人笑着说道，“这也必然是个巨大的挑战！”

截至 2022 年，蓝桥杯软件类大赛（实质上也是算法竞赛）已成为参赛人数最多的大学计算机编程竞赛：1600 余所高校参赛，累计参赛人数超过 60 万人。大赛对大学生综合评测、奖学金评定、升学考研、就业都有一定助益，但是，能用于大赛辅导的教材却寥寥无几，这也是我们出书的主要原因。

怎样才能写好一本书，就成了我们日思夜想的问题。

经过多方几轮的讨论，我们决定从参赛选手的视角，以训练计算思维和逻辑思维为出发点，采用逐级递进的方式细致地讲解真题，带领读者由浅入深地掌握蓝桥杯软件类大赛的核心考点和解题技巧。同时，为了降低读者在阅读过程中的枯燥感，我们几乎在每道真题的讲解中都添加了代码注释、示意图等有助于更好地学习与掌握算法知识、提高解题能力的“助推器”；考虑到难题的解读门槛，我们特地补充了难题的“低分”解法（可通过部分测试数据的解题方法），例如只能拿 20% 分数、40% 分数的解法，以保证具有不同编程水平的人能各取所需、各有所得。

“我们能做的就这么多了，接下来看他们的吧！”在编写工作的收尾期，我们这些创作者都不由自主地发出感叹。

由于参赛人数众多，蓝桥杯软件类大赛堪称“严酷”。我们所能提供的支持已尽在书中，如果你们想在算法竞赛的“星辰大海”中乘风破浪，还要靠自我驱动、奋勇前行。接下来就看你们的了！

下面就先来了解蓝桥杯软件类大赛的相关事宜，以及如何使用本书来学习和备赛。

关于赛制

蓝桥杯软件类大赛可分为省赛和总决赛两级。

1. 省赛

每个组别分别设立一、二、三等奖，原则上各奖项的比例为 10%、20%、30%。获奖比例仅作为参考，组委会专家组将根据赛题难易程度及整体答题情况，制定获奖最低分数线，未

达到获奖最低分数线者不得奖。

2. 国赛

每个组别分别设立一、二、三等奖及优秀奖。其中，一等奖不高于 5%，二等奖占 20%，三等奖不低于 35%，优秀奖不超过 40%，零分卷不得奖。

关于题型

每场蓝桥杯软件类大赛设有 10 道题目，总分为 150 分，竞赛时长为 4 小时。对于同一道题目，参赛选手可多次提交答案，评分时以最后一次提交的答案为准。

参赛选手必须通过浏览器方式提交自己的答案，在其他位置作答或以其他方式提交的答案无效。试题设有 “结果填空” 和 “编程大题” 两种题型。

1. 结果填空

要求参赛选手根据题目描述直接填写结果。求解方式不限，不要求源代码。参赛选手将答案直接通过浏览器提交即可，不要书写多余的内容。

2. 编程大题

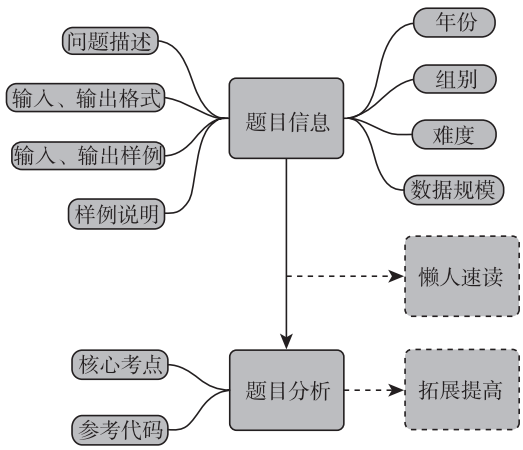
要求参赛选手设计的程序对于给定的输入能给出正确的输出结果。参赛选手所编写的程序只有在能运行出正确结果的情况下才有机会得分。

关于本书的学习

本书适合具有一定编程基础，对蓝桥杯软件类大赛有浓厚兴趣的算法竞赛爱好者及各类高等院校计算机专业的师生阅读。

如前所述，本书在编写时考虑了不同编程水平读者的学习特点和竞赛需求，下面就详细介绍本书所设置的章节结构，以便读者在了解我们的编写思路之后，能更好地应用本书来进行学习和备赛。

本书对于大部分试题及其解答过程的写作结构，主要有题目信息、懒人速读、题目分析、拓展提高这 4 个栏目，其具体构成如下图所示。



（1）题目信息。主要提供两类信息，一类是基本的 “竞赛信息”，例如该试题的所属年份、

组别、难度级别等；另一类是完整的试题信息，包括问题描述，输入、输出格式，输入、输出样例，样例说明，以及数据规模等。

（2）懒人速读。由于部分真题的描述显得烦琐，不利于读者快速把握题目的核心考点，我们通过此栏目带领读者高效地解读题目、准确地领会考查要点，有利于读者节省阅读与理解题意的时间，并能够快速地投入到解题的思考过程中。

（3）题目分析。从参赛选手的角度一步一步引出思考与解决问题的方式方法，此栏目下设置了两个重要模块：一是核心考点，用以给出题目主要考查的知识点；二是参考代码，用以给出试题对应的参考答案，而且大部分“参考代码”中都提供了贴心的注释。

（4）拓展提高。在原题目的基础上引导读者进行更深入的探讨，以拓展思路，掌握更多的解题技巧。

蓝桥杯大赛官方资源的获取

蓝桥云课是蓝桥杯大赛的官方资源平台。对于本书所提供的题目，读者都可以在蓝桥云课上进行模拟训练。蓝桥云课内嵌了在线评测系统，能进行自动判题，并返回有关正误的提示，从而帮助读者完全自主地高效学习编程。蓝桥云课所提供资源的相关链接如下。

链接 1：蓝桥杯大赛官方网站，dasai.lanqiao.cn。

链接 2：Python 3 自带标准库，https://docs.python.org/3/library/index.html。

链接 3：蓝桥杯大赛历届真题，https://www.lanqiao.cn/courses/2786。

链接 4：蓝桥杯官网题库，www.lanqiao.cn/problems，简称 lanqiaoOJ。

需要特别指出的是，蓝桥云课内嵌的在线评测系统提供了海量的算法竞赛经典例题及蓝桥杯软件类大赛的历年真题，真题所使用的评测数据与大赛评分时使用的测试数据一致。如果你参加了比赛，那么赛后在蓝桥云课的“链接 3”与“链接 4”中提交源代码，可得到最接近于实际成绩的结果。如果你正在备赛，那么可以在蓝桥云课中找到完整的算法学习路线与训练体系，能在很大程度上帮助你取得好成绩。

此外，蓝桥云课的社区资源相当丰富，不仅有专为蓝桥杯大赛展开的话题，还涵盖了计算机相关岗位求职就业的内容。

致谢

写作是一个痛苦并快乐的过程。在这个过程中，我们得到了蓝桥杯大赛负责人李艳萍老师的大力支持与指导；得到了华东理工大学程序设计竞赛主教练罗勇军老师提供的宝贵意见与建议；得到了蓝桥杯大赛参赛选手、算法竞赛爱好者刘子俊、常凤迪、刘会智、张晖的赞同与认可；得到了人民邮电出版社各位编辑给予的鼓励与帮助。在此深表感谢！

由于我们时间仓促、水平有限，书中难免存在错误或不当之处，恳请广大读者及时指正，以便“去伪存真”。若有任何关于本书的意见或建议，欢迎随时与我们沟通联系，我们教研团队的邮箱是 LQbook@lanqiao.cn，出版社的联系邮箱是 wangxudan@ptpress.com.cn。

期待您的反馈！

编　者

2022 年 8 月

目 录

CONTENTS

星号代表所涉及题目的难易程度

第 1 章 枚举与模拟

CHAPTER 1

1.1 卡片 (★)

题目信息

2021 年省赛-填空题

- ✧ C/C++ A 组第 1 题；C/C++ B 组第 2 题
- ✧ Java B 组第 2 题；Java C 组第 3 题
- ✧ Python 组第 1 题

【问题描述】

小蓝有很多数字卡片，每张卡片上都是一个数字（0 到 9）。

小蓝准备用这些卡片来拼一些数，他想从 1 开始拼出正整数，每拼一个，就保存起来，卡片就不能用来拼其他数了。

小蓝想知道自己能从 1 拼到多少。

例如，当小蓝有 30 张卡片，其中 0 到 9 各 3 张，则小蓝可以拼出 1 到 10，但是拼 11 时卡片 1 已经只有一张了，不够拼出 11。

现在小蓝手里有 0 到 9 的卡片各 2021 张，共 20210 张，请问小蓝可以从 1 拼到多少?

提示：建议使用计算机编程解决问题。

【答案提交】

这是一道结果填空题，你只需要算出结果后提交即可。本题的结果作为一个整数，在提交答案时只填写这个整数，填写多余的内容将无法得分。

懒人速读

> 小蓝有很多数字卡片，每张卡片上都必然刻有 0 ～ 9 中的一个数字。
> 卡片可以用来拼凑数字，假设小蓝拥有刻有数字 1 和数字 2 的卡片各一张，那么他可以利用这两张卡片拼凑出数字 $1, 2, 12, 21$ 这 4 个正整数。

> 现给定刻有数字 $0,1,\ldots,9$ 的卡片各 2021 张，小蓝将利用这些卡片从 1 开始连续拼凑数字，请问他最大能拼到数字几（注意：每张卡片只能利用一次）？

题目分析

核心考点

- ✧ 枚举
- ✧ 模拟

本题思路较为简单，可直接从数字 1 开始往后枚举，枚举的同时检查剩下的卡片能不能拼出当前枚举到的数字即可。

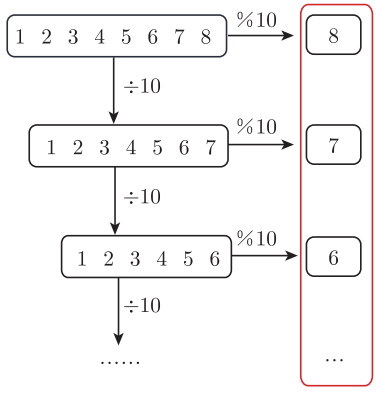
定义 $\mathrm{cnt}[i]$ 表示刻有数字 i 的卡片张数。那么按照题目的意思，初始化 $\mathrm{cnt}[0\sim 9]=2021$，参考代码 1-1-1 如下。

参考代码 1-1-1

```
int cnt[10];
for(int i = 0 ; i <= 9 ; i ++) cnt[i] = 2021;
```

若我们将步骤拆分为两步，则可分为枚举、检查。枚举没什么问题，写个循环即可，但该怎么检查呢?

首先就需要把一个数在十进制下的每一位数字求出来。怎么求呢? 可以将该数对 10 取模，这样就得到了个位上的数字；再除以 10，这样原本十位上的数字就跑到了个位上；然后再对 10 取模……反复这个过程，就可以得到这个数在十进制下所有数位上的数字了。



得到数位上的所有数字后再看看这些数字对应的卡片够不够，不够的话就说明这个数拼不了，就停止枚举，这样答案就为上一个枚举的数。

参考代码 1-1-2

```
bool check(int x){ // x 表示当前枚举的数
   while(x){
      int b = x % 10; // 获取十进制下的每个数位
```

```
4         if(cnt[b] > 0) cnt[b] --; // 这个数位对应的卡片个数 - 1
5         else return false; // 如果卡片不够了，则无法拼凑出该数
6         x /= 10; // 将十位变为个位
7     }
8     return true;
9 }
```

至此，检查的模块就完成了。不过我们还可以进行一点小小的优化。

我们的枚举是从 1 开始的，如果分别看枚举数的个位、十位……上的数字，那么会得到如下的内容。

	1	2	3	4	5	6	7	8	9	10	11	12	…	19	20	21	…
个位	1	2	3	4	5	6	7	8	9	0	1	2	…	9	0	1	…
十位	×	×	×	×	×	×	×	×	×	1	1	1	…	1	2	2	…
…																	

显然，个、十、百、千、万……每一数位的变化都是有周期性的。每个周期中各个数字出现的次数都是相同的，且每一数位都是从 1 开始变化的。因此，刻有数字 1 的卡片一定会被最早使用完，我们只需要对该卡片的张数作判断就好，具体代码可见参考代码第 4 ∼ 14 行。

参考代码 1-1-3

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int cnt[10];
4  bool check(int x){ // x 表示当前枚举的数
5      while(x){
6          int b = x % 10; // 获取十进制下的每个数位
7          if(b == 1) {
8              if(cnt[1] == 0) return false;
9              cnt[1] -- ;
10         }
11         x /= 10; // 将十位变为个位
12     }
13     return true;
14 }
15 signed main(){
16     for(int i = 0 ; i <= 9 ; i ++) cnt[i] = 2021;
17     for(int i = 1 ; ; i ++){
18         if(!check(i)) return cout << i - 1 << '\n' , 0;
19     }
20     return 0;
21 }
```

最终答案为 3181。

1.2 回文日期 (★)

题目信息

2020 年省赛-填空题

- ✧ C/C++ A 组第 7 题；C/C++ B 组第 7 题
- ✧ Java A 组第 7 题

【问题描述】

2020 年春节期间，有一个特殊的日期引起了大家的注意：2020 年 2 月 2 日。因为如果将这个日期按 “yyyymmdd” 的格式写成一个 8 位数是 20200202，恰好是一个回文数。我们称这样的日期是回文日期。

有人表示 20200202 是 “千年一遇” 的特殊日子。对此小明很不认同，因为不到 2 年之后就是下一个回文日期：20211202 即 2021 年 12 月 2 日。

也有人表示 20200202 并不仅仅是一个回文日期，还是一个 ABABBABA 型的回文日期。对此小明也不认同，因为大约 100 年后就能遇到下一个 ABABBABA 型的回文日期：21211212 即 2121 年 12 月 12 日。算不上 “千年一遇”，顶多算 “千年两遇”。

给定一个 8 位数的日期，请你计算该日期之后下一个回文日期和下一个 ABABBABA 型的回文日期各是哪一天。

【输入格式】

输入包含一个八位整数 N，表示日期。

对于所有评测用例，$10000101 \leqslant N \leqslant 89991231$，保证 N 是一个合法日期的 8 位数表示。

【输出格式】

输出两行，每行 1 个八位数。第一行表示下一个回文日期，第二行表示下一个 ABABBABA 型的回文日期。

【样例输入】

20200202

【样例输出】

20211202

21211212

【答案提交】

这是一道结果填空题，你只需要算出结果后提交即可。本题的结果作为一个整数，在提交答案时只填写这个整数，填写多余的内容将无法得分。

懒人速读

给定（输入）一个 8 位数的日期，请找出（输出）该日期之后的下一个回文日期以及下一个 ABABBABA 型回文日期。

若给定的日期为 20200202，则会有下列结果。

- 下一个回文日期为 20211202。
- 下一个 ABABBABA 型回文日期为 21211212。

题目分析

核心考点

✧（日期）枚举

✧ 合法日期判断

根据题意，我们可以将题目拆解为如何判断回文和如何枚举日期两个部分来解决。

1. 如何判断回文

对于一个 8 位数的整型日期，可以通过除法和取余运算来获取它的每一位数字。

举例说明

若整数 date 的值为 20050511，它从高到低的每一数位上的数可以通过下列方法得到。

- data / 10000000 = 2;
- data / 1000000 % 10 = 0;
- data / 100000 % 10 = 0;
- data / 10000 % 10 = 5;
- data / 1000 % 10 = 0;
- data / 100 % 10 = 5;
- data / 10 % 10 = 1;
- data % 10 = 1。

对于本题，若用 $a[1]$、$a[2]$、$a[3]$、$a[4]$、$a[5]$、$a[6]$、$a[7]$、$a[8]$ 分别存储每一位，则一个回文日期须满足:

$$a[1] = a[8]$$

$$a[2] = a[7]$$

$$a[3] = a[6]$$

$$a[4] = a[5].$$

一个 ABABBABA 型回文日期须满足以下条件。

$$a[1] = a[3] = a[6] = a[8]$$
$$a[2] = a[4] = a[5] = a[7].$$

对于一个日期字符串，可以直接通过下标来获取它的每一位数字。例如，string date = “20050511”，它从高到低的每一位分别可以通过 data[0], data[1], . . . , data[7] 得到。判断回文日期及 ABABBABA 型回文日期的方式和上述方法类似。

不难看出，字符串获取日期的每一位数字是要比整型简单的，所以在判断回文日期及 ABABBABA 型回文日期时可以将整型转换为字符串来操作，如参考代码 1-2-1 所示。

参考代码 1-2-1

```
int date = 20050511;
string s = to_string(date); 将整型转换为字符串, s = "20050511"

// 判断回文日期
bool check1(int date){
    string s = to_string(date);
    if(s[0] == s[7] && s[1] == s[6] && s[2] == s[5] && s[3] == s[4]) return true;
    return false;
}
// 判断 ABABBABA 型回文日期
bool check2(int date){
    string s = to_string(date);
    if(s[0] == s[2] && s[0] == s[5] && s[0] == s[7] && s[1] == s[3] && s[1] == s[4] &&
           s[1] == s[6]) return true;
    return false;
}
```

2. 如何枚举日期

对于一个整型日期,可以用最简单的方式枚举,即令该日期不断 +1,直到出现满足 ABABBABA 型的回文日期。

但这样存在一个问题：会枚举出许多不合法的日期（如 20209999）。为此，我们需要对日期进行合法性判断：设 y, m, d 分别表示年、月、日，month[i] 表示第 i 个月的天数，那么当 $m < 1$ 或 $m > 12$ 或 $d < 1$ 或 $d >$ month[m] 时日期不合法，反之日期合法，如参考代码 1-2-2 所示。

参考代码 1-2-2

```
int month[13] = {-1 , 31 , 28 , 31 , 30 , 31 , 30 , 31 , 31 , 30 , 31 , 30 , 31};
bool check(int date){
    string s = to_string(date);
```

```
    // stoi -> 将字符串转为整型
    int y = stoi(s.substr(0, 4)), m = stoi(s.substr(4, 2)), d = stoi(s.substr(6, 2));
    if(y % 400 == 0 || (y % 4 == 0 && y % 100 != 0)) month[2] = 29;
    else month[2] = 28;
    if(m < 1 || m > 12 || d < 1 || d > month[m]) return false;
    return true;
}
for(int i = date + 1 ; ; i ++){
    if(!check(i)) continue ;
    ...
}
```

不过这样的枚举量是相当大的，要在比赛中完成本题要求的所有测试数据不太现实。

那么，还有什么枚举方法呢?

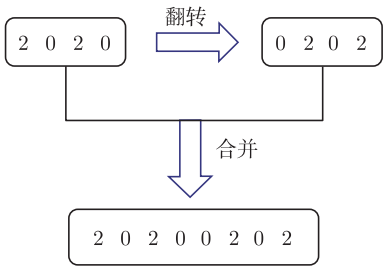
可以直接枚举合法日期。枚举合法日期只需模拟日期的变化规律即可，对应参考代码如下所示。

参考代码 1-2-3

```
int month[13] = {-1 , 31 , 28 , 31 , 30 , 31 , 30 , 31 , 31 , 30 , 31 , 30 , 31};
...
int date = 20050511;
int y = date / 10000 , m = date / 100 % 100 , d = date % 100;
// date = y * 10000 + m * 100 + d;

for(int i = y ; ; i ++){
    if(i % 400 == 0 || (i % 100 != 0 && i % 4 == 0)) month[2] = 29; // 闰年2月有29天
    else month[2] = 28;
    int j = (i == y) ? m : 1; // 如果是 i = y，则月份从 m 开始枚举，否则 m 从 1 开始枚举
    for(; j <= 12 ; j ++){
        int k = (i == y && j == m) ? d : 1; // 如果 i = y 并且 j = m，则日从 d 开始枚举，否则 d 从 1 开始枚举
        for(; k <= month[j] ; k ++){
            // date = i * 10000 + j * 100 + k
            ...
        }
    }
}
```

我们还可以只枚举年份。因为答案一定是个回文日期（即回文串），所以枚举年份 y，再将其翻转得到 y'，得到的 $y+y'$ 就一定是个回文串，如下页图所示。接着再对该串所对应的日期进行合法判断、ABABBABA 型回文判断即可。



参考代码 1-2-4 【枚举合法日期】

```
#include<bits/stdc++.h>
using namespace std;
int month[13] = {-1, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int date, ans1, ans2;
// 判断日期是否为回文
bool check1(int date){
    string s = to_string(date);
    if(s[0] == s[7] && s[1] == s[6] && s[2] == s[5] && s[3] == s[4]) return true;
    return false;
}
// 判断日期是否满足 ABABBABA 型回文
bool check2(int date)
{
    string s = to_string(date);
    if(s[0] == s[2] && s[0] == s[5] && s[0] == s[7] && s[1] == s[3] && s[1] == s[4] &&
           s[1] == s[6]) return true;
    return false;
}
signed main()
{
    cin >> date;
    int y = date / 10000, m = date / 100 % 100, d = date % 100;
    for(int i = y ; ; i ++) {
        if(i % 400 == 0 || (i % 100 != 0 && i % 4 == 0)) month[2] = 29;
        else month[2] = 28;
        int j = (i == y) ? m : 1;
        for(; j <= 12 ; j ++) {
            int k = (i == y && j == m) ? d + 1 : 1;
            for(; k <= month[j] ; k ++) {
                int date = i * 10000 + j * 100 + k;
                if(check1(date) && ans1 == 0) ans1 = date;
                if(check2(date)) return cout << ans1 << '\n' << date << '\n', 0; // 当找到了
                     ABABBABA 型回文日期时, 结束程序
```

```
            }
        }
    }
    return 0;
}
```

参考代码 1-2-5　【枚举年份】

```
#include<bits/stdc++.h>
using namespace std;
// month[1 ~ 12] 分别记录 1 ~ 12 月每月的天数
int date , month[13] = {-1 , 31 , 28 , 31 , 30 , 31 , 30 , 31 , 31 , 30 , 31 , 30 , 31};
// 判断日期是否合法
string check(int date){
    string s = to_string(date) , t = to_string(date); // 将整型转换为字符串
    reverse(t.begin() , t.end()); // 翻转
    s += t; // 将 s 翻转后再与 s 拼接，拼接后的字符串一定会是个回文串
    int y = stoi(s.substr(0 , 4)), m = stoi(s.substr(4 , 2)), d = stoi(s.substr(6 , 2));
    if(y % 400 == 0 || (y % 4 == 0 && y % 100 != 0)) month[2] = 29;
    else month[2] = 28;
    if(m < 1 || m > 12 || d < 1 || d > month[m]) return "-1";
    return s;
}
// 判断日期是否是 ABABBABA 型回文
string check2(int date){
    string s = to_string(date) , t = to_string(date);
    reverse(t.begin() , t.end()); // 翻转
    s += t;
    if(s[0] == s[2] && s[0] == s[5] && s[0] == s[7] && s[1] == s[3] && s[1] == s[4] &&
            s[1] == s[6]) return s;
    return "-1";
}
signed main()
{
    cin >> date;
    string ans1 = "";
    for(int i = date / 10000 ; ; i ++){
        // 注意：date（输入的日期）不能作为答案
        if(check(i) == "-1" || check(i) == to_string(date)) continue ;
        if(ans1 == "") ans1 = check(i);
        if(check2(i) != "-1") return cout << ans1 << '\n' << check2(i) << '\n' , 0;
    }
    return 0;
```

```
35 }
```

1.3 赢球票 (★)

题目信息

2016 年国赛-程序设计题

- ✧ C/C++ C 组第 4 题
- ✧ Java C 组第 4 题

【问题描述】

某机构举办球票大奖赛。获奖选手有机会赢得若干张球票。

主持人拿出 N 张卡片（上面写着 $1,\ldots,N$ 的数字），打乱顺序，排成一个圆圈。

你可以从任意一张卡片开始顺时针数数: $1,2,3\ldots$

如果数到的数字刚好和卡片上的数字相同，则把该卡片收入囊中，从下一张卡片重新数数。

直到再无法收获任何卡片，游戏结束。囊中卡片数字的和就是赢得球票的张数。

比如卡片排列是 1 2 3，我们从 1 号卡开始数，就把 1 号卡拿走。再从 2 号卡开始，但数的数字无法与卡片对上，很快数字越来越大，不可能再拿走卡片了。因此这次我们只赢得了 1 张球票。

还不算太坏！如果我们开始就傻傻地从 2 或 3 号卡片数起，那就一张卡片都拿不到了。

如果运气好，卡片排列是 2 1 3，那我们可以顺利拿到所有的卡片！

本题目标：已知顺时针卡片序列，随便你从哪里开始数，求最多能赢多少张球票（就是收入囊中的卡片数字之和）。

【输入格式】

第一行一个整数 $N(N \leqslant 100)$，表示卡片数目。

第二行 N 个整数，表示顺时针排列的卡片。

【输出格式】

输出一行，一个整数，表示最好情况下能赢得多少张球票。

【样例输入】

```
3
1 2 3
```

【样例输出】

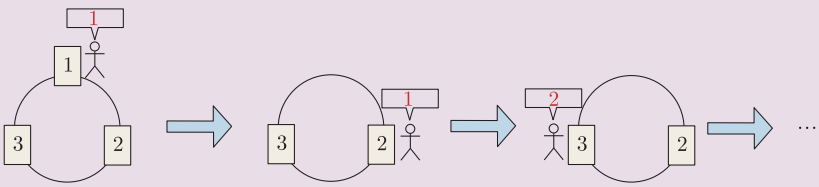
```
1
```

懒人速读

> 给定 n 张卡片，卡片上分别写着数字 $a_1, a_2, \ldots, a_n$。
> 现将这 n 张卡片围成一个圈。
> 我们可以从圈上任意一个位置开始顺时针数数：$1, 2, 3 \ldots$。
> 若当前数到的数字和卡片上的数字相同，则将卡片取走，并从下一张卡片所在位置重新数数：$1, 2, 3 \ldots$。
> 问我们取走的卡片上的数字之和最大可以是多少？

举例说明

若 $n = 3, a = [1, 2, 3]$，则从第一张卡片所在位置开始数数的过程如下。



最终只能取走第一张卡片，固数字之和为 1

题目分析

核心考点

- ✧ 枚举
- ✧ 拆环成链

这是一道十分经典的模拟题。刚着手解决本题时，可能绝大部分人都会提出 4 个问题。

问题 1. 题目给定的是一个环（圆圈），我们需要如何模拟在环上的移动呢?

问题 2. 如何表示被拿走的卡片呢?

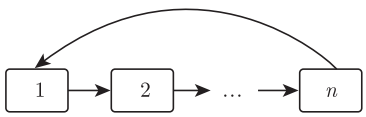
问题 3. 从哪个位置（起点）开始数数能拿走最多的卡片呢?

问题 4. 游戏什么时候会结束呢?

下面依次对这 4 个问题进行分析。

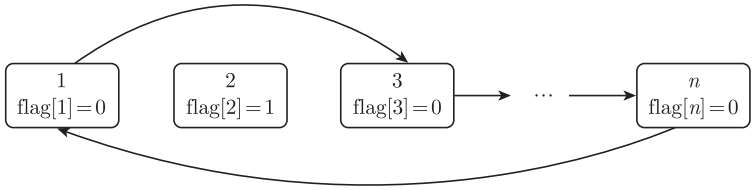
对于问题 1：

对于在序列上的移动，如果想从当前位置顺时针移动到下一个位置，只要让下标 $+1$ 即可。但当处于第 n 个位置时，由于没有第 $n+1$ 个位置，所以无法移动。而对于环，第 $n+1$ 个位置即第 1 个位置，所以从第 n 个位置移动到下一个位置只要让下标为 1 即可，其他位置的移动和序列相同。



对于问题 2：

可以对被拿走的卡片打上标记。定义 flag[] ，其中 flag[i] 表示第 i 张是否被取走（flag[i] = 1 表示被取走，flag[i] = 0 表示没有被取走）。下图展示了第 2 张卡片被拿走的情况。



对于问题 3：

不知道。既然不知道，就将每个位置都作为一次起点并模拟整个过程，再从中选出最大的卡片和（即答案）。

对于问题 4：

游戏结束分为以下两种情况。

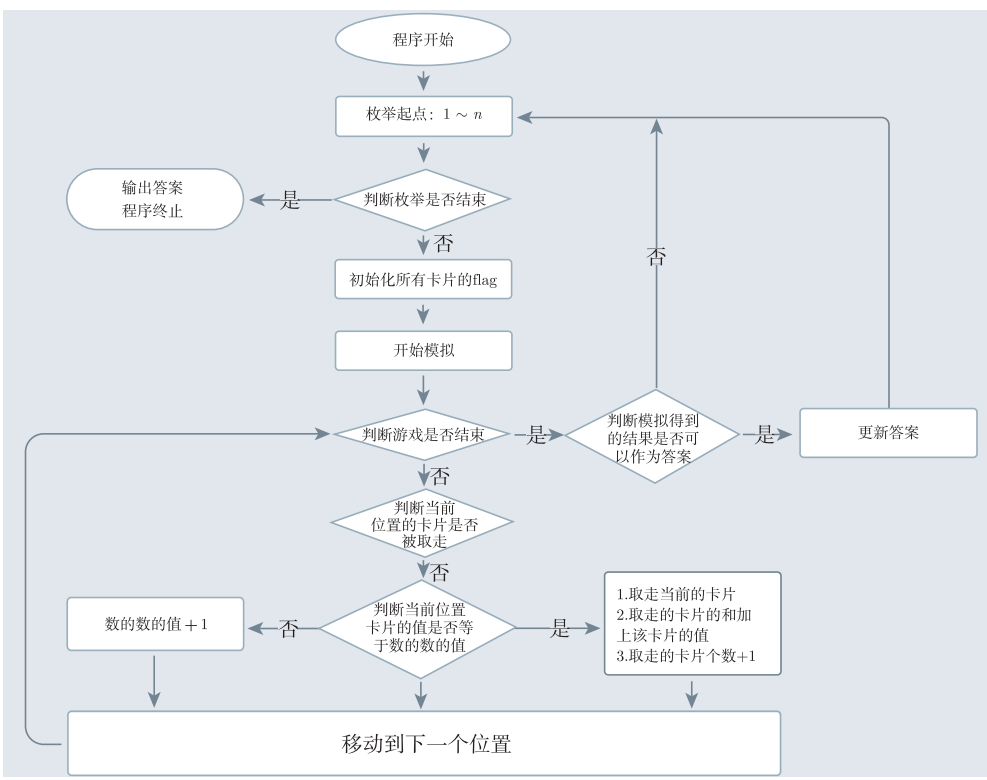
- 取走所有的卡片（即取走 n 张卡片）。
- 当前数的数大于 n（不可能再取走卡片了）。

解决了上述 4 个问题，就可以开始解题了。

按照上面的分析，我们需要完成以下几点。

（1）枚举起点，并在每次枚举了一个起点后将所有卡片的 flag 初始化为 0。

（2）有了起点后就可以模拟数数的过程，流程图如下。



- 判断游戏是否结束。
- 判断当前位置的卡片是否被取走（若被取走，则移动到下一个位置）。
- 判断当前位置的卡片的值是否和数的数的值相同（相同则取走该卡片，并重新开始数数）。
- 判断这次模拟得到的结果是否可以作为答案。

参考代码 1-3　【赢球票】

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int n, a[N], flag[N];
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++) cin >> a[i];
    int ans = 0; // ans 表示答案
    for(int i = 1 ; i <= n ; i ++){ // i 表示枚举的起点
        // 将所有卡片的 flag 初始化为 0
        for(int j = 1 ; j <= n ; j ++) flag[j] = 0; // flag[j] = 1 表示卡片被取走，反之没被取走
        int num = 1 , pos = i , sum = 0, cnt = 0; // num 表示数的数的值，pos 表示当前位置，sum 表示以 i 为起点数数所能取走的卡片的和，cnt 表示拿走的卡片的数量

        // 开始模拟
        while(1){
            // 判断游戏是否结束
            if(cnt == n || num > n) break; // 如果取走了所有卡片，或者数的数大于 n（不可能再取走卡片了），则游戏结束
            // 判断当前位置的卡片是否被取走
            if(flag[pos] == 1) { // 如果该卡片已被取走，则移动到下一个位置
                // 移动到下一个位置：如果当前位置 pos = n，则下一个位置为 1，否则下一个位置为 pos + 1
                if(pos == n) pos = 1;
                else pos ++;
                continue ;
            }
            // 数的数和当前位置卡片的值相同时取走该卡片（a[pos] 表示第pos张卡片的值）
            if(num == a[pos]){
                sum += a[pos]; // 取走卡片的和 + a[pos]
                cnt ++; // 取走的卡片个数 + 1
                num = 1; // 取走卡片后要重新数数
                flag[pos] = 1; // 取走卡片后该卡片对应的 flag 置为 1
                // 移动到下一个位置：如果当前位置 pos = n，则下一个位置为 1，否则下一个位置为
```

```
                pos + 1
32              if(pos == n) pos = 1;
33              else pos ++;
34          } else {
35              // 数的数和当前位置卡片的值不相同时
36              num ++ ; // 数的数的值 + 1
37              // 移动到下一个位置：如果当前位置 pos = n，则下一个位置为 1，否则下一个位置为
                    pos + 1
38              if(pos == n) pos = 1;
39              else pos ++;
40          }
41      }
42      // 模拟结束，判断该模拟结果是否可以作为答案
43      if(sum > ans) ans = sum;
44    }
45    cout << ans << '\n';
46    return 0;
47  }
```

1.4 既约分数 (★)

题目信息

2020 年省赛-填空题

- ✧ C/C++ A 组第 2 题；C/C++ B 组第 2 题
- ✧ Java A 组第 2 题

【问题描述】

如果一个分数的分子和分母的最大公约数是 1，这个分数称为既约分数。

例如 $\frac{3}{4}, \frac{1}{8}, \frac{7}{1}$，都是既约分数。

请问，有多少个既约分数，分子和分母都是 1 到 2020 之间的整数（包括 1 和 2020 ）?

【答案提交】

这是一道结果填空题，你只需要算出结果后提交即可。本题的结果作为一个整数，在提交答案时只填写这个整数，填写多余的内容将无法得分。

题目分析

核心考点

- ✧ gcd（最大公约数）

用一句话概括题意，即统计分子、分母的范围均为 $1 \sim 2020$ 且分子、分母的最大公约数（gcd）为 1 的分数的个数。

解题的思路较为简单，只需从 $1 \sim 2020$ 枚举分子，再从 $1 \sim 2020$ 枚举分母，然后判断分子、分母的最大公约数是否为 1（若为 1 则答案个数 $+1$）即可。不过在此之前，我们需要知道如何求解两个数的最大公约数。

求最大公约数的方法很多，如果想图个方便，可以直接调用 C++ 的 **algorithm** 库中的**__gcd()** 函数来求解。如果不知道这个函数也没有关系，因为想求解两个数的最大公约数并不难。我们可以采用常用的辗转相除法（国际上也称“欧几里得算法”），它的代码如下。

参考代码 1-4-1　【复杂度为 $O(\log)$】

```
int gcd(int a , int b){
    if(b == 0) return a;
    return gcd(b, a % b);
}
```

知道了如何求解两个数的最大公约数后，就可以枚举统计答案了。

参考代码 1-4-2　【复杂度为 $O(N^2 \log N)$】

```
#include<bits/stdc++.h>
using namespace std;
signed main()
{
    int ans = 0;
    for(int i = 1 ; i <= 2020 ; i ++){
        for(int j = 1 ; j <= 2020 ; j ++){
            if(__gcd(i , j) == 1) ans ++ ;
        }
    }
    cout << ans << '\n';
    return 0;
}
```

最终答案为 2481215。

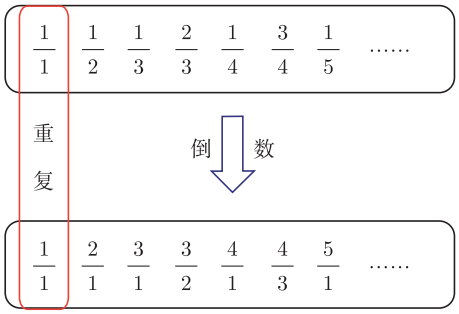
拓展提高

我们来考虑一下如何优化程序的时间复杂度。

若两个数 i, j（$j \leqslant i$）的最大公约数为 1，则 $\frac{j}{i}$ 是一个既约分数，$\frac{i}{j}$ 也是一个既约分数。所以我们可以从 $1 \sim 2020$ 枚举 i，从 $1 \sim i$ 枚举 j，这样得到的 i, j 就满足 $j \leqslant i$。接着判断 i, j 的最大公约数是否为 1（若为 1，则 $\frac{j}{i}$ 是既约分数）。

若用 cnt 来统计有多少对 $\frac{j}{i}$ 满足 $j \leqslant i$ 且 i, j 的最大公约数为 1，则答案为 $\text{cnt} \times 2 - 1$（乘 2 是因为 $\frac{j}{i}$ 的倒数 $\frac{i}{j}$ 也是既约分数，减 1 是因为 $\frac{1}{1}$ 的倒数还是 $\frac{1}{1}$，重复计算了一次，要

减掉，见下图）。



这样就可以细微地优化程序计算的次数了。

不过程序的复杂度仍为 $O(N^2 \log N)$。有没有什么办法能降低复杂度呢？答案自然是有。

核心考点

- ✧ 唯一分解定理
- ✧ 欧拉函数
- ✧ 欧拉筛

对于最大公约数为 1 的两个整数，我们称它们的关系为互质。在数论中，$1 \sim n$ 中与 n 互质的数的个数被称为欧拉函数（记作 $\text{phi}[n]$）。

由欧拉函数的定义可知满足 $j \leqslant i$，且 i, j 最大公约数为 1 的 $\frac{j}{i}$ 的个数就为 $\text{phi}[i]$。

我们要求的是 $1 \sim 2020$ 内满足 $j \leqslant i$ 且 i, j 最大公约数为 1 的 $\frac{j}{i}$ 的个数，即 $1 \sim 2020$ 内所有数的欧拉函数的和，即 $\text{cnt} = \sum_{i=1}^{2020} \text{phi}[i]$。

欧拉函数的通项公式：

$$\text{phi}[n] = n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\left(1-\frac{1}{p_3}\right)\ldots\left(1-\frac{1}{p_k}\right).$$

其中 $p_1, p_2, \ldots, p_k$ 为 n 的所有不重复质因子。

要想得到 n 的所有不重复质因子，我们可以采用数论利器——唯一分解定理。

提示

唯一分解定理就是对于一个大于 1 的整数 n，要么其本身是个质数，要么能被分解为有限个质数的乘积。

若用数学公式表示，则 $n = \prod_{i=1}^{k} p_i^{a_i} = p_1^{a_1} \times p_2^{a_2} \times \ldots \times p_k^{a_k}$，$p_i$ 表示质数（即 n 的质因子），a_i 表示 p_i 的个数。

例如，18 就可以表示为 $2^1 \times 3^2$，它的质因子有 2、3。其中质因子 2 的个数为 1，质因子 3 的个数为 2。

综上，我们可以写出如下代码，完成对 n 的不重复质因子的求解。

参考代码 1-4-3

```
int p[N]; // p[] 记录因子，p[1]是最小因子
int getPrime(int n){
    int k = 0; // k 记录 n 的质因子的个数
    for(int i = 2 ; i * i <= n ; i ++){
        if(n % i == 0) {
            p[++ k] = i;
            while(n % i == 0) n /= i; // 把 n 重复的质因子去掉
        }
    }
    if(n > 1) p[++ k] = n; // n 没有被除尽，是素数
    return n;
}
```

再根据欧拉函数的通项公式 $\left[\text{phi}[n] = n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\left(1-\frac{1}{p_3}\right)\dots\left(1-\frac{1}{p_k}\right)\right]$ 写出代码。

参考代码 1-4-4

```
int getPhi(int n){
    int phi = n , k = getPrime(n);
    for(int i = 1 ; i <= k ; i ++){ // 枚举所有质因子: p1,p2,...,pk
        phi = phi - phi / p[i]; // (phi - phi / p[i])等价于 phi(1 - 1 / p[i])
    }
    return phi;
}
```

因为在本题中质因子的作用只是为了计算欧拉函数，所以可以不用开辟数组记录质因子，而是每得到一个质因子就将其放入欧拉函数的通项公式中计算，这样两份代码就能合并在一起，具体代码可见参考代码 1-4-5 的第 3 ∼ 13 行。

参考代码 1-4-5　【时间复杂度为 $O(N\sqrt{N})$】

```
#include<bits/stdc++.h>
using namespace std;
int solve(int n){
    int phi = n;
    for(int i = 2 ; i * i <= n ; i ++){
        if(n % i == 0) { // i 是 n 的质因子
            phi = phi - phi / i; // 将其丢入欧拉函数的通项公式中计算
            while(n % i == 0) n /= i; // 把 n 重复的质因子去掉
        }
    }
    if(n > 1) phi = phi - phi / n; // n 没有被除尽，是素数，将其丢入欧拉函数的通项公式中计算
```

```
    return phi;
}
signed main()
{
    int ans = 0;
    for(int i = 1 ; i <= 2020 ; i ++) ans += solve(i);
    cout << ans * 2 - 1 << '\n';
    return 0;
}
```

当然，求一个数的欧拉函数还有更好的求法，比如用 Pollard_Rho 寻找因子、用 Miller_Rabin 测试质数，以将复杂度优化到 $O(N^{\frac{1}{4}})$。不过这种做法码量极大，所涉及的知识点难度也高，蓝桥杯赛场上几乎是不可能用上的。

此外，我们还可以使用欧拉筛，它的作用是以 $O(N)$ 的时间复杂度求出 $1 \sim N$ 中每个数的欧拉函数。这样，程序的复杂度就可以被进一步优化。

参考代码 1-4-6　【时间复杂度为 $O(N)$】

```
#include<bits/stdc++.h>
using namespace std;
int n , phi[2021] , prime[2021];
signed main(){
    // 欧拉筛
    phi[1] = 1;
    for(int i = 2 ; i <= 2020 ; i ++){
        if(!prime[i]) prime[++ n] = i, phi[i] = i - 1;
        for(int j = 1 ; j <= n && i * prime[j] <= 2020 ; j ++){
            prime[prime[j] * i] = 1;
            if(i % prime[j] == 0){
                phi[i * prime[j]] = phi[i] * prime[j];
                break ;
            }
            phi[i * prime[j]] = phi[i] * (prime[j] - 1);
        }
    }
    int cnt = 0;
    for(int i = 1 ; i <= 2020 ; i ++) cnt += phi[i];
    cout << 2 * cnt - 1 << '\n';
    return 0;
}
```

1.5　数的分解 (★)

题目信息

2019 年省赛-填空题

- ✧ C/C++ B 组第 4 题
- ✧ Java B 组第 4 题

【问题描述】

把 2019 分解成 3 个各不相同的正整数之和，并且要求每个正整数都不包含数字 2 和 4，一共有多少种不同的分解方法?

注意交换 3 个整数的顺序被视为同一种方法,例如 1000+1001+18 和 1001+1000+18 被视为同一种。

【答案提交】

这是一道结果填空题，你只需要算出结果后提交即可。本题的结果作为一个整数，在提交答案时只填写这个整数，填写多余的内容将无法得分。

题目分析

核心考点

- ✧ 枚举

本题可以用枚举来解决。不过在此之前，先来分析以下两个问题。

问题 1. 如何判断一个整数是否包含 2 或 4?

问题 2. 如何解决重复情况?

对于问题 1，只要取出该整数在十进制下的每一位的数字，并判断这些数字是否等于 2 或 4 即可，具体代码可见参考代码 1-5-1 第 4 ∼ 10 行。

对于问题 2，已知一个方案是由 3 个数组成（第 1 个数，第 2 个数，第 3 个数），这 3 个数互不相同且和为 2019。假设这 3 个数分别为 i, j, k，那么它们可以构造出 (i, j, k)、(i, k, j)、(j, i, k)、(j, k, i)、(k, i, j)、(k, j, i) 6 种方案。根据题意，这 6 种方案只能视为一种，所以可以先求出满足条件的所有方案数，再将总方案数除 6 即可。

分析完上述两个问题，就能轻松写出以下代码。

参考代码 1-5-1　【复杂度为 $O(N^3 \lg N)$，$\lg N$ 是 check() 函数的复杂度】

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  // 如果 x 包含 2 或 4 返回 false，否则返回 true
4  bool check(int x){
```

```
    while(x){
        if(x % 10 == 2 || x % 10 == 4) return false;
        x /= 10;
    }
    return true;
}
signed main(){
    int ans = 0;
    for(int i = 1 ; i <= 2019 ; i ++)
        for(int j = 1 ; j <= 2019 ; j ++)
            for(int k = 1 ; k <= 2019 ; k ++)
                // i, j, k 互不相同
                // i + j + k = 2019
                // i, j, k 都不包含 2 和 4
                if(i != j && i != k && j != k && i + j + k == 2019 && check(i) && check(j)
                    && check(k)){
                    ans ++ ;
                }
    ans /= 6;
    return 0;
}
```

虽然能轻松写出以上代码，但本题 n 的规模为 2019，代码要运行很久才可得出答案。因此，我们需要对复杂度进行优化。怎么优化呢? 可以考虑从以下 3 点入手。

（1）添加约束条件。对于任意一个方案，只有当该方案中的第 1 个数小于第 2 个数、第 2 个数小于第 3 个数时才视该方案为合法方案，如下图所示。这样不仅能保证任意 3 个整数只能构造出一个方案（符合题意），而且可以减少枚举次数（第 2 个数从第 1 个数 +1 开始枚举，第 3 个数从第 2 个数 +1 开始枚举）。

$i<j<k$	
$\bullet(i,\ j,\ k)$	√
$\bullet(i,\ k,\ j)$	×
$\bullet(j,\ i,\ k)$	×
$\bullet(j,\ k,\ i)$	×
$\bullet(k,\ i,\ j)$	×
$\bullet(k,\ j,\ i)$	×

参考代码 1-5-2

```
for(int i = 1 ; i <= 2019 ; i ++)
    for(int j = i + 1 ; j <= 2019 ; j ++) // i < j, 所以 j 从 i + 1 开始枚举
        for(int k = j + 1 ; k <= 2019 ; k ++) // j < k, 所以 k 从 j + 1 开始枚举
            if(i + j + k == 2019 && check(i) && check(j) && check(k))
                ans ++ ;
```

不过这样的复杂度还是 $O(N^3 \lg N)$，并没有起到很好的效果。

（2）减少枚举变量。对于任意一个方案，它的 3 个整数的和都为 2019，即只要知道了第 1 个数和第 2 个数的值，就能确定第 3 个数的值（第 3 个数 = 2019− 第 1 个数 − 第 2 个数）。这样，就可以省去对第 3 个数的枚举，复杂度降至 $O(N^2 \lg N)$。

参考代码 1-5-3

```
for(int i = 1 ; i <= 2019 ; i ++)
   for(int j = i + 1 ; j < 2019 - i - j ; j ++) // i < j < 2019 - i - j
      if(check(i) && check(j) && check(2019 - i - j))
         ans ++;
```

（3）提前检查、标记所有数。除了上述两步优化，还可以在枚举开始前对不包含 2 和 4 的整数进行标记，方法是定义数组 flag[]，如果整数 i 不包含 2 和 4，则 flag[i] = 1，否则 flag[i] = 0。这样在枚举的过程中，就可以通过 flag[] 数组以 O(1) 的复杂度判断一个数是否包含 2 和 4，复杂度降至 $O(N^2)$，具体代码见参考代码 1-5-4 第 14 ∼ 18 行。最终答案为 40785。

参考代码 1-5-4　【时间复杂度为 $O(N^2)$】

```
#include<bits/stdc++.h>
using namespace std;
// 如果 x 包含 2 或 4 返回 false，否则返回 true
bool check(int x){
   while(x){
      if(x % 10 == 2 || x % 10 == 4) return false;
      x /= 10;
   }
   return true;
}
bool flag[2020];
signed main(){
   int ans = 0;
   for(int i = 1 ; i <= 2019 ; i ++) if(check(i)) flag[i] = 1; // 在枚举前判断一个数是否包含 2 和 4，避免在枚举过程中判断
   for(int i = 1 ; i <= 2019 ; i ++)
      for(int j = i + 1 ; j < 2019 - i - j ; j ++) // i < j < 2019 - i - j
         if(flag[i] && flag[j] && flag[2019 - i - j]) // 通过 flag 判断，复杂度 O(1)
            ans ++;
   cout << ans << '\n';
   return 0;
}
```

拓展提高

如果觉得 $O(N^2)$ 的复杂度还是太高，那么再来进一步降低复杂度吧！

核心考点

✧（多维）数位 dp

本题可以对 3 个数同时进行数位 dp。不过在开始 dp 之前，需要先考虑维护的信息。

假设现在有 3 个整数 i,j,k，要使它们能构成一组合法方案，按照题意须满足以下条件。

（1）i,j,k 互不相等。

（2）$i+j+k=2019$。

（3）i,j,k 都不包含 2 或 4。

（4）(i,j,k)、(i,k,j)、(j,i,k)、(j,k,i)、(k,i,j)、(k,j,i) 都属于一种方法。

要如何处理这些条件呢?

- 对于条件 1、条件 4，可以采用前文优化方法中的第 1 点，即添加约束条件。这样不仅能保证 i,j,k 互不相等，也能保证 i,j,k 只能构造出一种方法。我们可以在数位 dp 中添加两个信息 ok1, ok2，分别维护 i 是否小于 j、j 是否小于 k（ok1 $=1$ 表示 $i<j$，ok2 $=1$ 表示 $j<k$）。
- 对于条件 2，可以在数位 dp 中添加一个信息 sum，用来记录 i,j,k 的和。
- 对于条件 3，由于数位 dp 是在数位上进行操作的，所以可以在操作时直接判断是否包含 2 或 4。

此外，i,j,k 的范围是 $1\sim2019$。为了防止在 dp 的过程中出现 i 等于 0 的情况，我们可以添加一个信息 ok3 ，来判断 i 是否大于 0（ok3 $=1$ 表示 $i>0$）。

如此，当 ok1 $=1$, ok2 $=1$, ok3 $=1$, sum $=2019$ 时其含义就为 $0<i<j<k$，且 i,j,k 的和为 2019。

有了这些信息，就可以定义 dp[len][limit1][limit2][limit3][ok1][ok2][ok3][sum]，其中：len 表示当前计算到了十进制下的第 len 位，limit1 表示 i 当前数位的上限是否受到限制，limit2 表示 j 当前数位的上限是否受到限制，limit3 表示 k 当前数位的上限是否受到限制。然后，应用记忆化搜索即可得出答案。具体代码见参考代码 1-5-5，该代码可以处理数位长度规模为 10^5 的情况。

参考代码 1-5-5　【时间复杂度为 $O(2^6N\ln N)$】

```
#include<bits/stdc++.h>
using namespace std;
int dp[5][2][2][2][2][2][2][2020] , num[5] , p[5];
int dfs(int len , bool limit1 , bool limit2 , bool limit3 , bool ok1 , bool ok2 , bool
    ok3, int sum){
   if(!len) return sum == 2019 && ok1 && ok2 && ok3 ;
   if(~dp[len][limit1][limit2][limit3][ok1][ok2][ok3][sum]) return dp[len][limit1]
            [limit2][limit3][ok1][ok2][ok3][sum];
   int res = 0;
   // num[4] = 2, num[3] = 0 , num[2] = 1 , num[1] = 9
```

```
9     int up1 = limit1 ? num[len] : 9; // i 当前数位是否达到上限
10    int up2 = limit2 ? num[len] : 9; // j 当前数位是否达到上限
11    int up3 = limit3 ? num[len] : 9; // k 当前数位是否达到上限
12    for(int i = 0 ; i <= up1 ; i ++){
13       if(i == 2 || i == 4) continue ; // 如果 i 当前数位的值为 2 或 4 则跳过这种情况
14       int d1 = ok1 ? 0 : i; // 如果 j 还没有大于 i，为保证 j > i，当前数位上的数字需要从 i
             开始枚举
15       for(int j = d1 ; j <= up2 ; j ++){
16          if(j == 2 || j == 4) continue ; // 如果 j 当前数位的值为 2 或 4 则跳过这种情况
17          int d2 = ok2 ? 0 : j; // 如果 k 还没有大于 j，为保证 k > j，当前数位上的数字需要
                从 j 开始枚举
18          for(int k = d2 ; k <= up3 ; k ++){
19             if(k == 2 || k == 4) continue ; // 如果k当前数位的值为2或4则跳过这种情况
20             // 如果 i, j, k 的和超过 2019 则跳过这种情况
21             if(sum + i * p[len - 1] + j * p[len - 1] + k * p[len - 1] > 2019) break;
22             res += dfs(len - 1 , limit1 && i == up1 , limit2 && j == up2 , limit3 && k
                   == up3 ,
23                      // 只要存在一个数位，在该数位下 j 的值大于 i 的值，ok1 就等于 1
24                      // 只要存在一个数位，在该数位下 k 的值大于 j 的值，ok2 就等于 1
25                      // 只要存在一个数位，在该数位下 i 的值大于 0，ok3 就等于 1
26                      ok1 || i < j , ok2 || j < k , ok3 || i,
27                      sum + i * p[len - 1] + j * p[len - 1] + k * p[len - 1]);
28          }
29       }
30    }
31    return dp[len][limit1][limit2][limit3][ok1][ok2][ok3][sum] = res;
32 }
33 int solve(int x){
34    memset(dp , -1 , sizeof(dp));
35    int len = 0;
36    while(x){ // 取出 x（2019）的每一位数字存储在 num 中，进行数位 dp
37       num[++ len] = x % 10;
38       x /= 10;
39    }
40    // 传入的 x = 2019，则 num[4] = 2, num[3] = 0, num[2] = 1, num[1] = 9
41    return dfs(len, 1, 1, 1, 0, 0, 0, 0);
42 }
43 signed main(){
44    // p[i] 表示 10 的 i 次方，即 p[1] = 10^1 = 10 , p[2] = 10^2 = 100,...]
45    p[0] = 1;
46    for(int i = 1 ; i <= 4 ; i ++) p[i] = p[i - 1] * 10;
47    cout << solve(2019) << '\n';
```

```
48      return 0;
49  }
```

1.6 巩固与练习

题目	难度	题目年份	组别
跑步锻炼	★	2020 年省赛	• C/C++ B 组第 4 题；C/C++ C 组第 3 题 • Java C 组第 3 题 • Python 组第 3 题
货物摆放	★	2021 年省赛	• C/C++ A 组第 3 题；C/C++ B 组第 4 题；C/C++ C 组第 3 题 • Java A 组第 3 题；Java B 组第 4 题 • Python 组第 3 题
特别数的和	★	2019 年省赛	• C/C++ B 组第 6 题 • Java B 组第 6 题
完全二叉树的权值	★	2019 年省赛	• C/C++ A 组第 6 题；C/C++ B 组第 7 题 • Java A 组第 6 题
等差素数列	★	2017 年省赛	• C/C++ B 组第 2 题
猴子分香蕉	★	2018 年省赛	• C/C++ C 组第 2 题 • Java C 组第 2 题
天干地支	★	2020 年国赛	• C/C++ C 组第 6 题 • Java C 组第 6 题 • Python C 组第 6 题

第2章 搜索与查找

CHAPTER 2

2.1 九宫幻方 (★)

题目信息

2017 年省赛 - 程序设计题

✧ C/C++ C 组第 8 题

【问题描述】

小明最近在教邻居家的小朋友小学奥数，而最近正好讲述到了三阶幻方这个部分，三阶幻方指的是将 $1 \sim 9$ 不重复地填入一个 3×3 的矩阵当中，使得每一行、每一列和每一条对角线的和都是相同的。

三阶幻方又被称作九宫格，在小学奥数里有一句非常有名的口诀：“二四为肩，六八为足，左三右七，戴九履一，五居其中”，通过这样的一句口诀就能够非常完美地构造出一个九宫格来。

4	9	2
3	5	7
8	1	6

有意思的是，所有的三阶幻方，都可以通过这样一个九宫格进行若干镜像和旋转操作之后得到。现在小明准备将一个三阶幻方（不一定是上图中的那个）中的一些数抹掉，交给邻居家的小朋友进行还原，并且希望她能够判断出究竟是不是只有一个解。

而你呢，也被小明交付了同样的任务，但不同的是，你需要写一个程序。

【输入格式】

输入仅包含单组测试数据。

每组测试数据为一个 3×3 的矩阵，其中为 0 的部分表示被小明抹去的部分。

给出的矩阵至少能还原出一组可行的三阶幻方。

【输出格式】

如果仅能还原出一组可行的三阶幻方，则将其输出，否则输出 “Too Many”（不包含引号）

【样例输入】

0	7	2
0	5	0
0	3	0

【样例输出】

6	7	2
1	5	9
8	3	4

懒人速读

给定一个 3×3 的矩阵，矩阵内所有元素的值的范围均为 $0 \sim 9$。

现要求用 $1 \sim 9$ 中的某一数字替换矩阵中值为 0 的元素，使得替换后的矩阵满足以下条件。

(1) 矩阵的每一行、每一列、每一条对角线的和都相同。

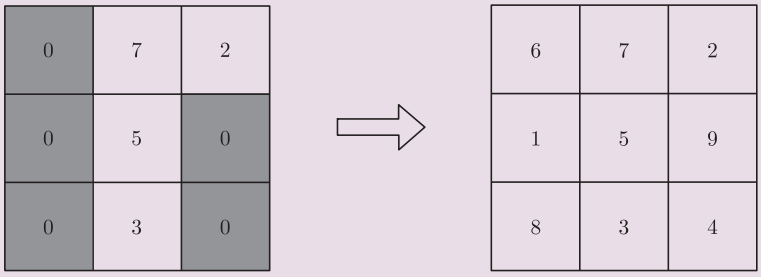
(2) 矩阵所有元素的值的范围均为 $1 \sim 9$。

(3) 矩阵中没有值相同的元素。

若仅有一种替换方案，则输出替换后的矩阵；若有多种替换方案，则输出 Too Many（保证至少有一种替换方案）。

举例说明

样例示意图如下。



题目分析

核心考点

- ✧ 全排列
- ✧ DFS

本题的做法很多，其中最为简单的方法莫过于使用全排列。

提示

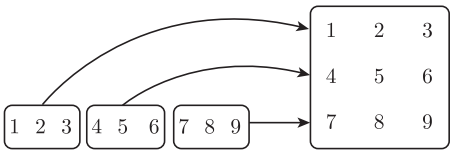
从 n 个不同元素中任取 m（$m \leqslant n$）个元素，按照一定的顺序排列起来，叫作从 n 个不同元素中取出 m 个元素的一个排列。

当 $m = n$ 时所有的排列情况叫全排列，比如 $1, 2, 3$ 的全排列有 $(1, 2, 3)$、$(1, 3, 2)$、$(2, 1, 3)$、$(2, 3, 1)$、$(3, 1, 2)$、$(3, 2, 1)$。

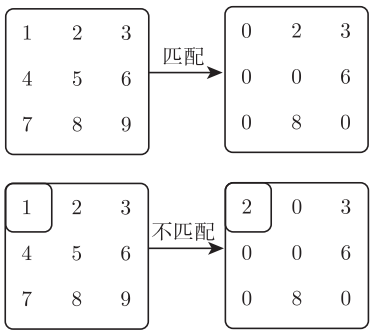
对于 n 个数 $(1, 2, 3, ..., n)$，全排列的规模为 $n!$。本题 n 的大小固定为 9，$9! = 362880$，规模较小，所以利用全排列是没有问题的，方法如下。

（1）通过 next_permutation 生成 $1 \sim 9$ 的全排列：$(1, 2, 3, 4, 5, 6, 7, 8, 9)$、$(1, 2, 3, 4, 5, 6, 7, 9, 8) \ldots$ 共 $9!$ 种。

（2）将所有生成的排列的第 $1 \sim 3$ 个数作为矩阵的第一行，第 $4 \sim 6$ 个数作为矩阵的第二行，第 $7 \sim 9$ 个数作为矩阵的第三行（转换为全排列矩阵），如下图所示。



（3）判断排列矩阵的每个部分是否和样例给定的矩阵相匹配（只看非零部分），如下图所示。若匹配，则说明样例给定的矩阵可以还原为该排列矩阵。



（4）对于匹配的排列矩阵，判断其是否为一个九宫幻方（每一行、每一列和每一条对角线的和相同）。若为九宫幻方，则将其记录，并判断当前的九宫幻方的个数是否大于 1，具体代码见参考代码 2-1-1。

参考代码 2-1-1　【时间复杂度为 $O(9 \times 9!)$】

```
#include<bits/stdc++.h>
using namespace std;
int p[10] , a[5][5] , b[5][5] , ans[5][5];
signed main()
{
```

```
    for(int i = 1 ; i <= 3 ; i ++) for(int j = 1 ; j <= 3 ; j ++) cin >> a[i][j]; //读入样例矩阵
    for(int i = 1 ; i <= 9 ; i ++) p[i] = i;
    int cnt = 0; // 统计九宫幻方的个数
    do{
      // 将排列转换为矩阵
       b[1][1] = p[1] , b[1][2] = p[2] , b[1][3] = p[3];
       b[2][1] = p[4] , b[2][2] = p[5] , b[2][3] = p[6];
       b[3][1] = p[7] , b[3][2] = p[8] , b[3][3] = p[9];
       // 判断排列矩阵和样例矩阵是否匹配
       bool flag = true; // flag = true 表示匹配， flag = false 表示不匹配
       for(int i = 1 ; i <= 3 ; i ++){
          for(int j = 1 ; j <= 3 ; j ++){
             if(!a[i][j]) continue ; // 只看非零部分
             if(a[i][j] != b[i][j]) flag = false;
          }
       }
       if(!flag) continue ; // 如果不匹配，就跳过
       // 判断排列矩阵是否是九宫幻方
       bool ok = true; // ok = true 表示排列矩阵是九宫幻方，ok = false 表示排列不是九宫幻方
       int sum = b[1][1] + b[2][2] + b[3][3]; // 取一条对角线的值
       if(sum != b[1][3] + b[2][2] + b[3][1]) continue ; // 判断另一条对角线的和是否等于 sum，不等于就跳过
       for(int i = 1 ; i <= 3 ; i ++){
          int tmp1 = 0, tmp2 = 0; // tmp1 表示行的和，tmp2 表示列的和
          for(int j = 1 ; j <= 3 ; j ++) tmp1 += b[i][j] , tmp2 += b[j][i];
          if(tmp1 != sum || tmp2 != sum) ok = false; // 如果行的和或列的和不等于 sum，则排列矩阵不是九宫幻方
       }
       if(!ok) continue ; // 如果不是九宫幻方，就跳过
       cnt ++ ; // 九宫幻方的个数+1
       if(cnt >= 2) return cout << "Too Many\n" , 0; // 九宫幻方的个数 >=2，直接输出 Too Many，结束程序
       for(int i = 1 ; i <= 3 ; i ++) for(int j = 1 ; j <= 3 ; j ++) ans[i][j] = b[i][j]; // 用 ans 记录下该九宫幻方
    }while(next_permutation(p + 1 , p + 1 + 9));
    // 程序没有结束则说明 cnt<=1。按照题意九宫幻方至少有一个，所以直接输出 ans 记录的矩阵即可
    for(int i = 1 ; i <= 3 ; i ++) for(int j = 1 ; j <= 3 ; j ++) cout << ans[i][j] << "\n"[j == 3];
    return 0;
}
```

本题也可使用 DFS 来处理。

通过题意能分析出题目给我们布置的任务有下面几项。

（1）将矩阵中为 0 的部分逐一修改为不重复的非零整数。

（2）判断修改完后的矩阵是否是九宫幻方。

（3）统计九宫幻方的个数。

对于这些任务，我们可以分为 4 个步骤来完成。

（1）将矩阵中为 0 的位置存储下来。

（2）将矩阵中已出现过的数打上标记。

（3）对于每个被存储下来的位置，尝试将其值修改为未被打上标记（未出现过）的数。

（4）待所有被存储下来的位置都被修改后，判断矩阵是否为九宫幻方。若为九宫幻方，则将其记录，并判断当前已出现的九宫幻方的个数是否大于 1。

参考代码 2-1-2　【时间复杂度为 $O(9 \times 9!)$】

```cpp
#include<bits/stdc++.h>
using namespace std;
int vis[10], a[5][5], ans[5][5];
int n, cnt;
pair<int, int>p[10];
bool check(){
    int sum = a[1][1] + a[2][2] + a[3][3]; // 取一条对角线的值
    if(sum != a[1][3] + a[2][2] + a[3][1]) return false; // 判断另一条对角线的和是否等于 sum，不等于则不是九宫幻方
    for(int i = 1 ; i <= 3 ; i ++) {
        int tmp1 = 0, tmp2 = 0; // tmp1 表示行的和，tmp2 表示列的和
        for(int j = 1 ; j <= 3 ; j ++) tmp1 += a[i][j], tmp2 += a[j][i];
        if(tmp1 != sum || tmp2 != sum) return false; // 如果行的和或列的和不等于 sum，则排列矩阵不是九宫幻方
    }
    return true; // 是九宫幻方
}
void dfs(int now){
    // now > n 表示所有被存储的位置都已经被修改
    if(now > n) {
        if(check()) { // 判断修改后的矩阵是否是九宫幻方
            cnt ++ ; // 九宫幻方的个数 + 1
            // 用 ans 记录下该九宫幻方
            for(int i = 1 ; i <= 3 ; i ++)
                for(int j = 1 ; j <= 3 ; j ++)
                    ans[i][j] = a[i][j];
        }
        return ;
    }
```

```
    int x = p[now].first , y = p[now].second;
    for(int k = 1 ; k <= 9 ; k ++) {
        if(vis[k]) continue ; // 如果 k 这个数已经存在，就不能被重复使用
        a[x][y] = k; // 尝试将该位置的值设置为 k
        vis[k] = 1; // k 被使用，为其打上标记
        dfs(now + 1); // 继续搜索
        // 回溯
        a[x][y] = 0;
        vis[k] = 0;
    }
}
signed main()
{
    for(int i = 1 ; i <= 3 ; i ++)
        for(int j = 1 ; j <= 3 ; j ++) {
            cin >> a[i][j]; // 读入样例矩阵
            if(!a[i][j]) p[++ n] = make_pair(i, j); // 如果值为0，就用 pair 存储下来
            vis[a[i][j]] = 1; // 将矩阵中已出现过的数打上标记
        }
    dfs(1); // 开始搜索
    if(cnt == 1) { // 九宫幻方的个数为 1，直接输出 ans 记录的矩阵
        for(int i = 1 ; i <= 3 ; i ++)
            for(int j = 1 ; j <= 3 ; j ++)
                cout << ans[i][j] << " \n"[j == 3];
    } else cout << "Too Many\n"; // 九宫幻方的个数为 2，则输出 Too Many
    return 0;
}
```

2.2 穿越雷区 (★)

题目信息

2015 年国赛 - 程序设计题

- ✧ C/C++ A 组第 4 题；C/C++ C 组第 5 题
- ✧ Java A 组第 4 题；Java B 组第 4 题

【问题描述】

X 星的坦克战车很奇怪，它必须交替地穿越正能量辐射区和负能量辐射区才能保持正常运转，否则将报废。

某坦克需要从 A 区到 B 区去（A,B 区本身是安全区，没有正能量或负能量特征），怎样走才能路径最短?

已知的地图是一个方阵，上面用字母标出了 A,B 区，其他区都标了正号或负号分别表示正、负能量辐射区。

例如：

$A+-+-$

$-+--+$

$-+++-$

$+-+-+$

$B+-+-$

坦克车只能水平或垂直方向上移动到相邻的区。

【输入格式】

第一行是一个整数 n，表示方阵的大小，$4 \leqslant n < 100$。

接下来是 n 行，每行有 n 个数据，可能是 $A,B,+,-$ 中的某一个，中间用空格分开。A,B 都只出现一次。

【输出格式】

输出一个整数，表示坦克从 A 区到 B 区的最少移动步数。

如果没有方案，则输出 -1。

【样例输入】

5

$A+-+-$

$-+--+$

$-+++-$

$+-+-+$

$B+-+-$

【样例输出】

10

懒人速读

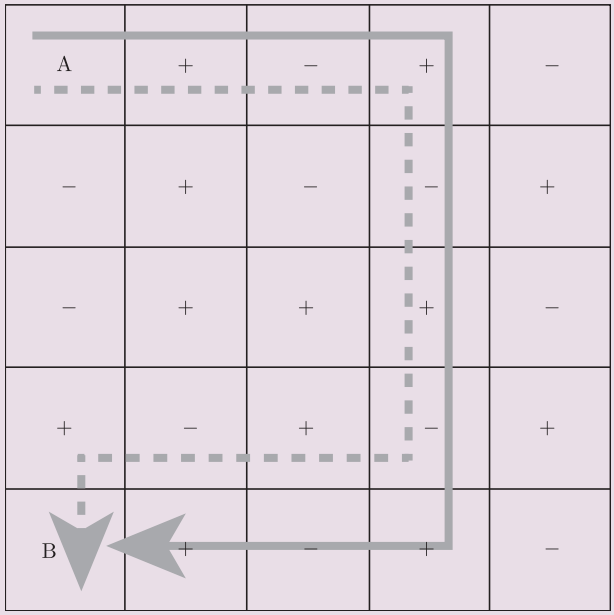
> 给定一个 $n \times n$ 的方阵，方阵由起点（记为 A）、终点（记为 B）、道路组成。
>
> 道路有两种属性：正（记为＋）、负（记为－）。
>
> 我们每次可以从一个位置移动到其相邻位置（上、左、下、右），但在移动的过程中需满足以下要求。
>
> (1) 不能离开方阵。
>
> (2) 必须正负交替移动：如果当前所处道路的属性为正，则下一步只能走到属性为负的道路或起点 A 或终点 B；如果当前所处道路的属性为负，则下一步只能走到属性为正

的道路或起点 A 或终点 B。

现在问从起点 A 走到终点 B 的最少移动次数是多少（若无法走到则输出 -1）？

举例说明

下图分别用实线与虚线渲染了样例对应的两条最优路线。



题目分析

核心考点

- ✧ BFS
- ✧ DFS

对于方阵上的某个位置，我们将其视作一个点，并用二维坐标来表示：(x, y) 表示第 x 行第 y 列。

对于点的正负能量值，可以使用二维数组 a 来表示。$a[x][y] = 1$ 表示点 (x, y) 的能量值为正，$a[x][y] = 0$ 表示点 (x, y) 的能量值为负，$a[x][y] = -1$ 表示点 (x, y) 没有能量特征（A 点、B 点）。

由于每次移动只能移动到相邻的位置，所以如果从 (x, y) 出发，只可能移动到 $(x + 1, y)$ 或 $(x - 1, y)$ 或 $(x, y + 1)$ 或 $(x, y - 1)$。需要注意以下两点。

（1）在移动的过程中，不能离开方阵（$1 \leqslant x, y \leqslant n$）。

（2）已经走过的位置，不重复走。

提示

因为第一次到达某个点和第二次到达某个点唯一的区别只在于它们的移动步数不同。根据 BFS 的性质，第一次到达该点的移动步数一定小于等于第二次到达该点的移动步数，所以为了保证移动步数最小，一个点只能经过一次。

（3）正负能量交替走。这是题目给定的限制条件。

那么在移动的过程中，我们需要维护哪些信息呢?

首先肯定要记录**移动时的坐标信息**，以此来判断是否到达了 B 点（也可用来判断当前位置的正负能量值）；其次需要记录**移动的步数**，这样到达 B 点后才能知道走了多少步。

分析了移动时需要记录的信息后，就可以制定一个解题步骤。

（1）读入并记录方阵的正负信息，记录 A 点和 B 点的坐标信息。

（2）从 A 点开始 BFS：

- 要求移动的正确性（不离开方阵、正负交替走）；
- 记录移动中的点的坐标、移动的步数、正负信息；
- 到达 B 点停止搜索，返回到达 B 点的移动步数。

参考代码 2-2-1

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int nex[4][2] = {{1, 0} , {-1, 0} , {0, 1} , {0, -1}}; // 表示移动时的4个方向
int n, a[N][N]; // a[i][j]表示点的正负信息。a[i][j]=1表示点(i,j)的能量值为正，a[i][j]=0表示点(i,j)的能量值为负,a[i][j] = -1表示点(i,j)为B
int vis[N][N]; // vis[i][j]表示在搜索的过程中点是否走过。vis[i][j]=1表示点(i,j)已经走过，vis[i][j]=0表示点(i,j)还没走过
pair<int, int>st, ed; // st记录点A的位置，ed存储点B的位置
struct node{
    int x, y, cnt;
};
bool check(int x, int y){
    if(x < 1 || x > n || y < 1 || y > n || vis[x][y]) return false;
    return true;
}
int bfs(int x, int y){
    pair<int, int>u = make_pair(x, y);
    queue<node>que;
    que.push(node{x, y, 0});
    vis[x][y] = 1;
    while(!que.empty()){
        node u = que.front();
```

```
        if(u.x == ed.first && u.y == ed.second) return u.cnt;
        que.pop();
        int x = u.x , y = u.y;
        for(int i = 0; i <= 3; i ++){
            // (tx , ty) 为 (x - 1 , y) 或 (x + 1 , y) 或 (x , y - 1) 或 (x , y + 1)
            int tx = x + nex[i][0];
            int ty = y + nex[i][1];
            // a[tx][ty] 的能量特征不能等于 a[x][y] 的能量特征
            if(!check(tx , ty) || a[tx][ty] == a[x][y]) continue ;
            vis[tx][ty] = 1;
            que.push(node{tx, ty, u.cnt + 1});
        }
    }
    return -1;
}
signed main(){
    cin >> n;
    for(int i = 1; i <= n; i ++){
        for(int j = 1; j <= n; j ++){
            char x;
            cin >> x;
            if(x == 'A') st.first = i, st.second = j, a[i][j] = -1;
            else if(x == 'B') ed.first = i, ed.second = j , a[i][j] = -1;
            else if(x == '+') a[i][j] = 1;
            else a[i][j] = 0;
        }
    }
    cout << bfs(st.first, st.second) << '\n';
    return 0;
}
```

这道题也可以用 DFS 来做，总体思路不变。

（1）读入并记录方阵的正负信息，记录 A 点、B 点的坐标。

（2）从 A 点开始 DFS：

- 要求移动的正确性（不离开方阵、正负交替走）。
- 记录移动中的点的坐标、移动的步数、正负信息。
- 到达 B 点停止搜索，返回到达 B 点的移动步数。

参考代码 2-2-2

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
```

```
4  int n , ans; // ans 记录答案
5  int a[N][N]; // a[i][j]表示点的正负信息。a[i][j] = 1表示点(i,j)的能量值为正，a[i][j]=0表示
       点(i,j)的能量值为负,a[i][j] = -1表示点(i,j)为B
6  int vis[N][N]; // vis记录到达 (x,y) 的移动步数
7  pair<int , int>st , ed; // st记录点A的位置，ed存储点B的位置
8  void dfs(int x , int y , int cnt) {
9      // x,y 表示当前点的位置，cnt 表示到达该点的移动步数
10     if(cnt > ans) return ; // 剪枝：如果移动步数大于 ans，那么该走法一定不是最优的，就没必要
           走下去了
11     if(cnt > vis[x][y]) return ; // 剪枝：如果到达(x,y)的移动步数大于 vis[x][y]，说明从起点
           走到(x,y)有更优的走法，那么该走法到达终点的移动步数一定不是最优的
12     if(x < 1 || x > n || y < 1 || y > n) return ; // 保证移动的正确性
13     if(x == ed.first && y == ed.second) {
14         ans = cnt; // 到达终点，记录（更新）答案
15         return ;
16     }
17     // 记录（更新）走到 (x,y) 的移动步数
18     vis[x][y] = cnt;
19     // a[tx][ty] 的能量特征不能等于 a[x][y] 的能量特征
20     int tx = x - 1 , ty = y;
21     if(a[tx][ty] != a[x][y]) dfs(tx , ty , cnt + 1);
22     tx = x + 1 , ty = y;
23     if(a[tx][ty] != a[x][y]) dfs(tx , ty , cnt + 1);
24     tx = x , ty = y - 1;
25     if(a[tx][ty] != a[x][y]) dfs(tx , ty , cnt + 1);
26     tx = x , ty = y + 1;
27     if(a[tx][ty] != a[x][y]) dfs(tx , ty , cnt + 1);
28 }
29 signed main(){
30     cin >> n;
31     ans = 0x3f3f3f3f;
32     for(int i = 1 ; i <= n ; i ++) for(int j = 1 ; j <= n ; j ++) vis[i][j] = 0x3f3f3f3f;
33     for(int i = 1; i <= n; i ++){
34         for(int j = 1; j <= n; j ++){
35             char x;
36             cin >> x;
37             if(x == 'A') st.first = i, st.second = j, a[i][j] = -1;
38             else if(x == 'B') ed.first = i, ed.second = j , a[i][j] = -1;
39             else if(x == '+') a[i][j] = 1;
40             else a[i][j] = 0;
41         }
42     }
```

```
43      dfs(st.first , st.second , 0);
44      cout << ans << '\n';
45      return 0;
46  }
```

2.3 小朋友崇拜圈 (★★)

题目信息

2017 年省赛 - 程序设计题

✧ C/C++ C 组第 8 题

【问题描述】

班里 N 个小朋友，每个人都有自己最崇拜的一个小朋友（也可以是自己）。

在一个游戏中，需要小朋友坐一个圈，每个小朋友都有自己最崇拜的小朋友在他的右手边。

求满足条件的圈最大多少人?

小朋友编号为 $1, 2, 3, \ldots, N$。

【输入格式】

输入第一行，一个整数 $N(3 < N < 10^5)$。

接下来一行 N 个整数，由空格分开。

【输出格式】

要求输出一个整数，表示满足条件的最大圈的人数。

【样例输入】

9

3 4 2 5 3 8 4 6 9

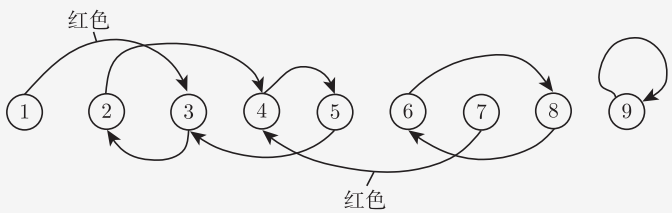
【样例输出】

4

【样例解释】

如下图所示，崇拜关系用箭头表示，红色表示不在圈中。

显然，最大圈是 [2 4 5 3] 构成的圈。



题目分析

核心考点

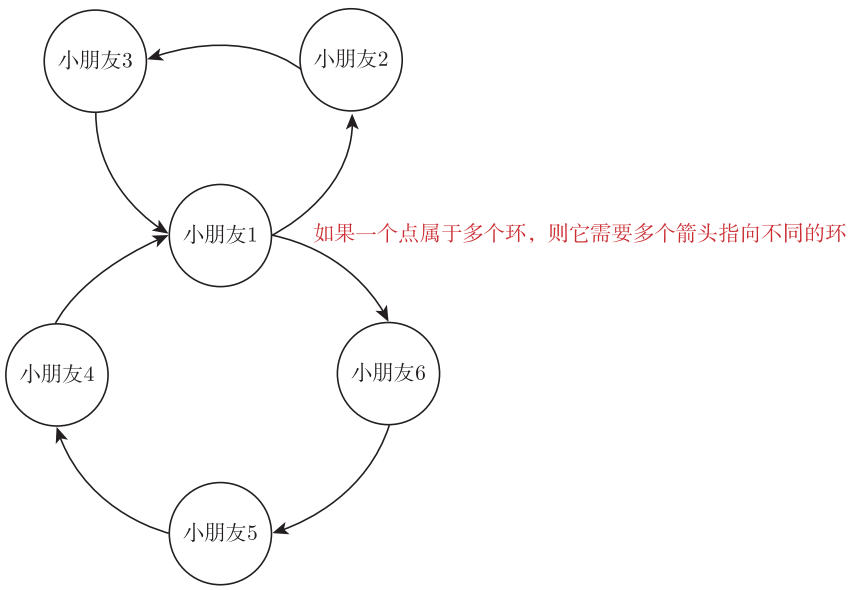
✧ DFS

这是一道经典的 DFS 找环问题。对于题目中提到的“小朋友”“圈”，我们分别用“点”“环”来解释它们的含义。

根据题意可知，每个小朋友可崇拜的对象只有一个（但一个小朋友可能被多个小朋友崇拜），进而可以得到一个信息：每个小朋友最多只可能属于一个环。

为什么呢?

我们可以用反证法证明：假设一个小朋友属于多个环，那么每个环内必然要有一个他的崇拜对象，即他要有多个崇拜对象，如下图所示。这并不符合题意，故假设不成立。



接下来，我们可以将小朋友用节点表示，小朋友之间的崇拜关系用箭头表示，以便构图分析。

假设小朋友之间的崇拜关系为 1->3，2->4，3->2，4->5，5->3，那么其构造顺序将如下页图所示（从一个节点开始不断移动直到遇上红色箭头）。

不难发现，当红色箭头出现后，图中出现了环，我们可以称红色箭头对应的边为**环边**。环边出现后，环也会出现，环的大小等于环边**两端节点之间的箭头数**。

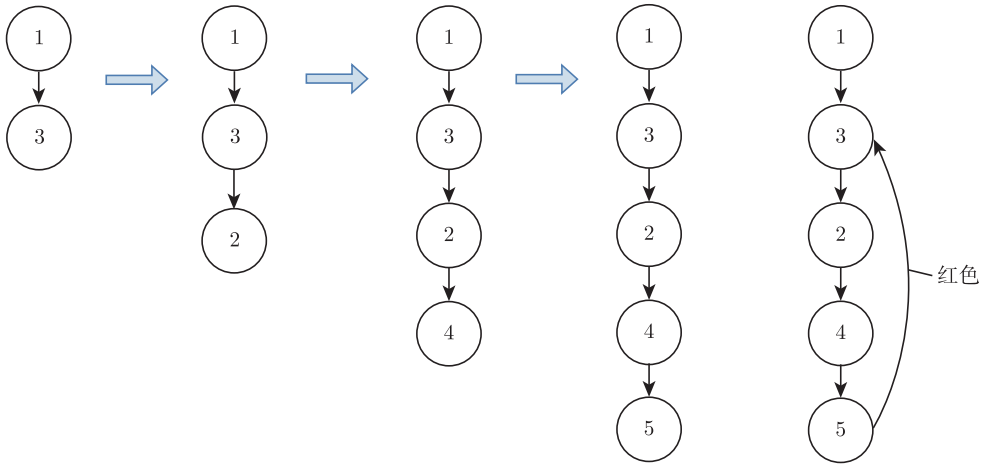
求出环的大小是我们解题的最终目的，确定了所有环的大小，就能确定答案。

按照上面的信息可知，求出环的大小的关键有两点：

（1）找到环边；

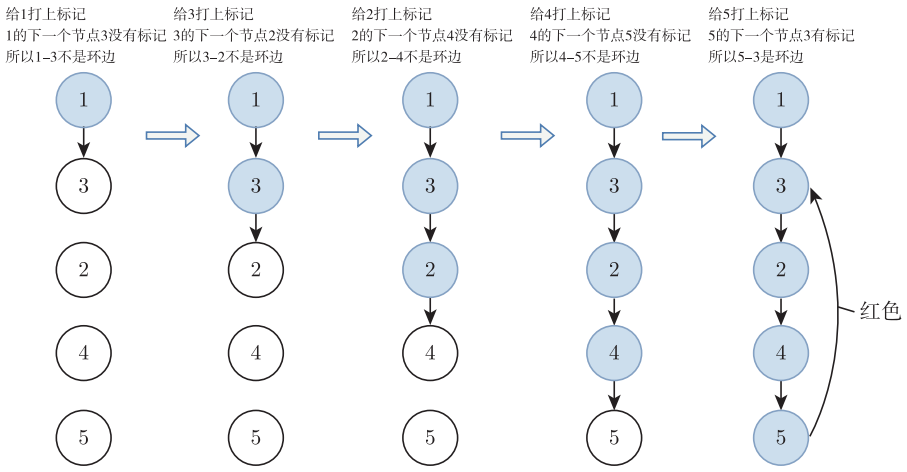
（2）求出环边两端节点之间的箭头数。

但是存在以下问题。



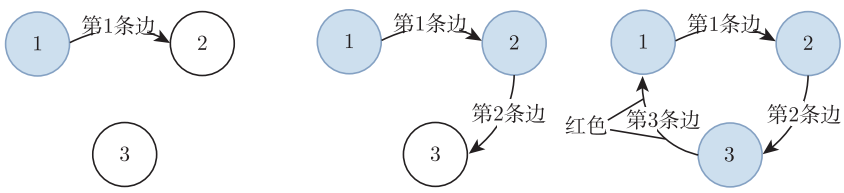
1. 怎么判断一条边是不是环边

答：很简单。环边指向的节点一定是已经出现了的节点，我们可以为节点打上标记来区分节点是否出现过，以此来判断一条边是否为环边。



2. 怎么求环边两端节点之间的箭头数

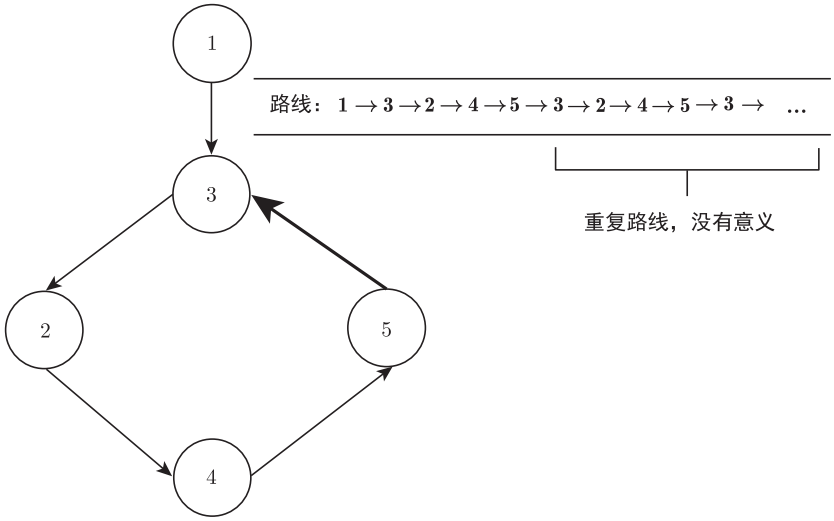
答：可以为每个箭头设定一个编号。对于环内的每一条边，它们的编号一定是连续的。因此，环内的箭头数 = 红色箭头编号 − 环的第一条箭头编号 +1。



环的大小=红色箭头的编号−环内第一条边的编号+1=3−1+1=3

3. 找到环后，是不是还需要继续移动

答：不需要。找到环后，如果接着移动那就相当于重新在这个环上走一遍，是没有意义的，如下图所示。所以找到环后就可以停下了。

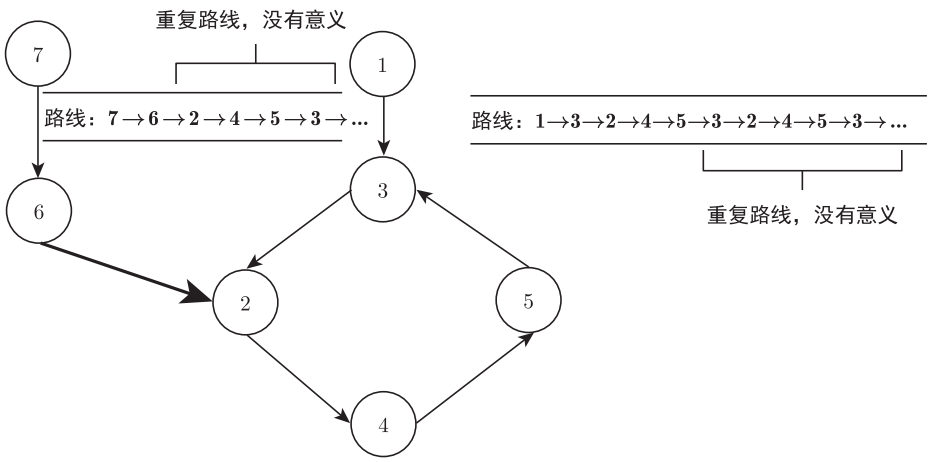


解决了上述 3 个问题，即可设计并实现 DFS 以解题：

（1）从每一个没有被标记的节点开始，一路走，直到出现环边，停止；

（2）走的同时为节点打上标记，并记录边的编号；

（3）当走到被其他路线标记过的点后，将重复走相同的路（见下图），没有意义，停止；



（4）出现环边后计算环的大小并更新答案。

参考代码 2-3　【时间复杂度为 $O(n)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
```

```cpp
int n , ans , nex[N] , number[N] , vis[N];
// nex[x] 表示 x 的崇拜对象
// number[x] 表示边 x->nex[x] 的编号
void dfs(int x , int cnt , int id){ // x表示当前走到的节点，cnt表示当前边的编号，id表示当前路线的编号
    if(vis[x] && vis[x] != id) return ; // 如果 x 已经被其他路线标记过，则返回
    if(vis[x]) { // 说明找到了环边
        ans = max(ans , (cnt-1) - number[x] + 1); // 更新答案：cnt-1表示上一条边的编号，即红色箭头的编号
        return ; // 结束递归
    }
    vis[x] = id; // 为节点 x 打上标记
    number[x] = cnt; // 为边 x->nex[i] 添加编号
    dfs(nex[x] , cnt + 1 , id); // 继续搜索
}
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++) cin >> nex[i];
    for(int i = 1 ; i <= n ; i ++) dfs(i , 1 , i);
    cout << ans << '\n';
    return 0;
}
```

2.4 迷宫与陷阱 (★★)

题目信息

2018 年国赛 - 程序设计题

✧ C/C++ C 组第 6 题

【问题描述】

小明在玩一款迷宫游戏，在游戏中他要控制自己的角色离开一间由 $N \times N$ 个格子组成的 2D 迷宫。

小明的起始位置在左上角，他需要到达右下角的格子才能离开迷宫。

每一步，他可以移动到上下左右相邻的格子中（前提是目标格子可以经过）。

迷宫中有些格子小明可以经过，我们用 '.'表示。

有些格子是墙壁，小明不能经过，我们用 '#'表示。

此外，有些格子上有陷阱，我们用'X'表示。除非小明处于无敌状态，否则不能经过。

有些格子上有无敌道具，我们用 '%'表示。

当小明第一次到达该格子时，自动获得无敌状态，无敌状态会持续 K 步。

之后如果再次到达该格子不会获得无敌状态了。

处于无敌状态时，可以经过有陷阱的格子，但是不会拆除/毁坏陷阱，即陷阱仍会阻止没有无敌状态的角色经过。

给定迷宫，请你计算小明最少经过几步可以离开迷宫?

【输入格式】

第一行包含两个整数 N, K $(1 \leqslant N \leqslant 1000, 1 \leqslant K \leqslant 10)$。

以下 N 行包含一个 $N \times N$ 的矩阵。

矩阵保证左上角和右下角是 `'.'`。

【输出格式】

一个整数表示答案。如果小明不能离开迷宫，输出 -1。

【样例输入】

```
5 3
...XX
##%#.
...#.
.###.
.....
```

【样例输出】

```
10
```

懒人速读

给定一个由 $N \times N$ 个格子组成的迷宫以及一个常数 K，迷宫的起点在左上角，终点在右下角，小明需要从起点走到终点。

迷宫中的格子有以下几种类型。

(1) 普通的格子（记为“.”）：可以经过。

(2) 墙壁（记为“#”）：无法经过。

(3) 拥有无敌道具的格子（记为“%”）：可以经过，经过后接下来的 K 步将处于无敌状态，但格子会变为普通的格子。

(4) 陷阱（记为“X”）：若处于无敌状态则可以经过，否则不可经过。

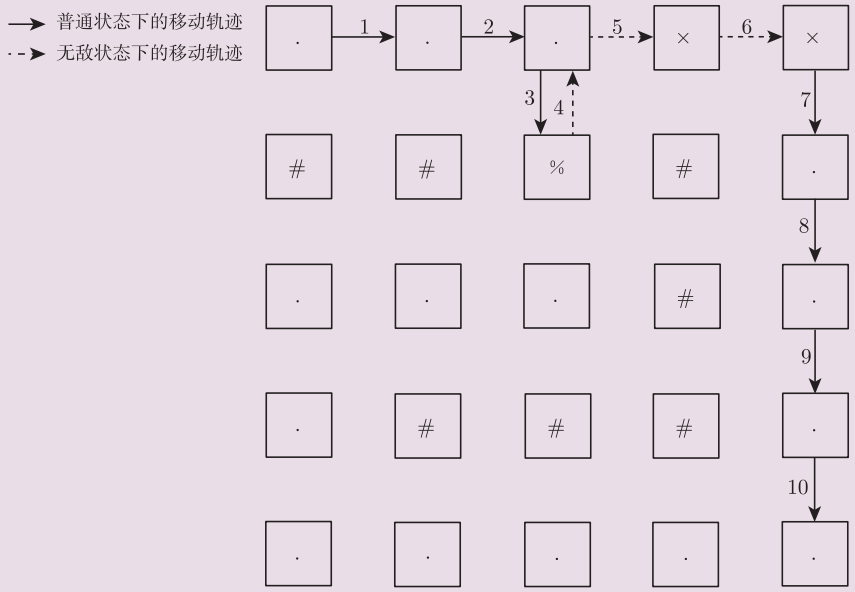
小明的每一步可以移动到当前位置上、下、左、右相邻的格子上，但前提是这些格子可以经过。

请问从起点走到终点的最少步数是多少（若不能离开，则输出 -1）？

保证起点和终点一定是普通的格子。

举例说明

样例示意图如下。



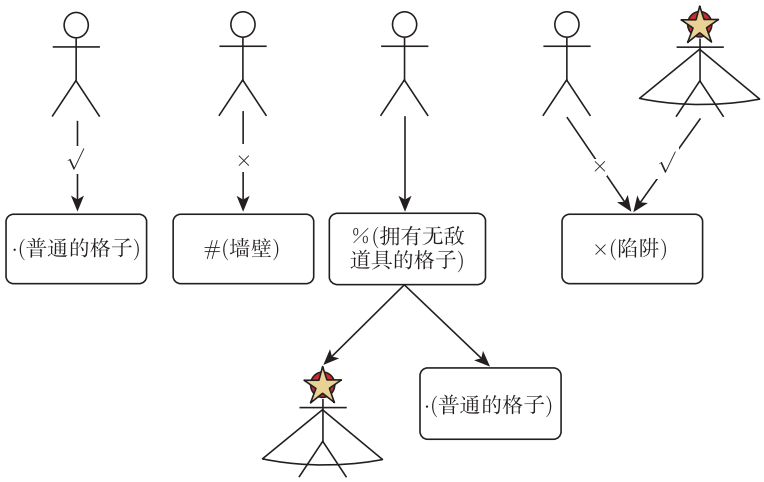
题目分析

核心考点

✧ BFS

这是一道 BFS 解迷宫的“变种题”。对于迷宫的格子，我们习惯性用二维坐标来表示，如 (x, y) 表示位于第 x 行第 y 列的格子。

在开始解题前，我们先来分析一下迷宫上格子的类型，如下图所示。



（1）.：普通的格子，小明可以经过。

（2）#：墙壁，小明不能经过。

（3）%：拥有无敌道具的格子，小明经过该格子后会进入无敌状态并持续 K 步，但格子会变为普通的格子。

（4）X：陷阱，如果处在无敌状态则可以经过，否则不可以经过。

分析了迷宫的结构后，我们来思考移动时需要记录的信息：

（1）需要记录移动的坐标，这样才能判断是否到达终点；

（2）需要记录移动的步数，这样才能知道到达终点后走了多少步；

（3）需要记录移动到下一个位置时的状态（无敌状态或普通状态），这样才能判断能否经过陷阱。

我们可以定义结构体来表示它们，代码如下。

参考代码 2-4-1

```
struct node{
  int x , y; // 表示移动到的位置
  int cnt;  // 表示移动的步数
  int status; // 表示接下来的 status 步都将保持无敌状态。status > 0 则处于无敌状态，status = 0 则处于普通状态
};
```

那么对于这些信息，如果从一个位置 (x, y) 移动到下一个位置，会发生哪些变化呢?

（1）坐标会变为 $(x+1, y)$ 或 $(x-1, y)$ 或 $(x, y+1)$ 或 $(x, y-1)$。

（2）移动的步数会 $+1$。

（3）移动到下一个位置时的状态：

- 如果是普通状态，则有可能保持普通状态，也可能变为无敌状态；
- 如果是无敌状态，则有可能保持无敌状态，也可能变为普通状态。

具体的状态变化分以下几种情况。

1. 普通状态

（1）如果下一个位置是拥有无敌道具的格子，则状态变为无敌状态，且接下来的 K 步都将保持无敌状态。

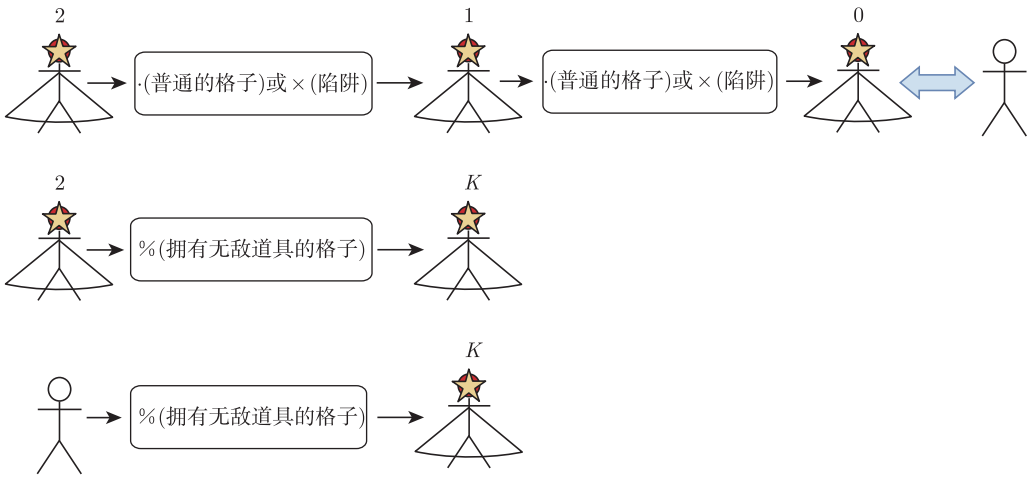
（2）如果下一个位置是普通的格子，则状态不变。

2. 无敌状态

（1）如果下一个位置是拥有无敌道具的格子，则状态保持无敌状态，且接下来的 K 步都将保持无敌状态。

（2）如果下一个位置是普通的格子或陷阱，就要判断距离上一次成为无敌状态已经走了多少步。

- 如果已经走了 K 步，则状态变为普通状态。
- 如果还没走 K 步，则状态保持普通状态。



确定了移动时需要记录的信息、移动时信息的变化，我们再来思考一个问题，即已经走过的点还可以继续走吗?

答案肯定是可以（只要移动是合法的就可以走）。但是，有没有再走一次的必要呢?

判断有没有必要即判断第二次走到该点时的信息是否优于第一次走到这个点时的信息，如果优于则有必要，反之没有必要。

那么怎么判断移动信息的优劣呢? 显然，**移动的步数越少越好，可保持无敌的状态越久越好**。

假设上一次走到 (x,y) 时，有以下两种情况。

（1）移动的步数为 cnt1。

（2）接下来的 status1 步都将保持无敌状态。

此次走到 (x,y) 时有以下两种情况。

（1）移动的步数为 cnt2。

（2）接下来的 status2 步都将保持无敌状态。

那么会有以下几种情况。

（1）cnt1 < cnt2，status1 > status2：由于此次移动的步数、可保持无敌状态的步数都劣于上一次，所以完全没必要再走一次。

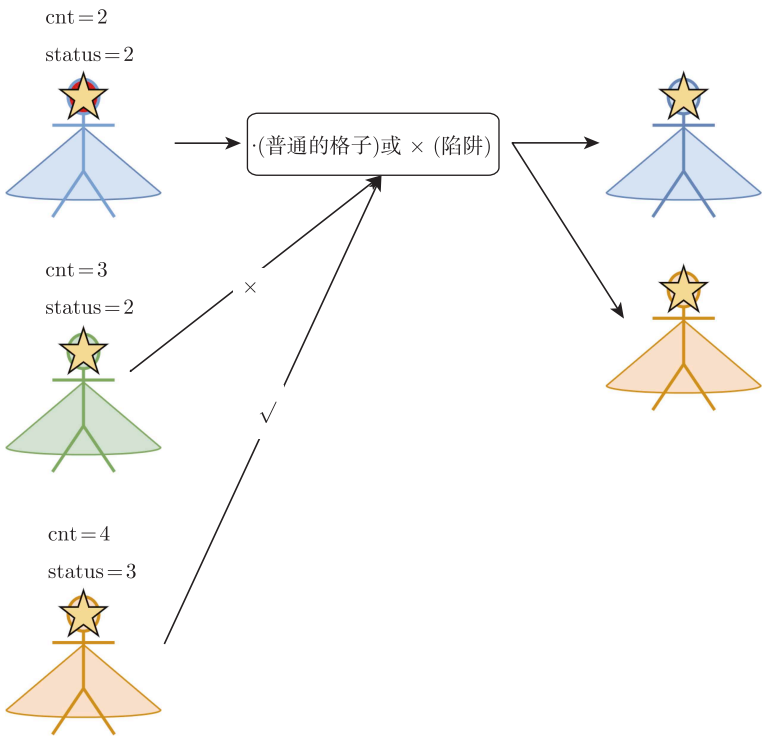
（2）cnt1 > cnt2，stauts1 < status2：由于此次移动的步数、可保持无敌状态的步数都优于上一次，所以是有必要再走一次的。

（3）cnt1 > cnt2，stauts1 < stauts1：此次移动的步数优于上一次，但可保持无敌状态的步数劣于上一次，无法判断哪种情况更优。为了保险起见，还是有必要再走一次的。

（4）cnt1 < cnt2，status1 > status2：此次移动的步数劣于上一次，但可保持无敌状态的步数优于上一次，无法判断哪种情况更优。为了保险起见，还是有必要再走一次的。

根据 BFS 的性质，一定满足第一次走到 (x,y) 的移动步数 < 第二次走到 (x,y) 的移动步数 < 第三次走到 (x,y) 的移动步数 < ……

因此，我们只要对 status 作出判断就好，如下图所示。



完成上面的分析与思考后，就可以设计具体的解题步骤了。

（1）读入迷宫，并用 $a[][]$ 来存储各个格子的类型。

- 普通的格子用 0 表示（$a[x][y]=0$）。
- 拥有无敌道具的格子用 1 表示（$a[x][y]=1$）。
- 陷阱用 2 表示（$a[x][y]=2$）。
- 墙壁用 3 表示（$a[x][y]=3$）。

（2）从点 (1,1) 开始 BFS，要求如下。

- 要求移动的正确性（不离开迷宫；不能经过墙壁；普通状态下不能经过陷阱）。
- 保证移动的必要性（对于没走过的格子，有必要走一次；对于走过的格子，判断有没有再走一次的必要）。
- 记录移动中的点的坐标、移动的步数、可保持无敌状态的步数。
- 到达 (n,n) 点停止搜索，返回到达 (n,n) 点的移动步数。

参考代码 2-4-2

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e3 + 10;
struct node{
  int x , y; // 表示移动到的位置
  int cnt;  // 表示移动的步数
```

```
 int status; // 表示接下来的 status 步都将保持无敌状态。status > 0 则处于无敌状态，status = 0 则处于普通状态
};
int n , k;
int nex[4][2] = {{1 , 0} , {-1 , 0} , {0 , 1} , {0 , -1}}; // 表示移动时的4个方向
int a[N][N] , vis[N][N] , s[N][N]; // a[][] 表示格子的类型，vis[][] 标记格子是否被访问过，s[][] 记录经过格子时最大的可保持无敌状态的步数
bool check(int x , int y , int status){
    // 不能离开迷宫，不能经过墙壁
    if(x < 1 || x > n || y < 1 || y > n || a[x][y] == 3) return false;
    // 如果不是无敌状态，不能经过陷阱
    if(a[x][y] == 2 && !status) return false;
    return true;
}
int bfs(){
    queue<node>que;
    que.push(node{1 , 1 , 0 , 0}); // 从 (1,1) 出发
    vis[1][1] = 1; // 标记 (1,1) 被访问过
    while(!que.empty()){
        node u = que.front();
        que.pop();
        if(u.x == n && u.y == n) return u.cnt; // 到达 (n,n) 点则停止搜索
        for(int i = 0 ; i <= 3 ; i ++){
            int tx = u.x + nex[i][0];
            int ty = u.y + nex[i][1];
            if(!check(tx , ty , u.status)) continue ;
            int status = max(0 , u.status - 1);
            if(a[tx][ty] == 1){ // 陷阱
                vis[tx][ty] = 1;
                a[tx][ty] = 0; // 陷阱走过就变为普通的格子
                que.push(node{tx , ty , u.cnt + 1 , k});
            }
            else { //普通的格子
                if(!vis[tx][ty]) { // 如果没有走过，那么有必要走一次
                    vis[tx][ty] = 1;
                    que.push(node{tx , ty , u.cnt + 1 , status});
                    continue ;
                }
                if(status <= s[tx][ty]) continue ; // 可保持的无敌状态劣于上一次，没有必要再走一次
                // 可保持的无敌状态优于上一次，有必要再走一次
                s[tx][ty] = status;
```

```
46              que.push(node{tx , ty , u.cnt + 1 , max(0 , status)});
47          }
48      }
49   }
50   return -1;
51}
52signed main(){
53   cin >> n >> k;
54   for(int i = 1 ; i <= n ; i ++){
55      for(int j = 1 ; j <= n ; j ++){
56         char x;
57         cin >> x;
58         if(x == '%') a[i][j] = 1;
59         else if(x == 'X') a[i][j] = 2;
60         else if(x == '#') a[i][j] = 3;
61      }
62   }
63   cout << bfs() << '\n';
64   return 0;
65}
```

2.5　扫地机器人 (★★)

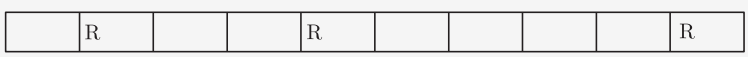
题目信息

2019 年省赛 - 程序设计题

- ✧ C/C++ C 组第 10 题
- ✧ Java C 组第 10 题

【问题描述】

小明公司的办公区有一条长长的走廊，由 N 个方格区域组成，如下图所示。



走廊内部署了 K 台扫地机器人，其中第 i 台在第 A_i 个方格区域中。已知扫地机器人每分钟可以移动到左右相邻的方格中，并将该区域清扫干净。

请你编写一个程序，计算每台机器人的清扫路线，使得

（1）它们最终都返回出发方格。

（2）每个方格区域都至少被清扫一遍。

（3）从机器人开始行动到最后一台机器人归位花费的时间最少。

注意多台机器人可以同时清扫同一方块区域，它们不会互相影响。

输出最少花费的时间。在上图所示的例子中，最少花费时间是 6。第一台路线：2-1-2-3-4-3-2，清扫了 $1,2,3,4$ 号区域。第二台路线 5-6-7-6-5，清扫了 $5,6,7$。第三台路线 10-9-8-9-10，清扫了 $8,9$ 和 10。

【输入格式】

第一行包含两个整数 N,K。

接下来 K 行，每行一个整数 A_i。

其中，$1 \leqslant K < N \leqslant 10^5, 1 \leqslant A_i \leqslant N$。

【输出格式】

输出一个整数表示答案。

【样例输入】

```
10 3
5
2
10
```

【样例输出】

```
6
```

懒人速读

n 个房间连成一排，第 i 个房间的编号为 i。

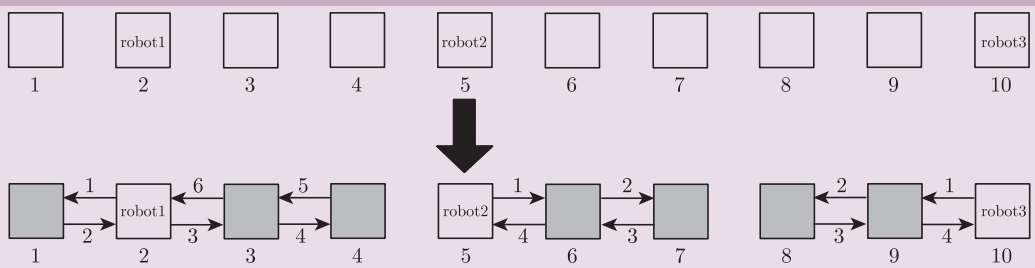
有 K 台机器人，起初它们分别位于 $a_1, a_2, \ldots, a_k$ 号房间。

机器人每分钟可以移动到左右相邻的房间中，并将房间清扫干净（多台机器人可同时启动，也可同时清理同一个房间）。

我们可以随意分配每台机器人的清扫路线，但要求这些机器人最终都要回归初始时所在的房间。

问题是，满足所有房间都至少被清扫一次且所有机器人都回归初始房间最少需要花费多长时间？

举例说明



题目分析

核心考点

- ✧ 贪心
- ✧ 枚举
- ✧ 二分

本题的 3 个关键信息包括清扫的时间、清扫的方法、清扫的区域面积（方格个数）。

其中清扫的（最短）时间是我们要求的答案，清扫的区域面积（方格个数）是已知的，而清扫的方法由我们自己决定。

显然，在同等时间下，使用效率高的清扫方法，清扫的区域面积（方格个数）就会多。反过来同理，要让所有区域都至少被清扫一遍，且清扫的时间最短，就需要用最有效率的清扫方法。

那么问题来了，什么样的方法是最有效率的呢?

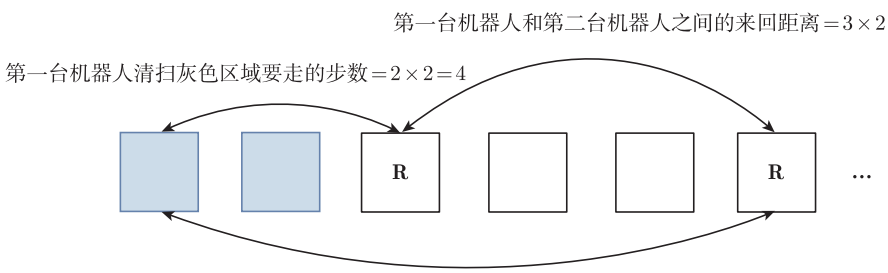
1. 求最优的清扫方法

假设有 k 台机器人，它们从左到右的编号分别为 $1,2,\ldots,k$，每台机器人清扫的时间为 t 分钟。

按照顺序，我们会从 1 号机器人左边的区域（可能为空）开始处理。

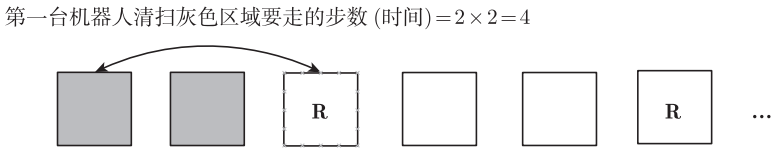
我们有 $1\sim k$ 台机器人，要选择哪台机器人去清扫这部分区域呢?

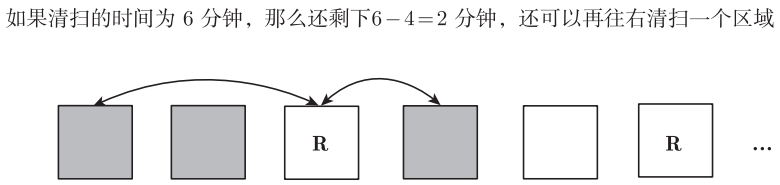
显然，让 1 号机器人去清扫效率会是最高的。如果让其他机器人清扫，就需要多花费一些时间（注意，此时只考虑第一台机器人左边的区域，而不考虑其他区域），如下图所示。



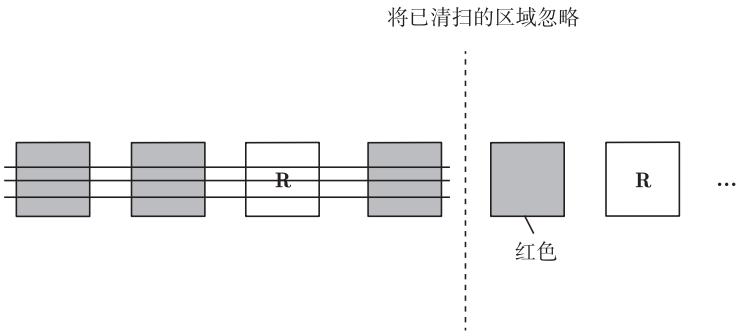
在只考虑清扫灰色区域时，第二台机器人要比第一台机器人多走一段距离

1 号机器人清扫完它左边的区域后，需要返回原位。返回原位后，如果还剩有时间，我们不能浪费，要让它继续去清扫它右边的区域，如下页图所示。这样可以减少下一台机器人需要往左清扫的范围（即减少下一台机器人的清扫时间）。





等清理环节结束后，我们将已清扫完的区域忽略，如下图所示。



不难发现，在忽略了已清扫完毕的区域后，原来的灰色区域相当于现在的红色区域，原来的第一台机器人相当于现在的第二台机器人，回到了和开始几乎一样的情况。

因此，可根据贪心思想确定清扫方法，即让每台机器人都要优先清扫其左边还未扫过的格子，然后再往右清扫，以保证每次清扫的效率都最高。

确定了清扫的方法、清扫的区域面积，我们就可以求清扫的时间。

2. 求最少的清扫时间

在开始求最少的清扫时间之前，我们需要明确以下两点。

（1）如果所有机器人在时间 t 内都用了效率最高的清扫方法，却没能清扫完所有区域，那么问题就只能出在“t 太小（时间不够）”上。

（2）如果所有机器人都用了效率最高的清扫方法，那么给的时间越多，可清扫的区域就会越多（在达到某个临界点之后，随着时间的增加，可清扫的区域个数将不会变化）。

由上，我们可以得出对于一个时间 t，它要想作为答案须满足以下两个条件。

（1）所有的机器人在 t 分钟内可清扫所有区域。

（2）t 是满足所有条件 1 的时间中最小的一个。

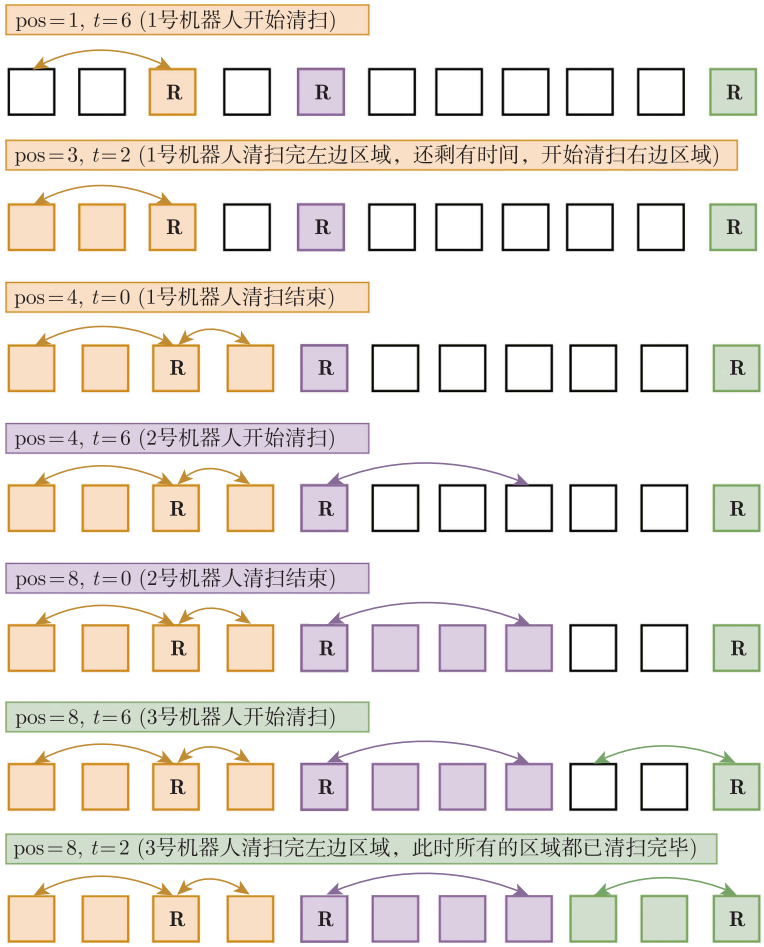
对于条件 1，我们可以模拟清扫的方法，从 1 号机器人开始一个个清扫区域。在所有机器人清扫结束后判断是否清扫了所有区域即可（见下页图），代码如下。

参考代码 2-5-1

```
定义 pos，初始值为 0。pos 用来表示 1~pos 区域已被清扫，pos+1~n 区域未被清扫

for(遍历 k 台机器人) {
  if(pos 小于第 i 台机器人的位置) {
    往左清扫，需要花费的时间为 2 * (a[i] - pos - 1)
  }
  if(还剩有时间) {
    往右清扫，可清扫的区域的个数为剩余的时间/2
  }
  更新 pos
}

判断 pos 是否大于等于 n
```



由于需要遍历所有机器人，所以复杂度为 $O(k)$。

对于条件 2，我们可以由小到大去枚举 t，这样第一个满足条件 1 的 t 就是答案。

参考代码 2-5-2

```
for(从 1 开始枚举 t) {
  if( t 满足条件 1 )
    t 为答案，结束枚举
}
```

不过需要注意的是，当题目给定的测试数据较为极端时，使用最优的清扫方法也可能需要接近 $2 \times n$ 的时间才可清扫完所有区域，即我们枚举的次数将接近 $2 \times n$。那么要想同时满足条件 1、条件 2，需要的时间复杂度就为 $O(n \times k)$。

这并不是最好的做法，我们来考虑一下如何优化复杂度。

已知本题的目的是求在使用最高效方法的前提下，清扫完所有区域所需要花费的最少时间。而根据上文分析可得：随着时间增加，可清理的区域只会多不会少。这说明时间是满足单调性的。

那么，我们就可以将用枚举法求 t 替换为用二分法求 t 来降低时间复杂度。

参考代码 2-5-3 【时间复杂度为 $O(k \log n)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
int n , k , a[N]; // a[i] 表示第 i 台机器人的位置
bool check(int mid){
    int pos = 0; // pos 用来表示 1~pos 区域已被清扫，pos+1~n 区域未被清扫
    for(int i = 1 ; i <= k ; i ++){ // 遍历 k 台机器人
        int t = mid;
        if(pos < a[i]) t -= (a[i] - pos - 1) * 2; // 往左清扫，需要花费的时间为 2 * (a[i] -
            pos - 1)
        if(t < 0) return false; // 如果时间小于 0，说明无法清扫完左边的区域，时间不够
        pos = a[i] + t / 2; // 如果还剩有时间，继续向右清扫
    }
    if(pos < n) return false; // 如果没有清扫完所有的区域，则说明时间不够
    return true;
}
signed main(){
    cin >> n >> k;
    for(int i = 1 ; i <= k ; i ++) cin >> a[i];
    sort(a + 1 , a + 1 + k); // 位置排序
    int l = 0 , r = 2 * n , ans = 2 * n;
    while(l <= r){
        int mid = l + r >> 1;
        if(check(mid)) ans = mid , r = mid - 1;
        else l = mid + 1;
```

```
25      }
26      cout << ans << '\n';
27      return 0;
28  }
```

2.6　123 (★★)

题目信息

2021 年国赛 - 程序设计题

- ✧ C/C++ A 组第 5 题；C/C++ B 组第 6 题；C/C++ C 组第 7 题
- ✧ Java A 组第 5 题；Java B 组第 6 题；Java C 组第 7 题
- ✧ Python 组第 6 题

【问题描述】

小蓝发现了一个有趣的数列，这个数列的前几项如下：

$1, 1, 2, 1, 2, 3, 1, 2, 3, 4, \ldots$

小蓝发现，这个数列前 1 项是整数 1，接下来 2 项是整数 1 至 2，接下来 3 项是整数 1 至 3，接下来 4 项是整数 1 至 4，以此类推。

小蓝想知道，这个数列中，连续一段的和是多少。

【输入格式】

输入的第一行包含一个整数 T，表示询问的个数。

接下来 T 行，每行包含一组询问，其中第 i 行包含两个整数 l_i 和 r_i，表示询问数列中第 l_i 个数到第 r_i 个数的和。

【输出格式】

输出 T 行，每行包含一个整数表示对应询问的答案。

【样例输入】

```
3
1 1
1 3
5 8
```

【样例输出】

```
1
4
8
```

【评测用例规模与约定】

对于 10% 的评测用例，$1 \leqslant T \leqslant 30, 1 \leqslant l_i \leqslant r_i \leqslant 100$。

对于 20% 的评测用例，$1 \leqslant T \leqslant 100, 1 \leqslant l_i \leqslant r_i \leqslant 1000$。
对于 40% 的评测用例，$1 \leqslant T \leqslant 1000, 1 \leqslant l_i \leqslant r_i \leqslant 10^6$。
对于 70% 的评测用例，$1 \leqslant T \leqslant 10000, 1 \leqslant l_i \leqslant r_i \leqslant 10^9$。
对于 80% 的评测用例，$1 \leqslant T \leqslant 1000, 1 \leqslant l_i \leqslant r_i \leqslant 10^{12}$。
对于 90% 的评测用例，$1 \leqslant T \leqslant 10000, 1 \leqslant l_i \leqslant r_i \leqslant 10^{12}$。
对于所有评测用例，$1 \leqslant T \leqslant 100000, 1 \leqslant l_i \leqslant r_i \leqslant 10^{12}$。

懒人速读

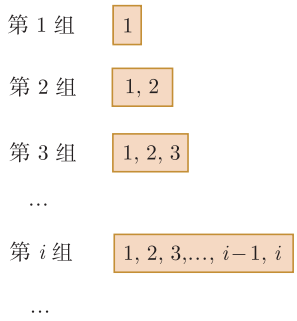
有一个序列，序列的第 1 项为整数 1，接下来 2 项为整数 1 至 2，接下来 3 项是整数 1 至 3，接下来 4 项是整数 1 至 4，依此类推，如 $1,1,2,1,2,3,1,2,3,4,\ldots$ 现给定 T 组询问，每组询问包含两个整数 l,r，要求出序列中第 l 项到第 r 项的和。

题目分析

核心考点

- 枚举；二分
- 前缀和

我们可以按照数列的规律将数列分为若干组：第 1 组为 1；第 2 组为 $1,2$；第 3 组为 $1,2,3$；...；第 i 组为 $1,2,3,\ldots i-1,i$，如下图所示。



方式 1：能拿 20% 分数的做法

题目要我们求出区间 $[l,r]$ 的和。

由于数据范围很小，所以我们可以按照数列变化的规律模拟出第 $l \sim r$ 项的值，然后记录第 $l \sim r$ 项的和就可以解决。

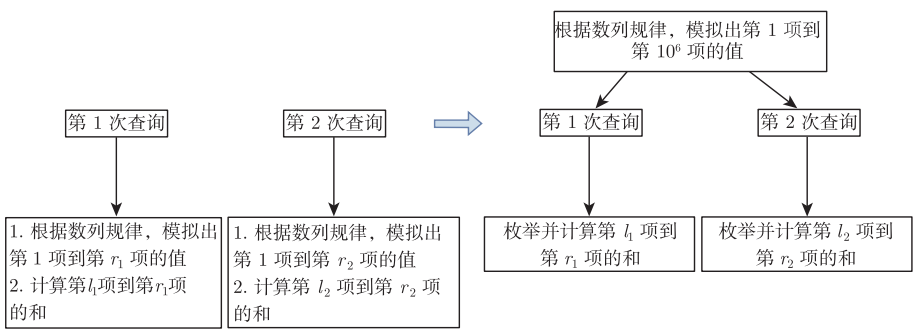
模拟出第 $l \sim r$ 项的复杂度为 $O(r)$。因为一共有 T 次查询，所以总的时间复杂度为 $O(Tr)$。

参考代码 2-6-1

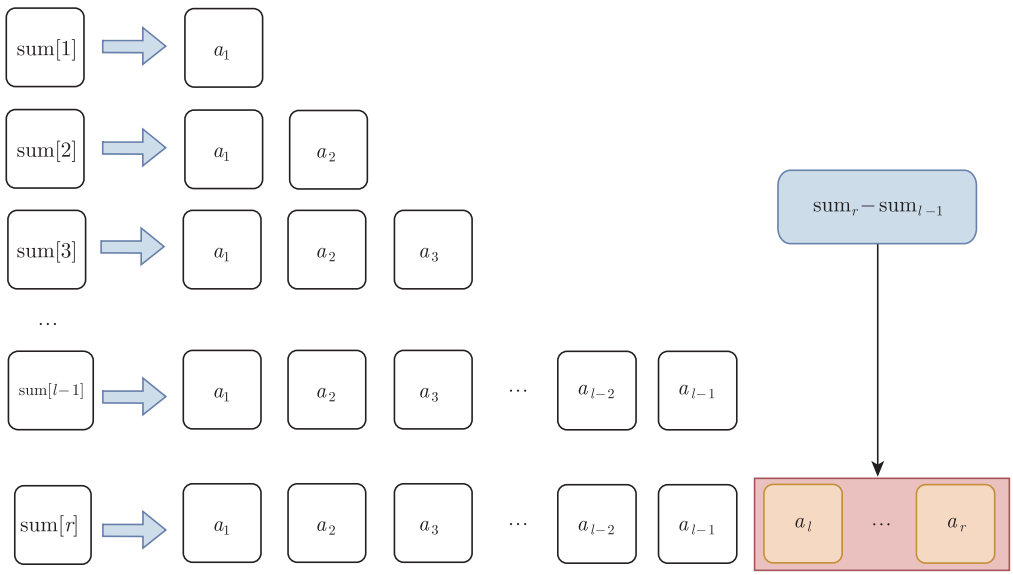
```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int a[N] , sum[N]; // a[] 存储数列的每一项， sum[] 为 a[] 的前缀和数组
signed main(){
  int T = 1;
  cin >> T;
  int up = 1 , x = 1; // x 表示数列第i项的值（从第1项开始），up 表示第i项所在组的编号（从第1组开始）
  for(int i = 1 ; i <= 1000000 ; i ++){
    a[i] = x;
    x ++ ;
    if(x > up) up ++ , x = 1; // 模拟数列变化的规律，当 x > up 时，进入下一组
  }
  for(int i = 1 ; i <= 1000000 ; i ++) sum[i] = sum[i - 1] + a[i];
  while(T --){
    int l , r;
    cin >> l >> r;
    cout << sum[r] - sum[l - 1] << '\n';
  }
  return 0;
}
```

方式 2：能拿 40% 分数的做法

按照 20% 分数的做法，求解一次区间 $[l,r]$ 的和需要的复杂度为 $O(r)$。如果我们只进行一次查询，固然没什么问题。但若涉及多次查询，这种方法就非常耗费时间。那么，有没有什么办法可以优化查询的复杂度呢？我们来分析一下。本题并没有涉及修改操作，即对于每次查询来说，数列每一项的值都是相同的。既然如此，我们不妨在所有查询开始前就模拟出数列第 $1 \sim 10^6$ 项的值，并将这些值分别存储在 $a_1 \sim a_{10^6}$ 中，如下图所示。这样，在查询过程中，我们就可以直接枚举 $a_{l\sim r}$ 并统计答案了。



不过这样的优化力度并不够。在最坏的情况下，查询区间 $[l,r]$ 的和需要的复杂度依然为 $O(r)$。那有没有什么办法可以降低复杂度呢？我们可以使用前缀和优化，即创建 sum[] 数组，并令 $\text{sum}[1]=a_1$，$\text{sum}[2]=a_1+a_2=\text{sum}[1]+a_2$，$\text{sum}[3]=a_1+a_2+a_3=\text{sum}[2]+a_3$，…，$\text{sum}[i]=\text{sum}[i-1]+a_i$，这样 $a_l+a_{l+1}+\ldots+a_r$ 就可以用 $\text{sum}_r-\text{sum}_{l-1}$ 代替，如下图所示。



我们同样可以在所有查询开始前求出 sum[] 数组。求出 sum[] 后，每次查询区间 $[l,r]$ 的和需要的复杂度仅为 $O(1)$。

参考代码 2-6-2　【时间复杂度为 $O(T+10^6)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int a[N] , sum[N]; // a[] 存储数列的每一项， sum[] 为 a[] 的前缀和数组
signed main(){
  int T = 1;
  cin >> T;
  int up = 1 , x = 1; // x 表示数列第i项的值（从第1项开始），up 表示第i项所在组的编号（从第1
      组开始）
  for(int i = 1 ; i <= 1000000 ; i ++){
    a[i] = x;
    x ++ ;
    if(x > up) up ++ , x = 1; // 模拟数列变化的规律，当 x > up 时，进入下一组
  }
  for(int i = 1 ; i <= 1000000 ; i ++) sum[i] = sum[i - 1] + a[i];
  while(T --){
    int l , r;
```

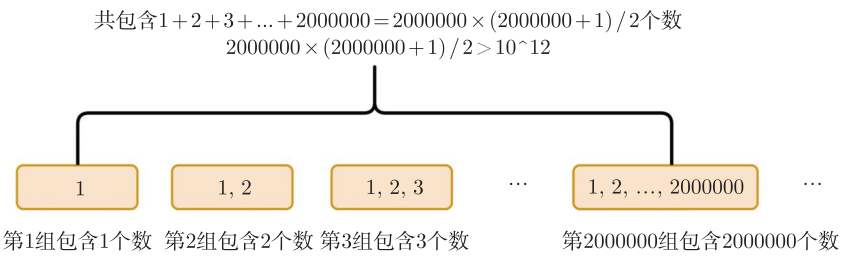
```
    cin >> l >> r;
    cout << sum[r] - sum[l - 1] << '\n';
  }
  return 0;
}
```

方式 3：满分做法

此时数据范围较大，在有限的时间内我们无法构造出数列的所有项，无法得到 sum[] 数组。因此，要重新考虑如何求解。

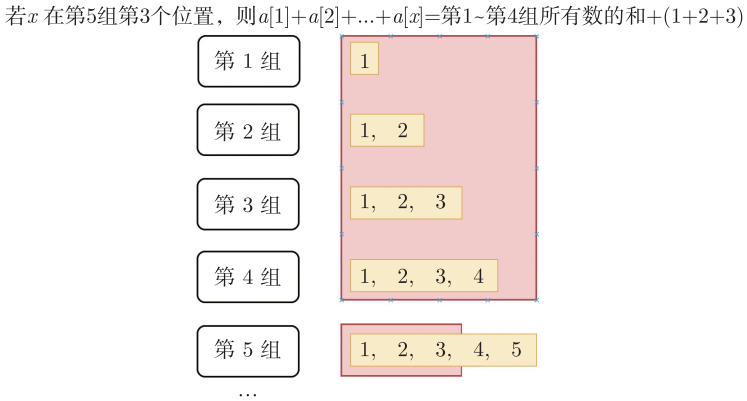
回到最开始，我们将数列分成了若干组，其中第 1 组有 1 个数、第 2 组有 2 个数、……、第 i 组有 i 个数，那么前 i 组就共有 $1+2+\ldots+i=\dfrac{i\times(i+1)}{2}$ 个数。

可以发现，使用不超过 2×10^6 组就能包含数列的前 10^{12} 个数，因此，我们只需关注前 2×10^6 组，如下图所示。



我们知道，数列中的每一项必然属于某一组中的某一位置，那么不妨让我们假设数列的第 x 项位于第 i 组的第 j 个位置。

根据分组方式容易发现，我们在能拿 **40%** 分数的做法中求得 $\text{sum}[x]$（$\text{sum}[x]=a[1]+a[2]+\ldots+a[x]$）可以用前 $i-1$ 组的所有数的和加上第 i 组的前 j 个数的和表示，即 $\text{sum}[x]=$ 第 1 组所有数的和 + 第 2 组所有数的和 + 第 3 组所有数的和 $+\ldots+$ 第 $i-1$ 组所有数的和 $+(1+2+3+\ldots+j)$。



对于前 i 组所有数的和，我们用数组 pre[] 表示。根据它的含义，可以轻松得到 pre[i] 的递推式。

$$\begin{aligned}
&\text{pre}[1] = 1\\
&\text{pre}[2] = (1) + (1+2) = \text{pre}[1] + (1+2)\\
&\text{pre}[3] = (1) + (1+2) + (1+2+3) = \text{pre}[2] + (1+2+3)\\
&\quad \dots\\
&\text{pre}[i] = (1) + (1+2) + (1+2+3) + \dots + (1+2+3+\dots+i)\\
&\qquad = \text{pre}[i-1] + (1+2+3+\dots+i)\\
&\qquad = \text{pre}[i-1] + \frac{i \times (i+1)}{2}
\end{aligned}$$

由于我们只需要考虑前 2×10^6 个组（即 $i \leqslant 2 \times 10^6$），所以可以直接根据递推式线性求出数组 pre[ ]。

参考代码 2-6-3

```
pre[1] = 1;
for(int i = 2 ; i <= 2000000 ; i ++) pre[i] = pre[i - 1] + i * (i + 1) / 2;
```

当然，pre[i] 也是可以通过推导公式直接计算出来的（请读者自行尝试推导）。

$$\begin{aligned}
\text{pre}[i] &= \sum_{j=1}^{i} \frac{j \times (j+1)}{2}\\
&= \frac{1}{2} \sum_{j=1}^{i} (j^2 + j)\\
&= \frac{1}{2} \times \left[\left(\frac{i \times (i+1) \times (2i+1)}{6} \right) + \frac{i \times (i+1)}{2} \right]\\
&= \frac{i \times (i+1) \times (i+2)}{6}
\end{aligned}$$

参考代码 2-6-4

```
int getPre(int x){ // 求 pre[x] 公式
  return x * (x + 1) * (x + 2) / 6;
}
```

在求出了 pre[] 后，sum[x] 就可以用 $\text{pre}[i-1] + 1 + 2 + \dots + j = \text{pre}[i-1] + \frac{j \times (j+1)}{2}$ 表示。

对于每组查询，其答案等于 $\text{sum}[r] - \text{sum}[l-1]$。所以我们只要分别求出 $l-1$、r 在第几组的第几个位置就可以求出 $\text{sum}[r], \text{sum}[l-1]$。

我们以求 sum[r] 为例来介绍。设 r 在第 i 组的第 j 个位置，那么问题就等价于如何求 i 和 j。

已知在第 i 组之前，有 $i-1$ 个组，这些组共有 $1+2+3+\ldots+i-1=\dfrac{(i-1)\times(i-1+1)}{2}$ 个数。显然，随着 i 增大，$\dfrac{(i-1)\times(i-1+1)}{2}$ 的值就会越来越接近 r，直到大于 r。这说明 i 是满足单调性的。于是我们可以用二分求解 i。而在知道了 i 后，就可以一步算出 j，即 $j=r-\dfrac{(i-1)\times(i-1+1)}{2}$。

$\mathrm{sum}[l-1]$ 的求解同理。

该做法每次查询都需要用二分查找出 r、$l-1$ 的组别和位置，因为共有 T 次查询，所以总的时间复杂度为 $O(T\log_2(2\times10^6)+2\times10^6)$。

参考代码 2-6-5

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int N = 2e6 + 10;
int pre[N];
int getPre(int x){ // 求 pre[x] 的公式
    return x * (x + 1) * (x + 2) / 6;
}
int calc(int x){
    int l = 0 , r = 2e6 , i = 0; // 用二分求 x 在第几组
    while(l <= r){
        int mid = l + r >> 1;
        if((mid + 1) * mid / 2 > x) r = mid - 1 , i = mid;
        else l = mid + 1;
    }
    int j = x - (i - 1) * (i - 1 + 1) / 2; // 求 x 在第几组的第几个位置
    // sum[x] = pre[i - 1] + j * (j + 1) / 2;
    return getPre(i - 1) + j * (j + 1) / 2;
}
signed main(){
    // 递推求 pre[1 ~ 2000000]
    pre[1] = 1;
    for(int i = 2 ; i <= 2000000 ; i ++) pre[i] = pre[i - 1] + i * (i + 1) / 2;
    int T = 1;
    cin >> T;
    while(T --){
        int l , r;
        cin >> l >> r;
        // sum[r] - sum[l - 1]
        cout << calc(r) - calc(l - 1) << '\n';
    }
```

```
32      return 0;
33  }
```

2.7 巩固与练习

题目	难度	题目年份	组别
组队	★	2019 年省赛	• C/C++ B 组第 1 题 • Java B 组第 1 题
递增三元组	★	2018 年省赛	• C/C++ B 组第 6 题 • Java B 组第 6 题
玩具蛇	★	2020 年国赛	• C/C++ A 组第 5 题；C/C++ B 组第 5 题 • Java A 组第 5 题；Java B 组第 5 题 • Python 组第 5 题
分巧克力	★	2017 年省赛	• C/C++ A 组第 9 题； • Java A 组第 9 题；Java B 组第 9 题
全球变暖	★★	2018 年省赛	• C/C++ A 组第 8 题；C/C++ B 组第 9 题 • Java A 组第 8 题；Java B 组第 9 题

第3章 思维与贪心

CHAPTER 3

3.1 重复字符串 (★★)

题目信息

2020 年国赛 - 程序设计题

- ✧ C/C++ C 组第 7 题
- ✧ Python 第 7 题

【问题描述】

如果一个字符串 S 恰好可以由某个字符串重复 K 次得到，我们就称 S 是 K 次重复字符串。例如 *abcabcabc* 可以看作是 *abc* 重复 3 次得到，所以 *abcabcabc* 是 3 次重复字符串。

同理 *aaaaaa* 既是 2 次重复字符串，又是 3 次重复字符串和 6 次重复字符串。

现在给定一个字符串 S，请你计算最少要修改其中几个字符，可以使 S 变为一个 K 次重复字符串?

【输入格式】

第一行包含一个整数 K。

第二行包含一个只含小写字母的字符串 S。

其中，$1 \leqslant K \leqslant 10^5, 1 \leqslant |S| \leqslant 10^5$。其中 $|S|$ 表示 S 的长度。

【输出格式】

输出一个整数代表答案。如果 S 无法修改成 K 次重复字符串，输出 -1。

【样例输入】

2

aabbaa

【样例输出】

2

题目解析

核心考点

✧ 贪心

先来分析，什么情况下 S 无法修改成 K 次重复字符串？

（1）S 的长度小于 K。K 次重复字符串的长度至少为 K。因为我们只能修改字符串，无法添加字符串，所以当 S 的长度小于 K 时，S 一定无法修改成 K 次重复字符串。

（2）S 的长度不是 K 的倍数。若 S 是 K 次重复字符串，则 S 是由某个字符串重复 K 次构成，所以 S 的长度一定是 K 的倍数。同样因为无法添加字符串，所以当 S 的长度不为 K 的倍数时，S 一定无法修改成 K 次重复字符串。

第 2 种情况实际上是包含了第 1 种情况的。因此判断 S 能否修改为 K 次重复字符串，只要判断 S 的长度是否是 K 的倍数即可。

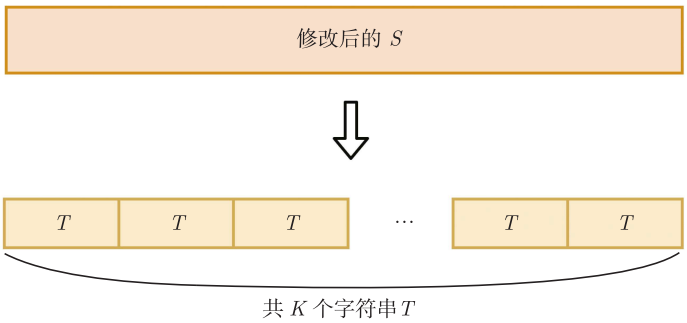
（3）对于其他情况，S 一定可以修改为 K 次重复字符串。

那么要如何修改，才能保证修改的次数最少呢？我们来分析一下。

假设我们以最少的修改次数将字符串 S 修改成了 K 次重复字符串（记为 S'）。根据定义，S' 可由某个字符串重复 K 次得到。

由于 K 是已知的，所以只要我们能求出某个字符串，就可将 S' 还原。还原了 S' 后，就可以将其和 S 一一比对，以求出需要修改的次数。于是问题就可以转换成如何求出某个字符串。

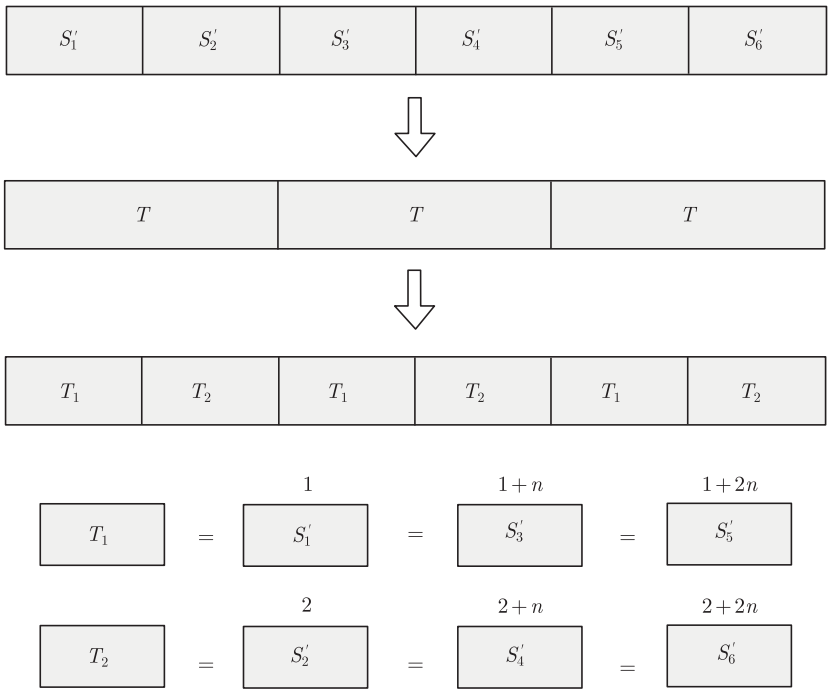
我们设某个字符串为 T，S' 可由 T 重复 K 次得到（$|T| = \dfrac{|S|}{K}$，为了方便后续表示，我们设 $|T|$ 的值为 n），如下图所示。



根据 K 次重复字符串的性质，如果我们将字符串 T 展开为 $T_1, T_2, \ldots, T_n$（T_1 表示 T 的第 1 个字符，T_2 表示 T 的第 2 个字符，…，T_n 表示 T 的第 n 个字符），则满足以下条件。

- $T_1 = S'_1 = S'_{1+n} = S'_{1+2n} = \ldots$
- $T_2 = S'_2 = S'_{2+n} = S'_{2+2n} = \ldots$
- …
- $T_n = S'_n = S'_{2n} = S'_{3n} = \ldots$

例如，S 的长度为 6（$|S| = 6$）、$K = 3$，则 $n = \dfrac{6}{3} = 2$，S' 可由 3 个 T 重复得到，如下页图所示。



观察上图可以发现，如果将 S' 中与 T_1 相同的部分组合在一起、与 T_2 相同的部分组合在一起 …… 与 T_n 相同的部分组合在一起，就会得到一个 $n \times K$ 的矩阵（$n \times K = \dfrac{|S|}{K} \times K = |S|$），如下表所示。

S_1'	S_{1+n}'	S_{1+2n}'	…	$S_{1+(k-1)n}'$
S_2'	S_{2+n}'	S_{2+2n}'	…	$S_{2+(k-1)n}'$
…	…	…	…	…
S_n'	S_{2n}'	S_{3n}'	…	S_{kn}'

该矩阵为字符串 S' 对应的矩阵，它满足每一行所有的元素值都相同。

S' 是我们对 S 进行了最少次数的修改后得到的 K 次重复字符串。从已知条件推断，它与 S 唯一的区别就在于它对应矩阵的每一行的所有元素值都相同。所以，只要用最少的修改次数将 S 对应矩阵的每一行元素都修改为相同的元素，得到的就是 S'，而这最少的修改次数就是我们想要的答案。

由于矩阵的行与行之间是互不关联的，因此最少的修改次数就等价于每行的最少修改次数之和。那么针对矩阵的某一行，如果我们要让所有元素的值都为 x，则需要修改的次数就为矩阵的列数 – 该行中值为 x 的元素个数，即为 $K - x$ 出现的次数。

显然，要使修改的次数最少，就要将该行的所有元素都修改为出现次数最多的元素。于是我们可以得出贪心策略，即将每一行的所有元素都修改为出现次数最多的元素。

由于修改某一行对其他行不会造成任何影响，且每一行的修改次数都是最少的，所以可以保证使用该策略得到的修改次数一定最少，而这最少的修改次数就是我们所要的答案。

总结：我们在刚开始思考的时候，将问题转换为了如何求解某个字符串 T，但最后并没有实际地解决这个问题，而是通过这个问题进一步去分析以得到最终的答案。其实很多时候，我们在面对一个问题时，都不能很快有一条清晰的解题思路。这时，与其马马虎虎猜结论、猜做法，不如脚踏实地，一步步深入思考，发掘解题的核心。

参考代码 3-1　【时间复杂度为 $O(n \times K) = O\left(\frac{|S|}{K} \times K\right) = O(|S|)$】

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
int n , k , ma[N] , cnt[30];
char s[N];
signed main(){
  cin >> k >> (s + 1); // 从 s[1] 开始存储字符串
  // 当 S 不是 K 的倍数时，无论 S 如何修改，也无法成为 K 次重复字符串
  if(strlen(s + 1) % k) return cout << -1 << '\n' , 0;
  int n = strlen(s + 1) / k , ans = 0;
  for(int i = 1 ; i <= n ; i ++) {
     // cnt[j]记录第i行第j个元素出现的次数。初始化cnt数组所有元素的值为0
     for(int j = 0 ; j <= 25 ; j ++) cnt[j] = 0;
     int ma = 0; // ma 统计第 i 行出现最多的元素
     for(int j = 1 ; j <= k ; j ++){
        int x = s[i + (j - 1) * n]; // S 对应矩阵的第 i 行的第 j 列个元素
        cnt[x - 'a'] ++ ; // 该元素出现的次数 + 1
        ma = max(ma , cnt[x - 'a']);// 更新出现次数最多的元素
     }
     ans += k - ma;
  }
  cout << ans << '\n';
  return 0;
}
```

3.2　翻硬币 (★★)

题目信息

2013 年省赛 - 程序设计题

✧ C/C++ A 组第 8 题；C/C++ B 组第 8 题

【问题描述】

小明正在玩一个“翻硬币”的游戏。

桌上放着排成一排的若干硬币。我们用 * 表示正面，用 o 表示反面（是小写字母，不是零）。

比如，可能情形是 **oo* * *oooo;

如果同时翻转左边的两个硬币，则变为 oooo* * *oooo。

现在小明的问题是，如果已知了初始状态和要达到的目标状态，每次只能同时翻转相邻的两个硬币, 那么对特定的局面，最少要翻动多少次呢?

我们约定：把翻动相邻的两个硬币叫作一步操作。

【输入格式】

两行等长的字符串，分别表示初始状态和要达到的目标状态。

每行的长度 < 1000。

【输出格式】

一个整数，表示最小操作步数。

【样例输入】

****

o****o****

【样例输出】

5

懒人速读

> 给定两个硬币序列，硬币的正面用 * 表示，反面用 o 表示。
>
> 小明可以对第一个硬币序列进行操作，每次操作可同时翻转（* → o、o → *）两个相邻的硬币。
>
> 问题是，最少要进行多少次操作，可使得第一个硬币序列变为第二个硬币序列?

题目分析

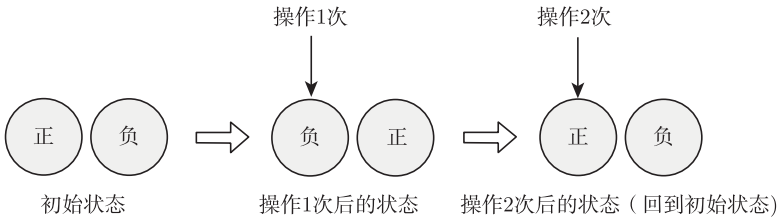
核心考点

- ✧ 贪心
- ✧ 思维分析

本题没有提到当无法将所有硬币的初始状态转换为目标状态时需要输出什么，因此可笃定题目必有解。

依题意，每次操作将同时翻转两枚硬币。为了方便表示，我们可认为**对第 i 枚硬币的操作会同时翻转第 i 枚硬币和第 $i+1$ 枚硬币**。题目要求的是用最少的操作将硬币的初始状态转换为目标状态。那么，如何求这最少的操作次数呢? 我们先从每次操作的意义入手。

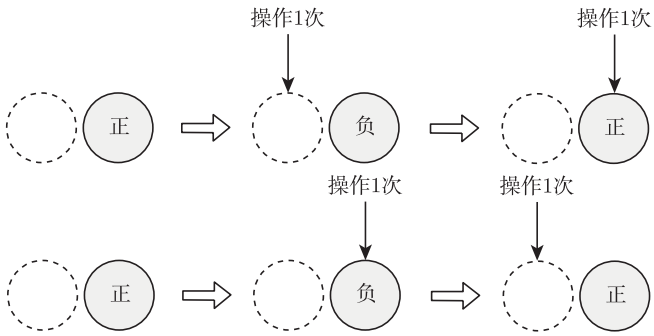
已知对某枚硬币操作一次，它与它相邻的硬币就会翻转一次。那么如果再对该硬币操作一次呢? 显然，这两枚硬币就会回到初始状态，相当于没有操作。从这里可以得出，**对于第 i 枚硬币，我们只会操作 0 次或者 1 次**。



再来思考下，一枚硬币最终的状态会受什么影响?

以第 i 枚硬币为例。只有当我们对第 i 枚硬币或第 $i-1$ 枚硬币操作之后，第 i 枚硬币的状态才会发生改变。这两种操作对第 i 枚硬币的影响实际上是相同的，都是令其翻转。因此，第 i 枚硬币的最终状态不会受两种操作的先后顺序影响，只会受两种操作的操作次数影响。

推广到所有硬币，可得出结论：**操作的先后顺序，对最终结果不会造成影响**，如下图所示。



通过上文的分析，我们得到了以下两条重要的信息。

（1）对于第 i 枚硬币，我们只会操作 0 次或者 1 次。

（2）操作的先后顺序对最终结果不会造成影响。

有了这两条重要的信息，我们便可以开始模拟硬币的翻转。

显然，当第 i 枚硬币的初始状态和目标状态不同时，我们就要将其翻转。将其翻转的方法可能有以下两种。

（1）对第 i 枚硬币进行操作。

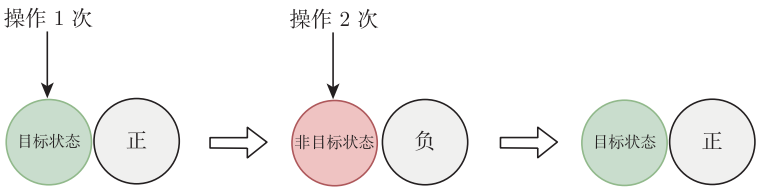
（2）对第 $i-1$ 枚硬币进行操作。

两种方法都能令第 i 枚硬币的状态转换为目标状态，但不同的方法对其他硬币的影响可能是不一样的，我们并不好去预估这两种方法对其他硬币带来的影响。因此，我们最好从第 1 枚硬币开始入手。因为第 1 枚硬币前并没有其他硬币，所以要想改变它的状态，只能对其自身进行操作。

根据信息 1 可得如果第 1 枚硬币的初始状态和目标状态相同，我们就不进行操作；反之，必须进行操作（操作次数为 1），使其状态变为目标状态。

那么，在我们处理完第 1 枚硬币（状态变为目标状态）后，还会对第 1 枚硬币进行操作吗?

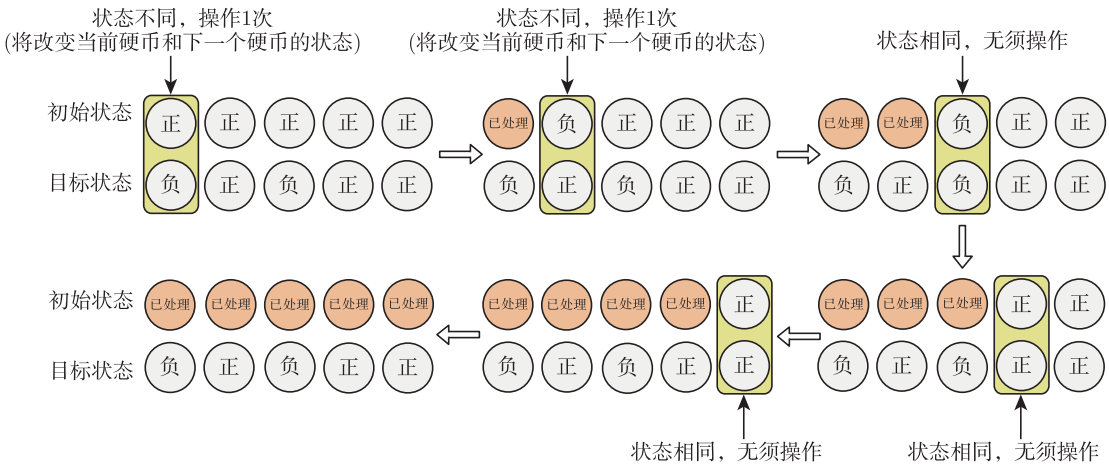
答案是不会的。如果还对第 1 枚硬币进行操作，则其状态将变得与目标状态不同，状态不同就又得再对第 1 枚硬币进行操作，使其状态变为目标状态。这样这两次操作的作用就被互相抵消了，如下图所示。所以当第 1 枚硬币的状态变为目标状态后，我们就不会再对第 1 枚硬币进行操作了。



因为不会再对第 1 枚硬币进行操作，所以我们就只要将注意力集中在剩余硬币上。

那剩余硬币要怎么处理呢? 同样也是从这当中的第 1 枚硬币入手。

例如，有 5 枚硬币，它们的初始状态为 *****，终止状态为 o*o**。如果使用上述方法，则硬币的状态变化将如下图所示。



可以发现，我们每次的操作都会满足以下两个特性。

（1）必要性：只有当被操作的硬币状态和目标状态不同时，才进行操作。

（2）独立性：不会影响已处理好（达到目标状态）的硬币。

显然，满足了必要性和独立性的操作即最优操作。每次都使用最优操作，那么总的操作次数就会是最少的。

参考代码 3-2-1　【时间复杂度为 $O(n)$】

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 signed main()
4 {
```

```
5   int cnt = 0;
6   string s, t;
7   cin >> s >> t; // s 表示硬币的初始状态 ， t 表示硬币的目标状态
8   for(int i = 0 ; i < s.size() - 1; i ++){
9     if(s[i] != t[i]){ // 如果第 i 枚硬币的初始状态和目标状态不同，则对第 i 枚硬币进行操作
10      // 操作会改变第 i 枚硬币、第 i+1 枚硬币的状态
11      // 由于操作之后第 i 枚硬币的状态必然会是目标状态，且之后不会再对第 i 枚硬币进行操作，所
            以可以省去对第 i 硬币的实际修改
12      if(s[i + 1] == '*') s[i + 1] = 'o';
13      else s[i + 1] = '*';
14      cnt ++ ;
15    }
16  }
17  cout << cnt << '\n';
18  return 0;
19 }
```

本题也可以换一种角度思考。

由于我们已笃定题目一定有解，所以根据操作的要求（每次翻转两枚硬币）可知初始状态不为目标状态的硬币个数一定为**偶数**。

当第 i 枚硬币的初始状态和目标状态不同时，就必须对其进行操作，使其状态变为目标状态。由于每次至少要翻转两枚硬币，所以对第 i 枚硬币的操作会影响与它相邻的硬币（第 $i+1$ 枚硬币）的状态，且影响分以下两种情况。

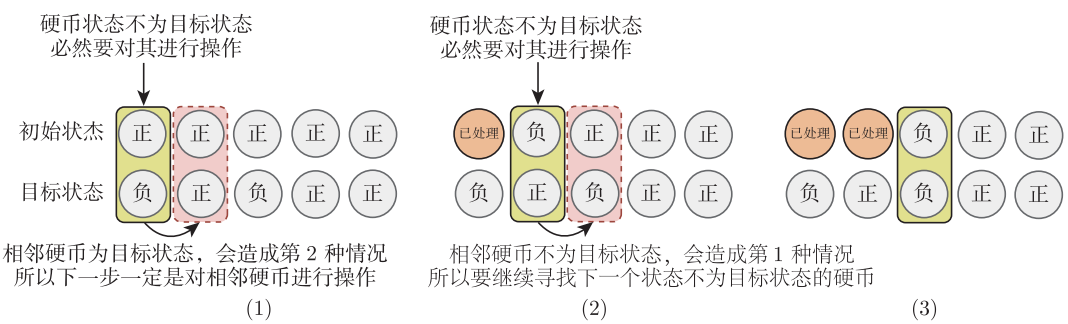
（1）第 $i+1$ 枚硬币的初始状态不为目标状态 $\rightarrow$ 第 $i+1$ 枚硬币的状态变为目标状态。

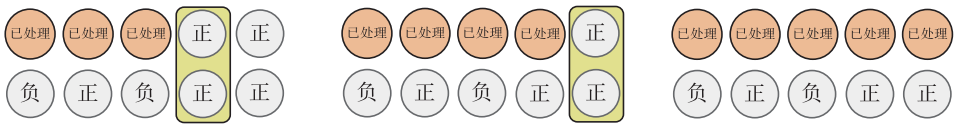
（2）第 $i+1$ 枚硬币的初始状态为目标状态 $\rightarrow$ 第 $i+1$ 枚硬币的状态不再为目标状态。

对于第 1 种情况，操作完成后第 i 枚硬币、第 $i+1$ 枚硬币的状态都达到了目标状态，因此我们无法确定下一枚初始状态不为目标状态的硬币的位置，就要继续寻找，直到找到初始状态不为目标状态的硬币。

对于第 2 种情况，操作完成后第 i 枚硬币的状态会达到目标状态，但第 $i+1$ 枚硬币的状态不会，所以我们可以确定下一次要对第 $i+1$ 枚硬币进行操作。当然如果对第 $i+1$ 枚硬币的操作又造成了第 2 种情况，则需要继续对第 $i+2$ 枚硬币进行操作。

显然，只有在某次操作出现了第 1 种情况后，连续的操作才会停止，如下图所示。





(4)　(5)　(6)

由上可得，我们的操作是由硬币的初始状态不为目标状态开始，也是由硬币的初始状态不为目标状态结束，操作的次数为硬币编号。因此，我们可以**成对**记录初始状态不为目标状态的硬币编号，而每一对硬币编号的差值的和就是答案。

参考代码 3-2-2　【时间复杂度为 $O(n)$】

```cpp
#include<bits/stdc++.h>
using namespace std;
signed main()
{
  int cnt = 0;
  string s, t;
  cin >> s >> t; // S 表示硬币的初始状态 ， T 表示硬币的目标状态
  vector<int>vec; // vec 用来存储初始状态不等于目标状态的硬币的编号
  for(int i = 0 ; i < s.size() - 1; i ++){
    if(s[i] != t[i]) vec.push_back(i);
  }
  for(int i = 0 ; i < vec.size() ; i += 2) { // 成对枚举
    cnt += vec[i + 1] - vec[i]; // 操作次数 = 结束硬币的编号 - 开始硬币的编号
  }
  cout << cnt << '\n';
  return 0;
}
```

3.3　乘积最大 (★★)

题目信息

2018 年国赛 - 程序设计题

✧ C/C++ B 组第 10 题

【问题描述】

给定 N 个整数 $A_1, A_2, \ldots, A_N$。请你从中选出 K 个数，使其乘积最大。

请你求出最大的乘积，由于乘积可能超出整型范围，你只需输出乘积除以 $10^9 + 9$ 的余数。

注意，如果 $X < 0$，我们定义 X 除以 $10^9 + 9$ 的余数是 $-X$ 除以 $10^9 + 9$ 的余数，

即：$0-((0-x)\%10^9+9)$。

【输入格式】

第一行包含两个整数 N, K。

以下 N 行每行一个整数 A_i。

其中，$1 \leqslant K \leqslant N \leqslant 10^5, -10^5 \leqslant A_i \leqslant 10^5$。

【输出格式】

输出一个整数，表示答案。

【样例输入】

5 3

−100000

−10000

2

100000

10000

【样例输出】

999100009

题目解析

核心考点

- ✧ 贪心
- ✧ 动态规划
- ✧ 对数大小比较

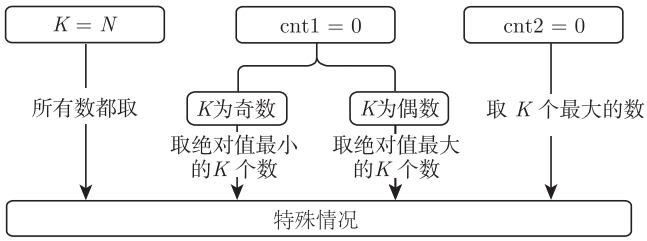
方式 1：满分做法

不妨设序列中有 cnt1 个非负数、cnt2 个负数（cnt1 + cnt2 = N）。

如果 cnt1 = N，那么我们只要把最大的 K 个数取出相乘即可，几乎没有难度。但如果 cnt1 < N，题目的难度就提高了，我们就需要讨论取负数的情况。

根据“负负得正”的性质，为了让乘积的结果最大，我们要尽量成对地取负数。

为什么说是尽量呢? 因为负数的个数可能是奇数。如果是奇数的话，最终的负数可能只剩下 1 个，就无法成对地取负数。所以有了负数的情况是较为复杂的，但存在一些特殊情况，如下页图所示。



（1）$K = N$。所有数都必须选取。

（2）cnt2 = 0。所有数都是非负数，那么取 K 个最大的数可令结果最大。

（3）cnt1 = 0。所有数都是负数，那么分以下两种情况。

- 如果 K 为奇数，则乘积必然为负数，所以取绝对值最小的 K 个数可令结果最大。
- 如果 K 为偶数，则乘积必然为正数，所以取绝对值最大的 K 个数可令结果最大。

对于剩余的情况，序列中一定会包含负数与非负数，且 K 一定小于 N。

- $K < N = \text{cnt1} + \text{cnt2} \rightarrow K \leqslant \text{cnt1} + \text{cnt2} - 1$
- $\text{cnt1} > 0, \text{cnt2} > 0$

我们可以按照 K, cnt1, cnt2 的奇、偶性，将所有情况分解为以下 8 类。

（1）K 为奇数，cnt1 为奇数，cnt2 为奇数。

（2）K 为奇数，cnt1 为奇数，cnt2 为偶数。

（3）K 为奇数，cnt1 为偶数，cnt2 为奇数。

（4）K 为奇数，cnt1 为偶数，cnt2 为偶数。

（5）K 为偶数，cnt1 为奇数，cnt2 为奇数。

（6）K 为偶数，cnt1 为奇数，cnt2 为偶数。

（7）K 为偶数，cnt1 为偶数，cnt2 为奇数。

（8）K 为偶数，cnt1 为偶数，cnt2 为偶数。

为了保证结果尽可能大，我们要尽量取**偶数个负数**和若干个非负数。

因为剩余情况满足 $K \leqslant \text{cnt1} + \text{cnt2} - 1$，所以无论如何，我们都一定可以取偶数个负数、若干个非负数，使得总个数等于 K，即无论属于以上哪一类，都可令乘积的结果大于等于 0。

提示

我们要选取 K 个数，$K \leqslant \text{cnt1} + \text{cnt2} - 1$，$\text{cnt1} + \text{cnt2} = N$，说明 N 个数中必然有一个数不会被选取。

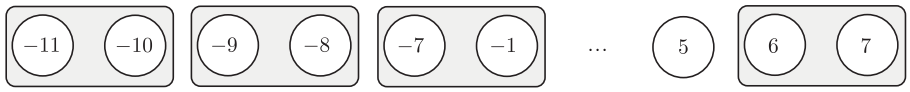
那么，存在以下两种情况。

- 如果 cnt2 是奇数，则我们可以选取一个负数将其从这 N 个数中剔除，使得可以选取的负数个数为 $\text{cnt2} - 1$（偶数）。
- 如果 cnt2 是偶数，则我们暂不进行任何处理。

于是问题可以转换为从偶数个负数、若干个（大于 0）非负数中选取 K 个数。显然，无论 $K(K \leqslant \text{cnt1} + \text{cnt2} - 1)$ 的值是多少，我们都一定可以保证选取的 K 个数中有偶数个负数。

可我们要如何取数才能保证结果在非负数的情况下乘积最大呢?

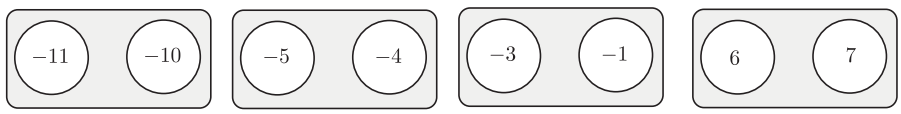
我们可以采用贪心策略，即每次选择符号相同、乘积最大的两个数，直到取完 K 个数，如下图所示。



第 1 次取 $(-11, -10)$
第 2 次取 $(-9, -8)$
第 3 次取 $(6, 7)$
...
每次取两个数，直至取完 K 个数（如果 K 为奇数，最后一次只取一个数）

但这样可能会面临一个问题，即当 K 为奇数时，最后只能取一个数。而在此之前所有的非负数都已经被取完 (只剩下负数)，我们就只能取负数，导致最终的结果变为负数。

按照贪心策略，若 $K = 5$，则取数的步骤如下。

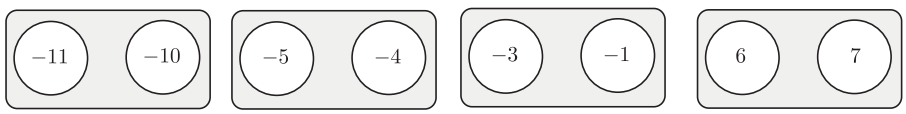


第 1 次：取 $(-11, -10)$，取完后 $K = 3$
第 2 次：取 $(6, 7)$，取完后 $K = 1$
第 3 次：因为 $K = 1$，所以只能从剩下的 $-5, -4, -3, -1$ 中取 1 个数（无论取哪个数，乘积的结果都将为负数）

如何解决这个问题呢？方法很简单，我们只要保证在取最后一个数的时候，**至少**还剩有一个非负数可取。

显然，当 $K \leqslant \text{cnt1}$ 时，就不会产生上面的情况。但是当 $\text{cnt1} < K$ 时，我们**至少**要取 $K - \text{cnt1}$ 个负数。我们可以先成对地选取负数，直到 $K \leqslant \text{cnt1}$ 后，再用上述贪心策略。

假设 cnt1=2, cnt2=6, $K = 5$ 则取数的步骤如下。



第 1 次：因为 cnt1=2<$K = 5$，所以优先取负数对 $(-11, -10)$，取完后 $K = 3$
第 2 次：因为 cnt1=2<$K = 3$，所以优先取负数对 $(-5, -4)$，取完后 $K = 1$
第 3 次：因为 $K = 1$，所以只能从剩下的 -3、-1、6、7 中取一个数。
因为只能取一个数，所以应优先从非负数中选择一个最大的数，即选 7。

参考代码 3-3-1　【时间复杂度为 $O(n \log n)$】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int N = 1e5 + 10 , mod = 1e9 + 9;
int n , k , cnt1 = 0 , cnt2 = 0 , res = 1 , a[N];
```

```
signed main(){
    cin >> n >> k;
    for(int i = 1 ; i <= n ; i ++) {
        cin >> a[i];
        if(a[i] >= 0) cnt1 ++; // 统计非负数的个数
        else cnt2 ++ ; // 统计负数的个数
    }
    sort(a + 1 , a + 1 + n); // 排序
    // 对特殊情况进行处理
    if(n == k){
        for(int i = 1 ; i <= n ; i ++) res *= a[i] , res %= mod;
        cout << res << '\n';
    }
    // 对特殊情况进行处理
    else if(!cnt2) {
        for(int i = n ; i >= n - k + 1 ; i --) res *= a[i] , res %= mod;
        cout << res << '\n';
    }
    // 对特殊情况进行处理
    else if(!cnt1) {
        if(k & 1) for(int i = n ; i >= n - k + 1 ; i --) res = res * a[i] % mod;
        else for(int i = 1 ; i <= k ; i ++) res = res * a[i] % mod;
        cout << res << '\n';
    }
    else{
        // 当 cnt1 < k 时，成对取负数，直到 k <= cnt1
        int p = 1;
        while(k > cnt1) {
            res *= (a[p] * a[p + 1]) % mod , res %= mod;
            p += 2 , k -= 2;
        }
        // 每次可以选择符号相同、乘积最大的两个数，直到取完 K 个数
        int p1 = p , p2 = n;
        while(k > 1) {
            if(a[p1] * a[p1 + 1] >= a[p2] * a[p2 - 1]) {
                res *= (a[p1] * a[p1 + 1]) % mod , res %= mod;
                p1 += 2;
            }
            else {
                res *= (a[p2] * a[p2 - 1]) % mod , res %= mod;
                p2 -= 2;
            }
```

```
48              k -= 2;
49          }
50          // 如果 K 为奇数，即最后只能取一个数，那么要选择非负数才能保证乘积最大
51          if(k) res = res * a[p2] % mod;
52          cout << res << '\n';
53      }
54      return 0;
55  }
```

方式 2：能拿 40% 分数的做法

考虑动态规划。

题目让我们从 N 个数中选择 K 个使得乘积最大。我们可以根据题目的要求，定义 $\mathrm{dp}[i][j]$，其含义为从前 i 个数中选择 j 个的最大乘积。这样，答案可以用 $\mathrm{dp}[N][K]$ 表示。

接着来推导 $\mathrm{dp}[i][j]$ 的转移方程。

我们从第 i 个数入手。第 i 个数有两种状态：选和不选。如果选了，则 $\mathrm{dp}[i][j]$ 的值等价于从前 $i-1$ 个数选 $j-1$ 个数求最大乘积，再乘以第 i 个数；如果不选，则 $\mathrm{dp}[i][j]$ 的含义等价于前 $i-1$ 个数选了 j 个数求最大乘积。

由此便可得出 $\mathrm{dp}[i][j]$ 的转移方程

$$\mathrm{dp}[i][j]=\max(\mathrm{dp}[i-1][j-1]\times a[i],\mathrm{dp}[i-1][j]).$$

不过这个转移方程并不包含所有可能。我们知道一个最大的乘积可能是由两个正数相乘得到，也可能是由两个负数相乘得到。因此，我们还需要维护一个 $f[i][j]$，其含义为从前 i 个数中选择 j 个数求最小乘积。

$\mathrm{dp}[i][j]$ 也可能是由前 $i-1$ 个数选择了 $j-1$ 个数求最小乘积，再乘以第 i 个数得到。所以，$\mathrm{dp}[i][j]$ 的完整转移方程应为

$$\mathrm{dp}[i][j]=\max\begin{cases}\mathrm{dp}[i-1][j]\\ \mathrm{dp}[i-1][j-1]\times a[i]\\ f[i-1][j-1]\times a[i].\end{cases}$$

$f[i][j]$ 的转移方程与 $\mathrm{dp}[i][j]$ 的类似，就不赘述。

转移的时间复杂度为 $O(N^2)$。

提示

因为乘积的结果可能过大导致超出整型范围，所以我们需要用字符串代替整型来存储并模拟计算过程。这会导致状态转移的复杂度有所变化，范围为 $O(N^{1.5})\sim O(N^2)$。于是该方法总的时间复杂度的范围为 $O(N^{3.5})\sim O(N^4)$。

由于常数较小（不容易跑满复杂度），因此还是可顺利拿到 40% 的分数。

此处参考代码略。

方式 3：能拿 60% 分数的做法

在 40% 分数的做法中，我们同时维护了从前 i 个数中选 j 个数的最大乘积、最小乘积（$\mathrm{dp}[i][j]$、$f[i][j]$），还用上了字符串代替整型来模拟计算过程，这是相当麻烦的。那么，如何优化这些烦琐的步骤呢？

（1）优化既要记录最大乘积，又要记录最小乘积。

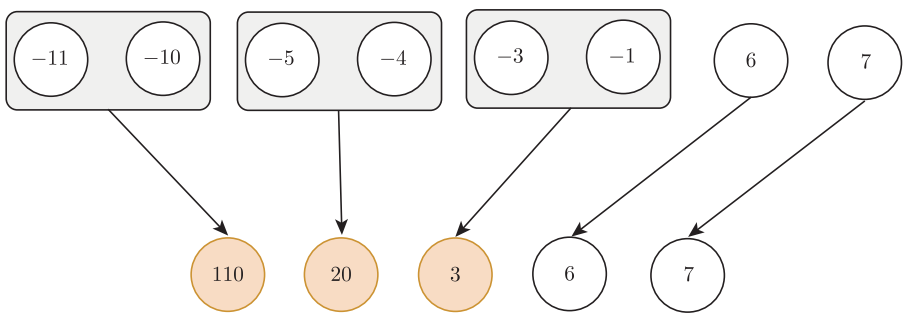
回想一下，我们为什么要记录最小乘积呢？

根据负负得正的性质，最大乘积可能由若干个非负数以及偶数个负数相乘得到（由若干个非负数以及奇数个负数构成的最小乘积再乘上一个负数得到）。

那如果只乘上奇数个负数呢？

根据**满分做法**中的讨论，只有当序列中不存在非负数时，才有可能乘上奇数个负数。对于这种情况，可以特别处理。但对于剩余的情况，最大乘积不会包含奇数个负数。

既然如此，为什么不将负数两两打包在一起，视为一个整体呢？这样，就能保证每次的乘积结果都是非负数，就不需要记录最小乘积，如下图所示。



可是将两个数打包在一起后，要怎么取数呢？

我们换个角度来思考。假设有一个容量为 K 的背包，以及若干个物品。每个物品都有自己的体积和价值，其中一些物品的大小为 2（打包在一起的两个负数），一些物品的大小为 1（非负数），那么如何选择物品可以使得背包能够恰好被装满，且所装物品的总价值最大？

这样，题目就被转换为经典的 01 背包问题。于是 $\mathrm{dp}[i][j]$ 的定义就为装入前 i 个物品，且背包的物品体积总和为 j 可以使价值最大（即乘积最大）。其转移式为 $\mathrm{dp}[i][j]=\max(\mathrm{dp}[i-1][j],\mathrm{dp}[i-1][j-w]\times v)$（$w$, v 分别表示物品的体积和价值）。

最后的答案为 $\mathrm{dp}[n][k]$。

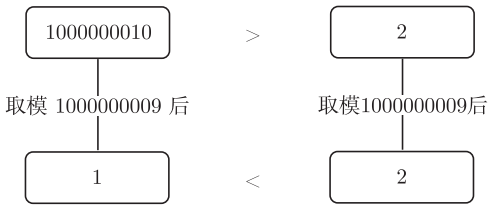
（2）优化使用字符串来代替整型。

回想一下，我们为什么要使用字符串来代替整型进行计算呢？

因为乘积可能超出整型范围，从而导致我们无法进行大小的比较。

但是题目中是让我们通过取模来避免超出整型范围，那为什么不按照题目的提示来呢？

因为题目只要求对最后的乘积结果取模。如果在求解过程中取模，就可能导致大小关系出现变化，如下页图所示。



那有没有什么办法既能使大小关系不出现变化，又能保证乘积结果不超过整型范围呢？有的。这里介绍一个小技巧 —— 用对数（如 $\log_2$）表示正数的大小关系。

- 如果 $X > Y$，则 $\log_2 X > \log_2 Y$。
- 如果 $a \times b > c \times d$，则 $\log_2(a \times b) = \log_2 a + \log_2 b > \log_2(c \times d) = \log_2 c + \log_2 d$。

将乘法替换为对数的加法，可以保证数值不超过整型。

$\mathrm{dp}[i][j] = \max(\mathrm{dp}[i-1][j], \mathrm{dp}[i-1][j-w] + \log_2 v)$（$w$, v 分别表示物品的体积和价值）。

不过这样记录的结果是对数值，并不能作为实际的答案。所以我们还得再定义一个 $\mathrm{ans}[i][j]$，其含义为装入前 i 个物品且物品的体积总和为 j 所能获得的最大价值（即最大乘积）。当 $\mathrm{dp}[i][j]$ 被更新时，同步更新 $\mathrm{ans}[i][j]$ 即可，参考代码如下。

参考代码 3-3-2

```
if(dp[i - 1][j] > dp[i][j]) {
  dp[i][j] = dp[i - 1][j];
  ans[i][j] = ans[i - 1][j];
}

...

if(dp[i - 1][j - w] + log2(v) > dp[i][j]) {
  dp[i][j] = dp[i - 1][j - w] + log2(v);
  ans[i][j] = ans[i - 1][j - w] * v % mod;
}
```

能拿 60% 分数的做法对应的完整代码如下。

参考代码 3-3-3 【时间复杂度为 $O(n^2)$】

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
const int N = 1e3 + 10 , mod = 1e9 + 9;
int a[N] , ans[N][N];
double dp[N][N];
signed main(){
   int n , k , cnt1 = 0 , cnt2 = 0 , res = 1;
   cin >> n >> k;
   for(int i = 1 ; i <= n ; i ++) cin >> a[i];
```

```
    sort(a + 1 , a + 1 + n); // 将数组排序
    for(int i = 1 ; i <= n ; i ++) {
        if(a[i] < 0) cnt2 ++ ;
        else cnt1 ++ ;
    }
    // 对特殊情况进行处理
    if(k == n) {
        for(int i = 1 ; i <= n ; i ++ ) res = res * a[i] % mod;
        return cout << res << '\n' , 0;
    }
    // 对特殊情况进行处理
    if(cnt2 == n){
        if(k & 1) for(int i = n ; i >= n - k + 1 ; i --) res = res * a[i] % mod;
        else for(int i = 1 ; i <= k ; i ++) res = res * a[i] % mod;
        return cout << res << '\n' , 0;
    }
    vector<pair<int , int> >vec;
    vec.push_back(make_pair(0 , 0));
    for(int i = 1 ; i <= n ; i ++) {
        // 将负数两两打包在一起，视为一个物品
        if(a[i] < 0 && a[i + 1] < 0) {
            vec.push_back(make_pair(a[i] * a[i + 1] , 2));
            i ++ ;
        }
        // 将正数单独作为一个物品
        else if(a[i] > 0) vec.push_back(make_pair(a[i] , 1));
    }
    memset(dp , -0x3f , sizeof(dp));
    dp[0][0] = 0 , ans[0][0] = 1; // 初始化
    for(int i = 1 ; i < vec.size() ; i ++) {
        int v = vec[i].first , w = vec[i].second;
        for(int j = 0 ; j <= k ; j ++){
            // 状态转移
            dp[i][j] = dp[i - 1][j];
            ans[i][j] = ans[i - 1][j];
            if(j - w >= 0){
                if(dp[i][j] < dp[i - 1][j - w] + log2(v)){
                    dp[i][j] = dp[i - 1][j - w] + log2(v);
                    ans[i][j] = ans[i - 1][j - w] * v % mod;
                }
            }
        }
```

```
53      }
54      cout << ans[vec.size() - 1][k] << '\n';
55      return 0;
56  }
```

3.4 巧克力 (★★)

题目信息

2021 年国赛 - 程序设计题

✧ C/C++ 研究生组第 8 题；C/C++ C 组第 9 题

【问题描述】

小蓝很喜欢吃巧克力，他每天都要吃一块巧克力。

一天小蓝到超市想买一些巧克力。超市的货架上有很多种巧克力，每种巧克力有自己的价格、数量和剩余的保质期天数，小蓝只吃没过保质期的巧克力，请问小蓝最少花多少钱能买到让自己吃 x 天的巧克力。

【输入格式】

输入的第一行包含两个整数 x, n，分别表示需要吃巧克力的天数和巧克力的种类数。

接下来 n 行描述货架上的巧克力，其中第 i 行包含三个整数 a_i, b_i, c_i，表示第 i 种巧克力的单价为 a_i，保质期还剩 b_i 天（从现在开始的 b_i 天可以吃），数量为 c_i。

【输出格式】

输出一个整数表示小蓝的最小花费。如果不存在让小蓝吃 x 天的购买方案，输出 -1。

【样例输入】

```
10 3
1 6 5
2 7 3
3 10 10
```

【样例输出】

```
18
```

【样例说明】

一种最佳的方案是第 1 种买 5 块，第 2 种买 2 块，第 3 种买 3 块。前 5 天吃第 1 种，第 6, 7 天吃第 2 种，第 8 至 10 天吃第 3 种。

【评测用例规模与约定】

对于 30% 的评测用例，$n, x \leqslant 1000$;

对于所有评测用例，$1 \leqslant n, x \leqslant 100000, 1 \leqslant a_i, b_i, c_i \leqslant 10^9$。

懒人速读

超市的巧克力有 n 种，每种巧克力都有 3 种属性 a,b,c，分别有以下含义。

- a：巧克力的单价。
- b：巧克力的保质期（该巧克力只能在第 1 天到第 b 天之间吃）。
- c：巧克力的数量。

小蓝需要在第 1 天去超市购买一批巧克力，使得接下来的 x 天，每天都至少有一块巧克力吃。请问，小蓝最少需要花费多少钱购买巧克力？

题目分析

核心考点

✧ 贪心
✧ 二分

本题是道经典的贪心题，题目的任务是用最少的花费购买巧克力，使小蓝在第 $1 \sim x$ 天都能吃到没过保质期的巧克力。

首先可以明确只要购买 x 块巧克力就足够了。因为小蓝 1 天吃 1 块巧克力即可（多吃 1 块就要多付 1 份巧克力的钱），共 x 天，所以最多只需要 x 块巧克力。

于是问题就可以转换成要如何选购这 x 块巧克力，才能既保证花费最少，又保证能吃 x 天。

简单分析一下。

对于 1 块巧克力，它有两种属性：价格和保质期。

要使花费的钱尽可能少，就要尽量选用价格低的巧克力。但价格低的巧克力保质期可能也低，保质期低的巧克力就可能出现买了却不能吃的问题。同时受到这两者相互间的牵制，情况是较为复杂的。到底该怎么选呢？下面就让我们来进行深入的探讨。

假设小蓝购买了 1 块巧克力，它的保质期为 b 天，那么，小蓝仅可能在第 $1 \sim b$ 天会吃它（b 天之后该巧克力就过了保质期，小蓝不吃过保质期的巧克力）。

此时如果小蓝已经安排好了第 $1 \sim b$ 天每天要吃的巧克力，则该巧克力就完全没有被吃的必要。像这样的巧克力，我们可以称其为**没用的巧克力**。

如果小蓝并没有安排好第 $1 \sim b$ 天每天要吃的巧克力，那这块巧克力安排在第几天吃最好呢？

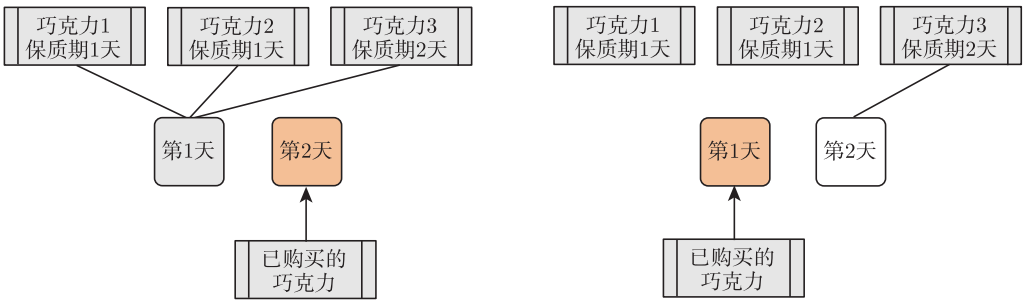
答案是放在第 b 天吃最好。

试想一下，若我们现在要分别为第 A 天、第 B 天挑选巧克力，则第 A 天可以选择的有保质期 $\geqslant A$ 的巧克力，第 B 天可选择的有保质期 $\geqslant B$ 的巧克力。

如果 $A > B$，则第 B 天可选择的巧克力不仅有保质期 $\geqslant A$ 的巧克力，还有保质期在 $B \sim A$ 的巧克力，但第 A 天可选择的巧克力却只有保质期 $\geqslant A$ 的巧克力。

毋庸置疑，可选择的巧克力越多，所花的费用只会越少。

因此，对于保质期为 b 天的巧克力，我们要尽量将它安排在第 b 天吃（物尽其用），这样就可以尽可能把前面的日期空出来，使可供选择的巧克力更多。



当然如果第 b 天已经安排好了要吃的巧克力，我们就尽量将它放在第 $b-1$ 天吃；如果第 $b-1$ 天已经安排好要吃的巧克力，就尽量把它放在第 $b-2$ 天吃 ……

这样，我们就能保证可以尽可能多地购买价格低的**有用的巧克力**。

分析完解题思路后，我们就可以制定具体的解题步骤。

（1）为了保证花费的钱最少，我们需要对 n 种巧克力按单价从小到大排序（若单价相同，则按保质期从大到小排序）。

（2）枚举排好序的巧克力（从单价小的开始枚举），判断该巧克力是否值得购买。

- 若值得（**有用的巧克力**），则需要确定该巧克力放在第几天吃最合适。
- 若不值得（**没用的巧克力**），则继续枚举下一块巧克力。

为了能快速确定**有用的巧克力**安排在哪天吃最合适，我们可以维护一个 set，用来存储那些还没安排巧克力吃的日期。这样我们就可以使用二分法在 set 中快速查找出一个**最大**的日期（找出之后，需要将该日期从 set 中删除），满足该日期没有过巧克力的保质期及该日期没有安排巧克力吃。

- 该日期没有过巧克力的保质期。
- 该日期没有安排巧克力吃。

当 set 为空时，就说明所有日期都已经安排好要吃的巧克力。若枚举完所有巧克力，set 还不为空，则说明不存在让小蓝吃 x 天的购买方案。

参考代码 3-4　【时间复杂度为 $O((n+x)\log n)$】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int N = 1e5 + 10;
set<int>se;
struct node{
    int val , L , cnt; // 单价，保质期，数量
```

```
8     bool operator < (const node & b) const { // 重载运算符
9         if(val == b.val) return L > b.L; // 单价相同，保质期越长越好
10        return val < b.val;        // 单价不同，价格越低越好
11    }
12 }a[N];
13 signed main(){
14    int x , n;
15    cin >> x >> n;
16    for(int i = 1 ; i <= n ; i ++){
17        cin >> a[i].val >> a[i].L >> a[i].cnt;
18    }
19    sort(a + 1 , a + 1 + n); // 将巧克力排序
20    for(int i = 1 ; i <= x ; i ++) se.insert(i); // 刚开始时， 1~x 天都没安排好要吃的巧克
          力，因此我们要将 1~x 天都插入 set 中
21    int p = 1 , res = 0;
22    while(se.size() && p <= n){ // 安排第 p 种巧克力
23        // 第 p 种巧克力一开始有 a[p].cnt 块，一块一块地考虑
24        while(a[p].cnt && se.size() && *se.begin() <= a[p].L){
25            res += a[p].val;
26            a[p].cnt --;
27            // 二分找出第一个大于 a[p].L 的日期
28            // 自减后，得到的就是小于等于 a[p].L 的最大日期
29            auto it = se.upper_bound(a[p].L);
30            it --;
31            se.erase(it); // 安排完巧克力后，要将该日期从 set 中删除
32        }
33        p ++ ;
34    }
35    if(se.size()) cout << -1 << '\n'; // 如果无法安排完所有巧克力，则不存在让小蓝吃 x 天的购
          买方案
36    else cout << res << '\n';
37    return 0;
38 }
```

3.5　答疑 (★★)

题目信息

2021 年国赛 - 程序设计题

✧ C/C++ A 组第 8 题；C/C++ B 组第 8 题；C/C++ C 组第 10 题

✧ Java C 组第 10 题
✧ Python 组第 8 题

【问题描述】

有 n 位同学同时找老师答疑。每位同学都预先估计了自己答疑的时间。

老师可以安排答疑的顺序，同学们要依次进入老师办公室答疑。一位同学答疑的过程如下：

- 首先进入办公室，编号为 i 的同学需要 s_i 毫秒的时间。
- 然后同学问问题老师解答，编号为 i 的同学需要 a_i 毫秒的时间。
- 答疑完成后，同学很高兴，会在课程群里面发一条消息，需要的时间可以忽略。
- 最后同学收拾东西离开办公室，需要 e_i 毫秒的时间。一般需要 10 秒、20 秒或 30 秒，即 e_i 取值为 10000,20000 或 30000。

一位同学离开办公室后，紧接着下一位同学就可以进入办公室了。

答疑从 0 时刻开始。老师想合理地安排答疑的顺序，使得同学们在课程群里面发消息的时刻之和最小。

【输入格式】

输入第一行包含一个整数 n，表示同学的数量。

接下来 n 行，描述每位同学的时间。其中第 i 行包含三个整数 s_i, a_i, e_i，意义如上所述。

$1 \leqslant n \leqslant 1000, 1 \leqslant s_i \leqslant 60000, 1 \leqslant a_i \leqslant 10^6, e_i 10000, 20000, 30000$，即 e_i 一定是 $10000, 20000, 30000$ 之一。

【输出格式】

输出一个整数，表示同学们在课程群里面发消息的时刻之和最小是多少。

【样例输入】

```
3
10000 10000 10000
20000 50000 20000
30000 20000 30000
```

【样例输出】

```
280000
```

懒人速读

> 一名同学进行答疑的过程如下。
> (1) 进入答疑现场: 花费 s 毫秒。
> (2) 参与答疑过程: 花费 a 毫秒（a 毫秒之后该同学会在课程群里发送一条信息)。
> (3) 离开答疑现场: 花费 e 毫秒。

规定只有当上一位同学离开答疑现场后下一位同学才可以进入答疑现场。
现有 n 位同学，每位同学都有各自的 3 个属性 s,a,e。
请安排同学的答疑顺序，使得 n 名同学发送信息的时刻之和（答疑从 0 时刻开始）最小，并且求出该最小时刻之和是多少。

题目分析

核心考点

- ✧ 贪心
- ✧ 逆向思维

在读完这道题后，我们可能会有两个切入点：动态规划和贪心。

不过本题并不适合用动态规划解决，因为无论是状态的定义，还是状态的转移，我们都可以预估出其难度之大。但是，我们可以考虑贪心。

对于一个同学，从他开始答疑到答疑结束，可以简单概括为 3 个阶段：答疑前、答疑时、答疑后。

我们的任务就是结合这 3 个阶段得出一种贪心策略，使得同学们在课程群里发消息的时刻之和最小。

接下来就可以通过对比得出这种贪心策略，对比的因素有 3 个，分别为 3 个阶段所要花费的时间，即答疑前要花的时间、答疑时要花的时间、答疑后要花的时间。

对于两个不同的同学，我们设第 1 个同学答疑前、答疑时、答疑后的时间分别为 s_1,a_1,e_1，第 2 个同学答疑前、答疑时、答疑后的时间分别为 s_2,a_2,e_2。若比较他们这 3 个因素的大小关系，则会有以下 8 种情况。

（1）$s_1 \leqslant s_2, a_1 \leqslant a_2, e_1 \leqslant e_2$。

（2）$s_1 \leqslant s_2, a_1 \leqslant a_2, e_1 > e_2$。

（3）$s_1 \leqslant s_2, a_1 > a_2, e_1 \leqslant e_2$。

（4）$s_1 \leqslant s_2, a_1 > a_2, e_1 > e_2$。

（5）$s_1 > s_2, a_1 \leqslant a_2, e_1 \leqslant e_2$。

（6）$s_1 > s_2, a_1 \leqslant a_2, e_1 > e_2$。

（7）$s_1 > s_2, a_1 > a_2, e_1 \leqslant e_2$。

（8）$s_1 > s_2, a_1 > a_2, e_1 > e_2$。

如果想同时对这 8 种情况进行分析以得出贪心策略无疑十分困难，即先得出贪心策略，再根据贪心策略安排同学们最优的答辩顺序以计算答案并不是一个好的解题思路。但我们可以尝试用逆向思维解题。

假如我们已经以最优的方式对 $1 \sim n$ 每个人答疑的顺序排好了序。

第 1 个人答疑前需要 s_1 毫秒的时间、答疑时需要 a_1 毫秒的时间、答疑后需要 e_1 毫秒的时间；

第 2 个人答疑前需要 s_2 毫秒的时间、答疑时需要 a_2 毫秒的时间、答疑后需要 e_2 毫秒的时间；

第 3 个人答疑前需要 s_3 毫秒的时间、答疑时需要 a_3 毫秒的时间、答疑后需要 e_3 毫秒的时间；

.....

第 n 个人答疑前需要 s_n 毫秒的时间、答疑时需要 a_n 毫秒的时间、答疑后需要 e_n 毫秒的时间。

那么，

第 1 个人发送消息的时刻为 $s_1 + a_1$；

第 2 个人发送消息的时刻为 $s_1 + a_1 + e_1 + s_2 + a_2$；

第 3 个人发送消息的时刻为 $s_1 + a_1 + e_1 + s_2 + a_2 + e_2 + s_3 + a_3$；

.....

第 n 个人发送消息的时刻为 $s_1 + a_1 + e_1 + s_2 + a_2 + e_2 + s_3 + a_3 + e_3 + ... + s_n + a_n$。

这 n 个人发消息的时刻之和（答案）为

$$n \times (s_1 + a_1) + (n-1) \times e_1 + (n-1) \times (s_2 + a_2) + (n-3) \times e_2 + ... + 1 \times (s_n + a_n) + 0 \times e_n$$

从上面的公式中可以发现，第 i 个答疑的人对答案的影响为 $i \times (s_i + a_i) + (i-1) \times e_i$。

显然，对于 e 相同的两个人，谁的 $s + a$ 小或者 $s + a + e$ 小谁就得比对方先答疑（这样才能保证答案尽可能小)。

对于 e 不相同的两个人，我们设第 1 个人答疑前需要 s_1 毫秒的时间、答疑时需要 a_1 毫秒的时间、答疑后需要 e_1 毫秒的时间，第 2 个人答疑前需要 s_2 毫秒的时间、答疑时需要 a_2 毫秒的时间、答疑后需要 e_2 毫秒的时间，那么有以下两种情况。

- 如果第一个人先答疑再到第二个人，则发消息的时刻之和为 $T1 = 2 \times (s_1 + a_1) + e_1 + s_2 + a_2$。
- 如果第二个人先答疑再到第一个人，则发消息的时刻之和为 $T2 = 2 \times (s_2 + a_2) + e_2 + s_1 + a_1$。

倘若第 1 个人先答疑比第 2 个人先答疑耗费的时间少，即

$$T1 \leqslant T2.$$

则有

$$2 \times (s_1 + a_1) + e_1 + s_2 + a_2 \leqslant 2 \times (s_2 + a_2) + e_2 + s_1 + a_1.$$

移项得

$$s_1 + a_1 + e_1 \leqslant s_2 + a_2 + e_2.$$

由此可得：$s + a + e$ 越小的同学越先答疑，答案就越优。

参考代码 3-5　【时间复杂度为 $O(n \log n)$】

```cpp
#include<bits/stdc++.h>
using namespace std;
#define int long long
const int N = 1e3 + 10;
struct node{
  int s , a , e;
  bool operator < (const node & x) const { // 重载运算符
    return (s + a + e) < (x.s + x.a + x.e);
  }
}stu[N];
signed main()
{
  int n , ans = 0;
  cin >> n;
  for(int i = 1 ; i <= n ; i ++) cin >> stu[i].s >> stu[i].a >> stu[i].e;
  sort(stu + 1 , stu + 1 + n); // 排序 :  (s + a + e) 越小的同学安排在越前面答疑
  for(int i = 1 ; i <= n ; i ++){
    int s = stu[i].s , a = stu[i].a , e = stu[i].e;
    ans += (n - i + 1) * (s + a) + (n - i) * e;
  }
  cout << ans << '\n';
  return 0;
}
```

3.6　皮亚诺曲线距离 (★★★★)

题目信息

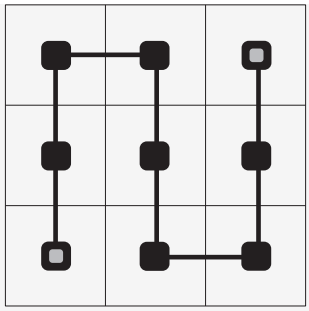
2020 年国赛 - 程序设计题

- ✧ C/C++ A 组第 6 题；C/C++ B 组第 6 题；C/C++ C 组第 8 题
- ✧ Java B 组第 7 题；Java C 组第 7 题

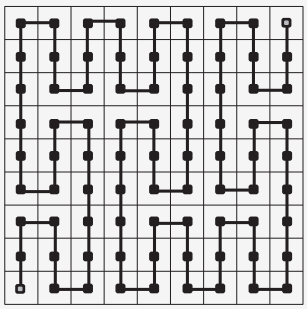
【问题描述】

皮亚诺曲线是一条平面内的曲线。

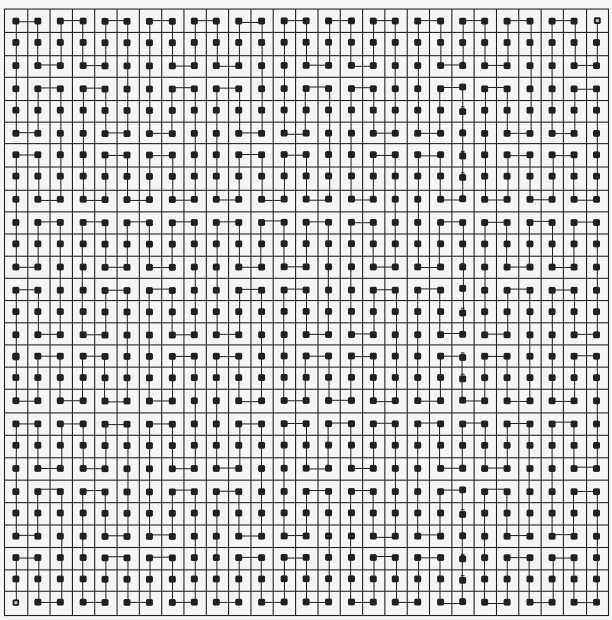
下图给出了皮亚诺曲线的 1 阶情形，它是从左下角出发，经过一个 3×3 的方格中的每一个格子，最终到达右上角的一条曲线。



下图给出了皮亚诺曲线的 2 阶情形，它是经过一个 $3^2 \times 3^2$ 的方格中的每一个格子的一条曲线。它是将 1 阶曲线的每个方格由 1 阶曲线替换而成的。



下图给出了皮亚诺曲线的 3 阶情形，它是经过一个 $3^3 \times 3^3$ 的方格中的每一个格子的一条曲线。它是将 2 阶曲线的每个方格由 1 阶曲线替换而成的。



皮亚诺曲线总是从左下角开始出发，最终到达右上角。

我们将这些格子放到坐标系中，对于 k 阶皮亚诺曲线，左下角的坐标是 $(0,0)$，右上角坐标是 $(3^k-1,3^k-1)$，右下角坐标是 $(3^k-1,0)$，左上角坐标是 $(0,3^k-1)$。

给定 k 阶皮亚诺曲线上的两个点的坐标，请问这两个点之间，如果沿着皮亚诺曲线走，距离是多少?

【输入格式】

输入的第一行包含一个正整数 k，皮亚诺曲线的阶数。

第二行包含两个整数 x_1,y_1，表示第一个点的坐标。

第三行包含两个整数 x_2,y_2，表示第二个点的坐标。

其中有，$0\leqslant k\leqslant 100, 0\leqslant x_1,y_1,x_2,y_2<3^k, x_1,y_1,x_2,y_2\leqslant 10^{18}$。数据保证答案不超过 10^{18}。

【输出格式】

输出一个整数，表示给定的两个点之间的距离。

【样例输入】

```
1
0 0
2 2
```

【样例输出】

```
8
```

题目分析

核心考点

- ✧ 找规律
- ✧ 思维分析

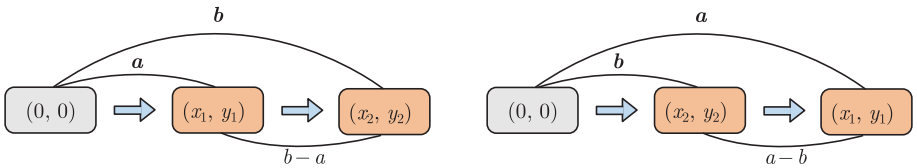
本题是一道“思维”大题，它极其考验选手的观察力、分析力及判断力，难度颇高。

显然，想要根据观察规律直接求出皮亚诺曲线上任意两个点的坐标几乎不可能。因此，我们需要对问题进行优化。

对于皮亚诺曲线上两个不同的点 (x_1,y_1)、(x_2,y_2)，它们的位置关系有以下两种。

- 从起点 $(0,0)$ 出发，先走到 (x_1,y_1) 再走到 (x_2,y_2)，那么从 $(0,0)$ 走到 (x_2,y_2) 就可以分解为 $(0,0)\to(x_1,y_1)\to(x_2,y_2)$。
- 从起点 $(0,0)$ 出发，先走到 (x_2,y_2) 再走到 (x_1,y_1)，那么从 $(0,0)$ 走到 (x_1,y_1) 就可以分解为 $(0,0)\to(x_2,y_2)\to(x_1,y_1)$。

假设 (x_1,y_1)、(x_2,y_2) 到点 $(0,0)$ 的距离分别为 a,b，则它们之间的距离必然为 $|b-a|$，如下图所示。



于是，求解 (x_1, y_1) 到 (x_2, y_2) 的问题就可简化为求点 (x_1, y_1)、(x_2, y_2) 到点 $(0, 0)$ 的距离。

如何求解一个点到起点 $(0, 0)$ 的距离呢？我们接着优化。

根据定义可得，一个 n 阶的皮亚诺曲线一定可以由 9 个 $n-1$ 阶的皮亚诺曲线拼凑构成，方程如下。

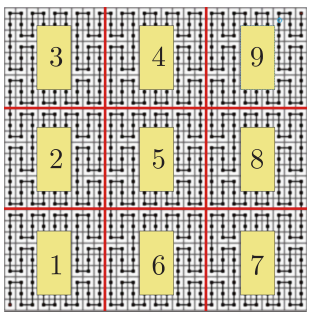
$$3^n \times 3^n = (3^{n-1} \times 3) \times (3^{n-1} \times 3) = 3^{n-1} \times 3^{n-1} \times 9$$

若按照先后顺序将 n 阶的皮亚诺曲线划分为 9 个块（九宫格），则每个块都将会是一个 $n-1$ 阶的皮亚诺曲线。因此，我们可以通过缩放将点映射到块上来简化问题。

对于点 (x, y)，其缩放后坐标会变为 $\left(\left\lfloor \frac{x}{3^{n-1}}, \frac{y}{3^{n-1}} \right\rfloor\right)$，比如在 3 阶皮亚诺曲线中坐标为 $(22, 13)$ 的点缩放后的坐标就为 $\left(\left\lfloor \frac{33}{3^2}, \frac{13}{3^2} \right\rfloor\right)$，即 $(2, 1)$。

3 阶皮亚诺曲线在对点缩放后会有以下几种情况。

- 坐标为 $(0, 0)$ 的点将位于九宫格的第 1 个块上。
- 坐标为 $(0, 1)$ 的点将位于九宫格的第 2 个块上。
- 坐标为 $(0, 2)$ 的点将位于九宫格的第 3 个块上。
- 坐标为 $(1, 2)$ 的点将位于九宫格的第 4 个块上。
- 坐标为 $(1, 1)$ 的点将位于九宫格的第 5 个块上。
- 坐标为 $(1, 0)$ 的点将位于九宫格的第 6 个块上。
- 坐标为 $(2, 0)$ 的点将位于九宫格的第 7 个块上。
- 坐标为 $(2, 1)$ 的点将位于九宫格的第 8 个块上。
- 坐标为 $(2, 2)$ 的点将位于九宫格的第 9 个块上。



对于一个块，我们可以为其添加两个新的概念。

（1）块的出口：表示一个块内距离起点 $(0, 0)$ 最近的点。

（2）块的大小：表示从进入块到离开块所要走的步数（一个 $n-1$ 阶的块的大小等于 $3^{n-1} \times 3^{n-1}$）。

设 (x, y) 位于九宫格的第 i 个块，则它到起点的距离可以分解为以下两部分。

（1）(x, y) 到第 i 个块的出口的距离。

（2）第 i 个块的出口到第 1 个块的出口的距离。

通过观察可以得知第 i 个块的出口到第 1 个块的出口的距离等于 $i-1$ 个块的大小，即 $(i-1) \times 3^{n-1} \times 3^{n-1}$。接下来，我们只要能求出 (x, y) 到第 i 个块的出口的距离，就可求出 (x, y) 到起点的距离。

我们知道，第 i 个块的结构相当于一个 $n-1$ 阶的皮亚诺曲线。根据已有的分析可知，当 $n-1$ 很大时，我们是很难求出 (x, y) 到 $n-1$ 阶皮亚诺曲线的出口的距离的。不过如果 $n-1$ 很小，比如 $n-1=1$ 时，我们就可以通过暴力、模拟，甚至肉眼的观察轻松求出 (x, y) 到 $n-1$ 阶皮亚诺曲线的出口的距离。那么，有没有办法将 $n-1$ 以大化小呢?

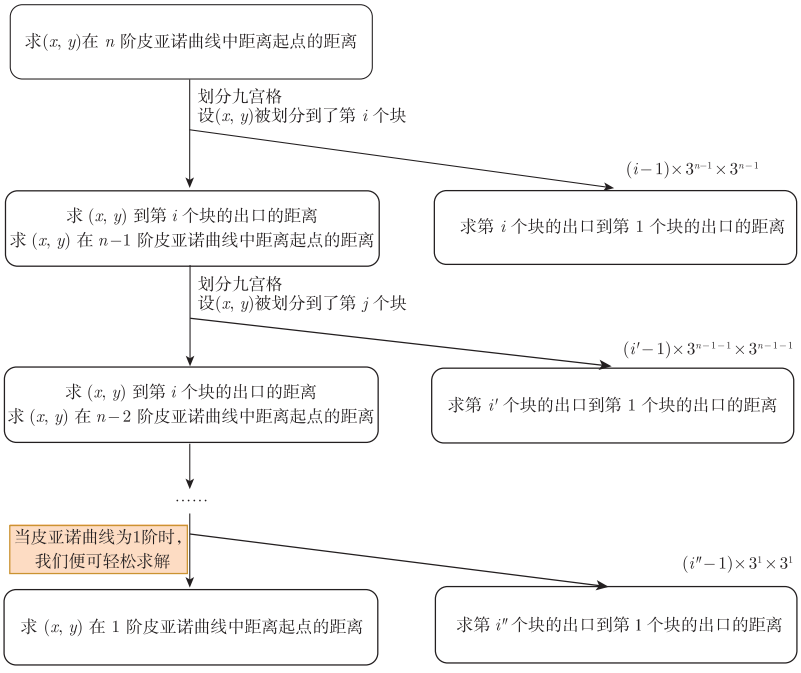
如果我们将第 i 个块看作是一个全新的 $n-1$ 阶的皮亚诺曲线，则求 (x, y) 到第 i 个块的出口的距离就等价于求 (x, y) 到全新的 $n-1$ 阶皮亚诺曲线的出口的距离。

如果我们将这个全新的 $n-1$ 阶皮亚诺曲线划分为新的九宫格，并设 (x, y) 位于新的九宫格中的第 i' 个块，那么此时 (x, y) 距离新的九宫格起点的出口又可以分为两部分。

（1）(x, y) 到新九宫格第 i' 个块的出口的距离。

（2）新九宫格第 i' 个块的出口到新九宫格第 1 个块的距离。

从新九宫格第 i' 个块的出口到新九宫格第 1 个块的出口的距离距离为 $(i'-1) \times 3^{n-2} \times 3^{n-2}$；而求 (x, y) 到第 i' 个块的出口的距离可以重复上述步骤，将其转换为新的问题，如下图所示。

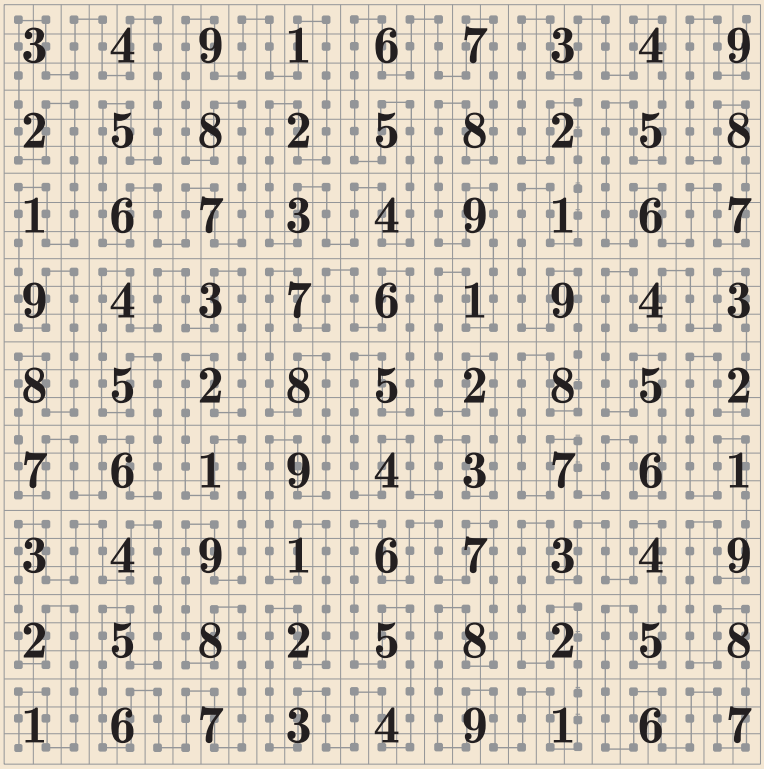


需要注意的是，$n-1$ 阶的九宫格和 n 阶九宫格的块的顺序（结构）不一定相同。

提示

由下图可以发现，n 阶九宫格中存在以下规律。

- 第 $1,3,7,9$ 个块构成的新的九宫格的结构不变。
- 第 $2,8$ 个块构成的新的九宫格的块的编号对应的坐标由 (x,y) 变为 $(2-x,y)$。
- 第 $4,6$ 个块构成的新的九宫格的块的编号对应的坐标由 (x,y) 变为 $(x,2-y)$。
- 第 5 个块构成的新的九宫格的块的编号对应的坐标由 (x,y) 变为 $(2-x,2-y)$。



整个求解通过层层转换的方式将问题逐渐简化为一个与原问题相似，但规模较小的问题来求解，与递归的性质极其吻合，因此我们可以用递归的方式来完成求解。

首先声明一个递归函数 **calc(n , x , y , ??)**，用于计算位于 n 阶皮亚诺曲线中的点 (x,y) 距离第 1 个块的出口的距离，?? 用来表示这个 n 阶皮亚诺曲线的结构（同阶的皮亚诺曲线的结构不一定相同，比如第 1 个块与第 2 个块）。

根据前文，可以分析出以下两种情况。

- 当 $n>1$ 时，$\text{calc}(n,x,y,??)=\text{calc}(n-1,x \bmod 3^{n-1},y \bmod 3^{n-1},??)+(i-1)\times 3^{n-1}\times 3^{n-1}$，其中 $x \bmod 3^{n-1},y \bmod 3^{n-1}$ 是为了方便确定 (x,y) 在 $n-1$ 阶的皮亚诺曲线中位于哪个块。
- 当 $n=1$ 时，$\text{calc}(n,x,y,??)=(x,y)$ 所在块的编号 -1。

那么，要怎么表示 n 阶皮亚诺曲线的结构，又要如何解决结构的变化呢?

我们可以定义一个 3×3 的二维数组 mp[][]，并用 mp[0][0]、mp[0][1]、mp[0][2]、mp[1][0]、mp[1][1]、mp[1][2]、mp[2][0]、mp[2][1]、mp[2][2] 分别表示对应位置的块的编号，再根据上图中 3 阶九宫格每个块变化的规律，对 mp 进行修改即可。

至此，本题的解题分析完成，具体代码见参考代码 3-6。

参考代码 3-6　【时间复杂度为 $O(9\times\log_3 10^{18})$】

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
ll p[40];
map<pair<int, int>, int> mp;
ll calc(int n, ll x, ll y, map<pair<int, int>, int> mp) {
    ll x_ = x / p[n - 1], y_ = y / p[n - 1]; // 将 (x , y) 缩放得到 (x_ , y_)
    if (n == 1) return mp[make_pair(x_, y_)] - 1; // 如果 n = 1, 返回所在块的编号 - 1
    ll ans = (mp[make_pair(x_, y_)] - 1) * p[n - 1] * p[n - 1];
    map<pair<int, int>, int> mp1; // mp1 记录 n-1 阶皮亚诺曲线的变化
    if ((x_ == 0 && y_ == 1) || (x_ == 2 && y_ == 1)) {
        for(int i = 0; i <= 2; i++) for(int j = 0; j <= 2; j++) mp1[make_pair(2 - i, j)] =
            mp[make_pair(i, j)];
    }
    else if ((x_ == 1 && y_ == 2) || (x_ == 1 && y_ == 0)) {
        for(int i = 0; i <= 2; i++) for(int j = 0; j <= 2; j++) mp1[make_pair(i, 2 - j)] =
            mp[make_pair(i, j)];
    }
    else if (x_ == 1 && y_ == 1) {
    for(int i = 0; i <= 2; i++) for(int j = 0; j <= 2; j++) mp1[make_pair(2 - i, 2 - j)]
        = mp[make_pair(i, j)];
    }
    else mp1 = mp;
    return ans += calc(n - 1, x % p[n - 1], y % p[n - 1], mp1);
}
signed main(){
    p[0] = 1;
    for(int i = 1; i < 40; i++) p[i] = p[i - 1] * 3;
    mp[make_pair(0, 0)] = 1; mp[make_pair(0, 1)] = 2; mp[make_pair(0, 2)] = 3;
    mp[make_pair(1, 2)] = 4; mp[make_pair(1, 1)] = 5; mp[make_pair(1, 0)] = 6;
    mp[make_pair(2, 0)] = 7; mp[make_pair(2, 1)] = 8; mp[make_pair(2, 2)] = 9;
    ll k, x1, y1, x2, y2;
    cin >> k >> x1 >> y1 >> x2 >> y2;
    // 由于 3^39 > 10^18, 所以 39 阶皮亚诺曲线已包含了横、纵坐标在 1 ~ 10^18 范围内的所有点
    // 所以从 39 阶皮亚诺曲线开始递归即可
```

```
33      ll a = calc(39, x1, y1, mp);
34      ll b = calc(39, x2, y2, mp);
35      cout << abs(a - b) << '\n';
36      return 0;
37  }
```

3.7 巩固与练习

题目	难度	题目年份	组别
排序	★	2020 年省赛	• Java B 组第 5 题 • Python 组第 5 题
对局匹配	★	2017 年国赛	• C/C++ B 组第 5 题 • Java A 组第 5 题
交换瓶子	★	2019 年省赛	• C/C++ B 组第 9 题 • Java A 组第 9 题
蛇形填数	★★	2020 年省赛	• C/C++ C 组第 3 题；C/C++ B 组第 3 题；C/C++ C 组第 5 题 • Java A 组第 3 题；Java B 组第 3 题 • Python 组第 4 题
付账问题	★★	2018 年省赛	• C/C++ A 组第 10 题 • Java A 组第 10 题

第4章 简单数论

CHAPTER 4

4.1 阶乘约数 (★★)

题目信息

2020 年国赛 - 填空题

- ✧ C/C++ B 组第 3 题；C/C++ C 组第 4 题
- ✧ Java B 组第 3 题；Java C 组第 4 题
- ✧ Python 组第 3 题

【问题描述】

定义阶乘 $n! = 1 \times 2 \times 3 \times \ldots \times n$。

请问 100!（100 的阶乘）有多少个正约数。

【答案提交】

这是一道结果填空题，你只需要算出结果后提交即可。本题的结果作为一个整数，在提交答案时只填写这个整数，填写多余的内容将无法得分。

题目分析

核心考点

- ✧ 数学思维
- ✧ 唯一分解定理
- ✧ 约数定理

题目要我们求的是 100! 的约数个数。约数即因子，如果按照最简单的方法来做题，我们会习惯性地先计算出 100! 的值（设该值为 n），再枚举 $1 \sim \sqrt{n}$ 中的所有数以找出并统计 n 的因子及因子个数，具体代码见参考代码 4-1-1。

参考代码 4-1-1

```
1 定义 n = 1 , ans = 0; // n 用来表示 100! 的值，ans 用来统计因子个数
```

```
for( 从 1 ~ 100 枚举 i ) n = n * i; // 计算 100!
for( 从 1 ~ √n 枚举 i){
  if(n 能整除 i){
    // 若能整除，则 i 和 n / i 都是 n 的因子
    // 如果 i 和 n / i 是两个不同的因子，则答案的个数+2，否则答案的个数+1
    if(n / i 不等于 i) ans += 2;
    else ans += 1;
  }
}
```

这个方法虽然很简单，但阶乘却是“残酷”的，因为仅仅是 20! 的值就已经超过 10^{18}，而 100! 更是个“天文数字”。即使我们能求出 100! 的值，也不可能在有限的时间内枚举出 100! 的所有因子。所以，我们需要转变思路。

上述方法的本质是求出所有因子，以此来统计因子的个数。但我们真的需要求出所有的因子吗?

不需要。正如上文所说，100! 是个“天文数字”，其因子个数可能也十分庞大。在不确定其因子个数的情况下，想求出其所有的因子显然是不合理的。因此，可以使用唯一分解定理和约数定理来解决。

根据唯一分解定理可得，对于一个大于 1 的整数 n，n 可以分解质因数：$\prod_{i=1}^{k} p_i^{a_i} = p_1^{a_1} \cdot p_2^{a_2} \dots p_k^{a_k}$，其中 p_i 表示 n 的第 i 个质因子，a_i 表示 p_i 的幂次，$\prod$ 表示连乘符号。

根据约数定理可得，n 的正约数的个数为 $\prod_{i=1}^{k}(a_i+1) = (a_1+1)(a_2+1)\dots(a_k+1)$。所以我们只要求得 100! 的每个质因子对应的幂次即可求出答案。但怎么求 100! 的每个质因子呢? 我们以 5! 为例简单演示一遍。

对于 5!，其值为 $1\times2\times3\times4\times5 = 120$，其质因子有 $2,3,5$，对应的幂次分别为 $3,1,1$，即 $120 = 2^3 \times 3^1 \times 5^1$。如果我们对 2,3,4,5 分别做质因数分解，则 $2 = 2^1$,$3 = 3^1$,$4 = 2^2$,$5 = 5^1$。不难发现，120 的每个质因子的幂次会等于 2,3,4,5 对应质因子的幂次之和。同理，100! $= 1\times2\times3\times\dots\times100$，所以它的每个质因子的幂次就会等于 $2,3,\dots,100$ 对应质因子的幂次之和。因此我们只要对 $1\sim100$ 的每个数做质因子分解，统计每个质因子的幂次，再对应相加就可得出 100! 的每个质因子的幂次。

求出了每个质因子的幂次后，用约数定理即可求出答案。

需要注意的是，本题答案很大，需要开 long long 来存储答案。

参考代码 4-1-2　【时间复杂度为 $O(100\times\sqrt{100})$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int cnt[N]; // cnt[i] 存的是质因子 i 的幂次
signed main(){
  for(int i = 1 ; i <= 100 ; i ++){
```

```
    int x = i;
    // 质因子分解
    for(int j = 2 ; j * j <= x ; j ++){
      if(x % j == 0) while(x % j == 0) x /= j , cnt[j] ++ ; // j 是其中一个质因子
    }
    if(x > 1) cnt[x] ++ ; // x 是其中一个质因子
  }
  long long ans = 1;
  // 约数定理
  for(int i = 1 ; i <= 100 ; i ++) if(cnt[i] != 0) ans *= (cnt[i] + 1);
  cout << ans << '\n';
  return 0;
}
```

最终答案为 39001250856960000。

4.2　求值 (★★)

题目信息

2019 年国赛 - 填空题

✧ C/C++ B 组第 4 题

【问题描述】

学习了约数后，小明对于约数很好奇，他发现，给定一个正整数 t，总是可以找到含有 t 个约数的整数。小明对于含有 t 个约数的最小数非常感兴趣，并把它定义为 S_t 。

例如 $S_1 = 1, S_2 = 2, S_3 = 4, S_4 = 6, \ldots$ 。

现在小明想知道，当 $t = 100$ 时，S_t 是多少? 即 S_{100} 是多少?

【答案提交】

这是一道结果填空题，你只需要算出结果后提交即可。本题的结果作为一个整数，在提交答案时只填写这个整数，填写多余的内容将无法得分。

题目分析

核心考点

✧ 数学思维　　✧ 唯一分解定理

✧ 约数定理　　✧ DFS

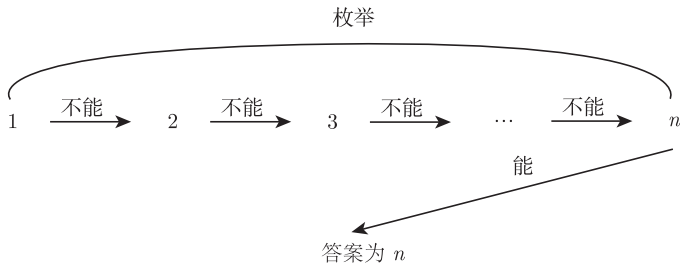
✧ 动态规划

为了追求简单，我们可以先尝试用枚举法来搜寻答案。方法为从小到大枚举整数，枚举的同时判断当前枚举到的数是否能作为答案。

怎么判断一个数是否能作为答案呢？同样为了追求简单，我们可以求出该数的所有因子以统计因子的个数，再判断因子数是否为 100 即可。

判断的结果有两种：能和不能。

由于我们是从小到大枚举的，所以如果判断的结果为能，那么判断的数就一定会是最小的、满足条件的答案（结束枚举）；如果判断的结果为不能，就继续枚举，如下图所示。



显然，这种简单的方法是一定能找到正确答案的，但问题在于该方法找到答案所需要花费的时间是不确定的，可能需要 1 年，也可能连 1 秒都不要。对此，我们可以粗略分析一下。

假设本题的正确答案为 n，那么枚举和判断的总的时间复杂度就为 $O(n\sqrt{n})$。

- 根据唯一分解定理可得，对于一个大于 1 的整数 n，n 可以分解质因数：$\prod_{i=1}^{k} p_i^{a_i} = p_1^{a_1} \cdot p_2^{a_2} \ldots p_k^{a_k}$，其中 p_i 表示 n 的第 i 个质因子，a_i 表示 p_i 的幂次，$\prod$ 表示连乘符号。
- 根据约数定理可得：n 的约数个数为 $\prod_{i=1}^{k}(a_i+1) = (a_1+1)(a_2+1)\ldots(a_k+1)$。

有了这两个定理后，我们就可以任取一个数来确定 n 的上限。假设我们取的数为 x，它的值为 $2^4 \times 3^4 \times 5^3$，由约数定理可得 x 的约数个数为 $(4+1) \times (4+1) \times (3+1) = 100$。

因为 n 的定义是约数个数为 100 的最小整数，即 $n \leqslant x \rightarrow n \leqslant 2^4 \times 3^4 \times 5^3 \rightarrow n \leqslant 162000$，所以 $O(n\sqrt{n})$ 的复杂度可以在很短的时间内得出答案。

因此，该做法可行，对应参考代码 4-2-1 如下所示。

参考代码 4-2-1

```
#include<bits/stdc++.h>
using namespace std;
// 判断 x 的因子个数是否为 100
bool check(int x){
  int cnt = 0;
  for(int i = 1 ; i * i <= x ; i ++) {
    if(x % i == 0) {
      if(x / i == i) cnt += 1;
      else cnt += 2;
    }
```

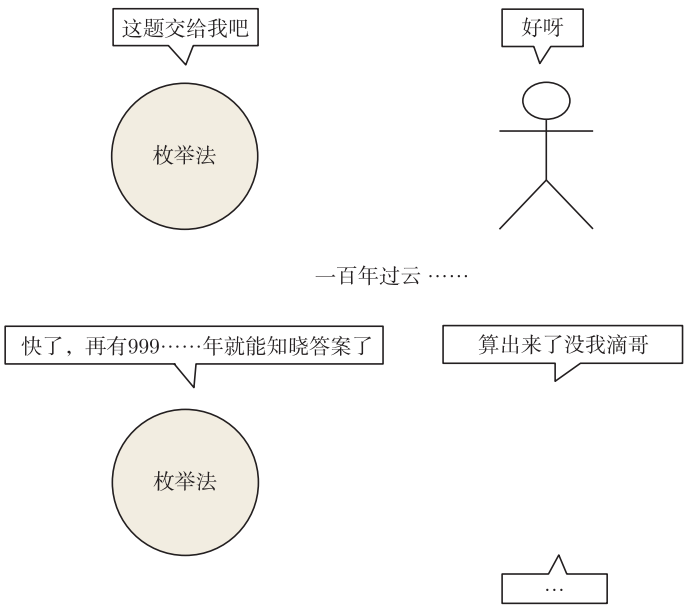
```
  }
  return cnt == 100;
}
signed main(){
  for(int i = 1 ; ; i ++) if(check(i)) {
    cout << i << '\n';
    break ;
  }
  return 0;
}
```

最终答案为 45360。

拓展提高

如果题目要我们求的不是 S_{100}，而是 S_{1000}（保证所求的值小于等于 10^{18}），那怎么办呢？还能用枚举法一一搜寻答案吗？

答案是不可以。我们可以求出 $1 \sim 10^6$ 内每个数的因子个数，会发现其中因子数最多的也只有 240 个，远远不及 1000 个。因此，不难想象 S_{1000} 的答案是很大很大的，以枚举法的效率在有限的时间内是肯定搜寻不出答案的，如下图所示。



既然枚举法搜寻不出答案，那要如何搜寻答案呢？

有一个暴力搜索的办法：通过 DFS 搜索质因子及质因子的幂次来搜寻答案。

以求 S_{1000} 为例，为了方便起见，我们设 $S_{1000} = n$，即设 n 是约数个数为 1000 的最小整数。

那么，根据唯一分解定理，$n=p_1^{a_1}p_2^{a_2}\dots p_k^{a_k}$。根据约数定理，$n$ 的约数个数为 $(a_1+1)(a_2+1)\dots(a_k+1)$。

为了减少答案的搜寻范围，我们先对答案的性质进行分析。

由于 n 的定义是约数个数为 1000 的最小整数，所以为了保证 n 的值最小，须满足 $a_1\geqslant a_2\geqslant\dots\geqslant a_k$。

什么意思?

我们来试想一下，如果数值小的质因子的幂次小于数值大的质因子的幂次，那么，只要我们将两个质因子的幂次交换，n 的值就会变小，而约数个数不会改变。

举例说明

设 $n=2^{19}\times3^{49}$，约数个数为 $(1+19)(1+49)=1000$；如果我们将 2,3 的幂次交换，则 $n=2^{49}\times3^{19}$，约数个数为 $(1+49)(1+19)=1000$，依然是 1000，但 n 的值却小了。

因此可得，数值越小的质因子对应的幂次一定越大，即 $a_1\geqslant a_2\geqslant\dots\geqslant a_k$。

另外根据答案的值小于 10^{18}，我们还可以再获得以下两条信息。

（1）n 的质因子个数最多为 15 。

（2）n 的质因子的最高幂次为 61。

为什么呢? 我们可以采用反证法来证明。

- 对于第 1 条信息，如果 n 的质因子个数大于 15，则必然有

$$\begin{aligned}&n\geqslant\text{16 个最小的质因子相乘}\\&\geqslant\overbrace{2\times3\times5\times7\times11\times13\times17\times19\times23\times29\times31\times37\times41\times43\times47\times53}^{\text{16 个最小的质因子}}\\&\geqslant3258915847719004473 0\\&>10^{18}\end{aligned}$$

不符合题意，所以 n 的质因子个数最多为 15 。

- 对于第 2 条信息，如果 n 的质因子的幂次大于 61，则必然有

$$n\geqslant2^{62}=4611686018427387904>10^{18}$$

不符合题意，所以 n 的质因子的最高幂次为 61。

完成了以上分析后，我们就可以更好地锁定搜索范围，以确定答案。

（1）约数个数为 1000 的最小整数。

（2）大小不超过 10^{18}。

（3）质因子个数不超过 15。

（4）质因子的幂次不超过 61。

（5）数值越小的质因子对应的幂次越大。

参考代码 4-2-2

```
#include<bits/stdc++.h>
using namespace std;
int prime[]={-1 , 2 , 3 , 5 , 7 , 11 , 13 , 17 , 19 , 23 , 29 , 31 , 37 , 41 , 43 , 47};
    // prime[0] 用于占位，prime[1] ~ prime[15] 分别表示第 1~15 个质数
long long ans = 1e18; // ans 表示答案
void dfs(int pos , long long val , int num , int up){ // pos 表示枚举到的质因子，val 表示
    我们搜寻的数，num 表示搜寻的数的质因子个数，up 表示质因子的幂次上限
  if(num > 1000 ) return ; // 约数的个数不超过 1000，故当 sum > 100 时结束搜索
  if(pos > 15) return; // 质因子的个数不超过 15，故当 pos > 15 时结束搜索
  if(num == 1000 && val < ans) {ans = val ; return ;} // 取约数个数为 1000 的最小正数
  for(int i = 1 ; i <= up ; i ++) {
    if(val * prime[pos] > ans) break ; // 剪枝
    val *= prime[pos];
    // 由于质因子越大幂次越小，所以下一个质因子的幂次不能大于 i（当前质因子的幂次）
    dfs(pos + 1 , val , num * (i + 1) , i);
  }
}
signed main(){
  dfs(1 , 1 , 1 , 61);
  cout << ans << '\n';
  return 0;
}
```

此外我们也可以使用动态规划解决，即定义 dp[i][j]，其含义为由前 i 个质数作为因子组成约数个数为 j 的最小整数。

根据约数定理，我们可轻松推出 dp 的状态转移方程。

$$\mathrm{dp}[i][j\times(k+1)]=\min(\mathrm{dp}[i-1][j]\times \mathrm{prime}[i]^k).$$

pirme[i] 表示第 i 个质数，$i\leqslant 15$，$\mathrm{prime}[i]^k\leqslant 10^{18}$。

答案为 dp[15][1000]。

参考代码 4-2-3　【时间复杂度为 $O(15\times 1000\times \log 10^{18})$】

```
#include<bits/stdc++.h>
using namespace std;
int prime[]={-1 , 2 , 3 , 5 , 7 , 11 , 13 , 17 , 19 , 23 , 29 , 31 , 37 , 41 , 43 , 47};
    // prime=[0]用于占位，prime[1]~prime[15]分别表示第 1 ~ 15 个质数
long long dp[16][1001] , INF = 1e18;
signed main(){
   memset(dp , 0x3f , sizeof(dp)); // 初始化为无穷大
   dp[0][1] = 1;
   for(int i = 1 ; i <= 15 ; i ++){
```

```
9         for(int j = 1 ; j <= 1000 ; j ++) {
10            long long p = 1;
11            for(int k = 0 ; k <= 61 ; k ++) { // 幂次的范围为 0~61
12              // 约数个数不能大于 1000 , 即 j * (k + 1) <= 1000
13              // INF = 10^18, 答案的值不超过 10^18, 即 prime[i] * p <= INF
14                if(j * (k + 1) > 1000 || INF / prime[i] < p) break ;
15                if(k > 0) p *= prime[i]; // 第 k 轮循环时, p = prime[i]^k。
16                if(dp[i][j * (k + 1)] / p >= dp[i - 1][j]) dp[i][j * (k + 1)] = p * dp[i -
                      1][j];
17            }
18        }
19    }
20    cout << dp[15][1000] << '\n';
21    return 0;
22 }
```

4.3 循环小数 (★★)

题目信息

2020 年国赛 - 程序设计题

✧ Java 研究生组第 6 题

【问题描述】

已知 S 是一个小于 1 的循环小数，请计算与 S 相等的最简真分数是多少。

例如 $0.3333\ldots$ 等于 $\frac{1}{3}$，$0.1666\ldots$ 等于 $\frac{1}{6}$。

【输入格式】

输入第一行包含两个整数 p 和 q，表示 S 的循环节是小数点后第 p 位到第 q 位。

第二行包含一个 q 位数，代表 S 的小数部分前 q 位。

其中，$1 \leqslant p \leqslant q \leqslant 10$。

【输出格式】

输出两个整数，用一个空格分隔，分别表示答案的分子和分母。

【样例输入】

1 6

142857

【样例输出】

1 7

题目分析

核心考点

- ✧ 数学思维
- ✧ 循环节

在开始做题之前，我们先来看看循环小数的定义：一个数的小数部分从某一位起，有一个或多个数字依次重复出现（循环节）的无限小数叫循环小数。循环小数又分以下两种。

- 纯循环小数：循环节是从小数点后第一位开始循环的小数。
- 混循环小数：循环节不是从小数点后第一位开始循环的小数。

循环小数属于有理数，而任意一个有理数都有分数形式，本题的任务就是将一个循环小数转换为分数形式。

为了方便表示，我们设该循环小数为 n。

根据题目的描述，我们能获取到的信息有以下两点。

（1）n 的小数点后有 q 位数字。

（2）n 的循环节是小数点后第 p 位到第 q 位。

那么，要怎么将 n 转换为分数形式呢？考虑以下两种情况。

（1）n 为纯循环小数。

（2）n 为混循环小数。

1. 情况 1

我们假设 n 的循环节为 a、分数形式为 $\frac{x}{y}$，即 $n = 0.aaaaa\ldots = \frac{x}{y}$（$a$,$x$,$y$ 均为整数）。$\frac{x}{y}$ 即为我们所要求的答案。于是，我们可以从等式 $n = \frac{x}{y}$ 入手。

等号的左边是个循环小数。显然，对于一个循环小数而言，要想将其转换为分数形式是十分困难的。但是，对于一个整数而言，我们就能轻而易举地将其转换为分数。

因此，我们需要尝试将等号的左边转换为整数来处理。

怎么将等号的左边 —— 一个循环小数转换为整数呢？

简单来说，只要将其**小数部分删除（减去小数部分），保留整数部分**即可。

由于 n 小于 1 ，所以 n 的小数部分就是它本身，因此删除 n 的小数部分就相当于令 n 减去它本身，这并没有什么意义。那么，不妨让我们来充分发挥 n 作为循环小数的性质。

假设 n 的循环节 a 是一个 2 位数（即 $\lfloor \lg(a) \rfloor + 1 = 2$，$\lfloor\rfloor$ 表示向下取整），那么 $n \times 10^2 = a.aaaaa\ldots$，整数部分为 a，小数部分为 $0.aaaaa\ldots$。

和 n 一样，$0.aaaaa\ldots$ 是一个以 a 为循环节的小于 1 的循环小数 $(0.aaaaa\ldots = n)$，所以有

$$n \times 100 = a + n.$$

移项得

$$n = \frac{a}{99}.$$

解得

$$x = a, y = 99.$$

提示

该方法的核心在于 n 是一个无限循环小数，所以 $n \times 10^{\lfloor \lg(a) \rfloor + 1}$ 的小数部分不会改变，但整数部分却会变为 a。所以令 $n \times 10^{\lfloor \lg(a) \rfloor + 1}$ 减去 n 后，得到的就是一个可以用 n 表示的整数 $n \times (10^{\lfloor \lg(a) \rfloor + 1} - 1)$，其值为 a。

所以我们可以得出 $n = \dfrac{a}{10^{\lfloor \lg(a) \rfloor + 1} - 1} = 0.aaaaa\ldots$。

2. 情况 2

我们假设 n 的循环节为 a，循环节前一部分的小数为 b，分数形式为 $\dfrac{x}{y}$，即 $n = 0.baaaaa\ldots = \dfrac{x}{y}$（$a$,$b$,$x$,$y$ 均为整数）。

我们可以用情况 1 使用的方法将其进行转换，即 $n = \dfrac{b.aaaaa\ldots}{10^{\lfloor \lg b \rfloor + 1}} = \dfrac{b}{10^{\lfloor \lg(b) \rfloor + 1}} + \dfrac{0.aaaaa\ldots}{10^{\lfloor \lg(b) \rfloor + 1}}$。

由情况 1 得

$$0.aaaaa\ldots = \frac{a}{10^{\lfloor \lg(a) \rfloor + 1} - 1}.$$

所以有

$$n = \frac{b \times (10^{\lfloor \lg(a) \rfloor + 1} - 1) + a}{10^{\lfloor \lg(b) \rfloor + 1} \times (10^{\lfloor \lg(a) \rfloor + 1} - 1)}.$$

得

$$x = b \times (10^{\lfloor \lg(a) \rfloor + 1} - 1) + a, y = (10^{\lfloor \lg(b) \rfloor + 1}) \times (10^{\lfloor \lg(a) \rfloor + 1} - 1).$$

参考代码 4-3

```
#include<bits/stdc++.h>
using namespace std;
signed main(){
  int p , q;
  string s;
  cin >> p >> q >> s;
  if(p == 1) { // 情况1: n 为纯循环小数
    long long a = 0;
    for(int i = 0 ; i < s.size() ; i ++) a = a * 10 + s[i] - '0';
    long long x = a , y = pow(10 , (int)log10(a) + 1) - 1;
    cout << x / __gcd(x , y) << " " << y / __gcd(x , y) << '\n'; // 最简形式
  }
```

```
13    else { // 情况2:  n 为混循环小数
14      long long a = 0 , b = 0;
15      for(int i = 0 ; i < p - 1 ; i ++) b = b * 10 + s[i] - '0';
16      for(int i = p - 1 ; i < q ; i ++) a = a * 10 + s[i] - '0';
17      long long x = b * (pow(10 , (int)log10(a) + 1) - 1) + a , y = pow(10 , (int)log10(b)
            + 1) * (pow(10 , (int)log10(a) + 1) - 1);
18      cout << x / __gcd(x , y) << " " << y / __gcd(x , y) << '\n'; // 最简形式
19    }
20    return 0;
21  }
```

4.4　等差数列 (★★)

题目信息

2019 年省赛 - 程序设计题

- ✧ C/C++ B 组第 8 题；C/C++ C 组第 9 题
- ✧ Java C 组第 9 题

【问题描述】

数学老师给小明出了一道等差数列求和的题目。但是粗心的小明忘记了一部分的数列，只记得其中 N 个整数。

现在给出这 N 个整数，小明想知道包含这 N 个整数的最短的等差数列有几项。

【输入格式】

输入的第一行包含一个整数 N。

第二行包含 N 个整数 $A_1, A_2, \ldots, A_N$。(注意 $A_1 \sim A_N$ 并不一定是按等差数列中的顺序给出)。

其中，$2 \leqslant N \leqslant 10^5, 0 \leqslant A_i \leqslant 10^9$。

【输出格式】

输出一个整数表示答案。

【样例输入】

5

2 6 4 10 20

【样例输出】

10

【样例说明】

包含 2,6,4,10,20 的最短的等差数列是 2,4,6,8,10,12,14,16, 18,20。

题目分析

核心考点

- ✧ 数学思维
- ✧ 等差数列性质
- ✧ gcd(最大公约数)

题意是给定 n 个数 $A_1, A_2, \ldots, A_n$，要求构造出一个长度最短（项数最少的）的等差数列，使得该等差数列包含了这 n 个数，问最短长度（最少项数）为多少。

既然和等差数列相关，那我们就尝试从等差数列的性质入手。

我们知道，一个等差数列必然是有序的，但题目给定的 n 个数不一定是有序的，因此，我们需要将这 n 个数排序。

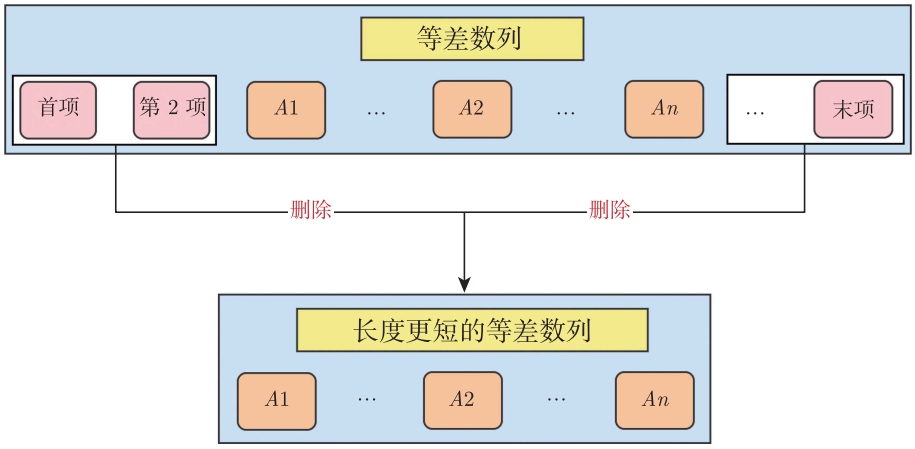
等差数列相邻两项的差值均为一个常数，这个常数我们称之为公差。

公差是等差数列中十分重要的一个概念。假设我们要构造的等差数列的公差为 d，那么对于该等差数列而言，从第二项开始，每一项的值都会等于前一项的值 $+ d$。因此，我们可推断出该等差数列任意一项的值必定可由首项的值加上若干个 d 得到，即要使 $A_1, A_2, \ldots, A_n$ 均被等差数列包含，则须满足这 n 个数中的每一个数都可由首项的值加上若干个 d 得到。

那首项的值取多少合适呢?

取 A_1 最为合适。

因为如果 A_1 不是首项，那么即使去除了 A_1 前面的数，剩余的数所构成的数列依然会是包含 $A_1, A_2, \ldots, A_n$ 的等差数列。因此，要使等差数列的长度最短，取 A_1 为首项最为合适。同理，等差数列末项的值取 A_n 最为合适，如下图所示。



在确定了首项（A_1）和末项（A_n）的取值后，我们就可以根据公式：项数 = [(末项 A_n − 首项 A_1)/ 公差 d] +1 来得到答案。

显然，公差 d 越大，项数就越小。因此，d 越大越好。

不过 d 存在一个限制条件，即 $A_1, A_2, \ldots, A_n$ 中的每一个数都必须可由 A_1 加上若干个数得到。换言之，$A_2, A_3, \ldots, A_n$ 中的每一个数与 A_1 的差值都必须是 d 的倍数。

如果用 $c_2, c_3, \ldots, c_n$ 分别表示 $A_2, A_3, \ldots, A_n$ 与 A_1 的差值，那么我们的任务就是要找出最大的 d，使得 d 是 $c_2, c_3, \ldots, c_n$ 中每一个数的约数。

到这步不难发现，我们要找的 d 其实就是 $c_2, c_3, \ldots, c_n$ 的最大公约数。

于是我们有 $d = \gcd(c_2, c_3, \ldots, c_n)$。

在求出了 d 后，可得答案（项数）$\text{ans} = \dfrac{(A_n - A_1)}{d} + 1$。

参考代码 4-4　【时间复杂度为 $O(n \log n)$】

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
int n , d , a[N];
signed main(){
  cin >> n;
  for(int i = 1 ; i <= n ; i ++) cin >> a[i];
  sort(a + 1 , a + 1 + n);
  for(int i = 1 ; i <= n ; i ++) d = __gcd(d , a[i] - a[1]); // 求差值的最大公约数
  if(!d) cout << n << '\n'; // 如果 d = 0, 说明 a[1] = a[2] = a[3] = ... = a[n]
  else cout << (a[n] - a[1]) / d + 1 << '\n';
  return 0;
}
```

4.5　最大比例 (★★★)

题目信息

2016 年省赛 - 程序设计题

✧ C/C++ A 组第 10 题；C/C++ B 组第 10 题

【问题描述】

X 星球的某个大奖赛设了 M 级奖励。每个级别的奖金是一个正整数。

并且，相邻的两个级别间的比例是个固定值。

也就是说，所有级别的奖金数构成了一个等比数列，比如：16,24,36,54，其等比值为 $\dfrac{3}{2}$。

现在，我们随机调查了一些获奖者的奖金数。

请你据此推算可能的最大的公比值。

【输入格式】

第一行为数字 N $(0 < N < 100)$，表示接下的一行包含 N 个正整数。

第二行 N 个正整数 $X_i(X_i < 10^{12})$，用空格分开。每个整数表示调查到的某人的奖金数额。

【输出格式】

一个形如 A/B 的分数，要求 A, B 互质。表示可能的最大比例系数测试数据保证了输入格式正确，并且最大比例是存在的。

【样例输入 1】

3

1250 200 32

【样例输出 1】

25/4

【样例输入 2】

4

3125 32 32 200

【样例输出 2】

5/2

题目分析

核心考点

- ✧ 数学思维
- ✧ gcd(最大公约数)
- ✧ 等比数列性质
- ✧ 更相减损术

本题的题意可以理解为有一个等比数列，我们并不知道该等比数列的任何信息。于是，我们决定从这个等比数列中随机地取出 n 个数，然后通过这 n 个数来推算出该等比数列可能最大的公比值。

既然涉及了等比数列，那么我们就从等比数列的性质入手。

我们知道，一个等比数列必然是有序的，但题目给定的 n 个数不一定是有序的，因此，我们可以将这 n 个数从小到大进行排序以便后续的操作（假设排好序的 n 个数分别为 $a_1, a_2, \ldots, a_n$）。

对于等比数列而言，它从第 2 项起，每项的值都会等于前一项的值乘以一个常数，这个常数我们称之为公比。

公比是等比数列中十分重要的一个概念。

假设一个等比数列的公比为 q，首项为 x，那么该等比序列的每一项就可以用 $xq^0, xq^1, xq^2, xq^3 \ldots$ 表示。

从这里我们不难发现，等比数列任意两项的比值都会是其公比的倍数。

由于题目给定的 $a_1, a_2, \ldots, a_n$ 是从同一个等比数列中随机取出来的，所以 $\frac{a_2}{a_1}, \frac{a_3}{a_2}, \ldots, \frac{a_n}{a_{n-1}}$ 的值也都会是该等比数列的公比的倍数。

我们设 $\frac{a_2}{a_1}, \frac{a_3}{a_2}, \ldots, \frac{a_n}{a_{n-1}}$ 的值分别为 $q_1, q_2, \ldots, q_{n-1}$，公比值为 Q。

要找出等比数列可能最大的公比值，即找出一个最大的数，使其满足公比的性质；或者找出最大的 Q，使 Q 是 $q_1, q_2, \ldots, q_{n-1}$ 中每一个数的约数；或者求出 $q_1, q_2, \ldots, q_{n-1}$ 的最大公约数，也就是 $Q = \gcd(q_1, q_2, \ldots, q_{n-1})$。

到这里，问题已初步解决，接下来才是本题的难点，即 $q_1, q_2, \ldots, q_{n-1}$ 可能是整数，也可能是分数，而对于分数，我们要怎么求出它们的最大公约数呢（最大公约数也可能是分数）？

求分数的最大公约数

准确来说，$q_1, q_2, \ldots, q_{n-1}$ 不仅仅是 Q 的倍数，还是 Q 的幂次。

如果我们将 Q 用**最简、不可分成幂**的形式 $\left(\frac{u}{d}\right)^k$ 表示，比如 $Q = \frac{9}{4}$ 用 $\left(\frac{3}{2}\right)^2$ 表示，则 $q_1, q_2, \ldots, q_n$ 就可以表示为

$$\left(\frac{u}{d}\right)^{c_1}, \left(\frac{u}{d}\right)^{c_2}, \ldots, \left(\frac{u}{d}\right)^{c_{n-1}}$$

$$Q = \gcd(q_1, q_2, \ldots, q_{n-1}) = \gcd\left(\left(\frac{u}{d}\right)^{c_1}, \left(\frac{u}{d}\right)^{c_2}, \ldots, \left(\frac{u}{d}\right)^{c_{n-1}}\right)$$

$u,d,k,c_1,\ldots,c_{n-1}$ 均为整数，$c_1, c_2, \ldots, c_{n-1}$ 为 k 的倍数。

显然，k 的值越大，Q 的值也就越大。根据 k 为 $c_1, c_2, \ldots, c_{n-1}$ 中每个数的约数可得，最大的 k 即为 $c_1, c_2, \ldots, c_{n-1}$ 的最大公约数，即最大的 $Q = \left(\frac{u}{d}\right)^k = \left(\frac{u}{d}\right)^{\gcd(c_1, c_2, \ldots, c_{n-1})}$。

可问题来了，$u,d,k,c_1,\ldots,c_{n-1}$ 都是我们假设出来的参数，我们并没有 $u,d,k,c_1,\ldots,c_{n-1}$ 的确切值，即我们无法直接通过 $Q = \left(\frac{u}{d}\right)^{\gcd(c_1, c_2, \ldots, c_{n-1})}$ 来得到答案。

那怎么办呢?

有个简单的方法。

我们可以不断地将 $\left(\frac{u}{d}\right)^{c_1}, \left(\frac{u}{d}\right)^{c_2}, \ldots, \left(\frac{u}{d}\right)^{c_{n-1}}$ 中最大的数开方，直至所有的数都相同。此时它们的值就为我们想要的 $\left(\frac{u}{d}\right)^k$。

为什么?

因为 $c_1, c_2, \ldots, c_{n-1}$ 中的每一个数都是 k 的倍数，所以对 $\left(\frac{u}{d}\right)^{c_1}, \left(\frac{u}{d}\right)^{c_2}, \ldots, \left(\frac{u}{d}\right)^{c_{n-1}}$ 分别开若干次方后 $\left(\frac{c_1}{k}, \frac{c_2}{k}, \ldots, \frac{c_{n-1}}{k}\text{ 次方}\right)$ 就必然会得到 $\left(\frac{u}{d}\right)^k$。

我们以 3 个数 $\left(\frac{3}{2}\right)^4, \left(\frac{3}{2}\right)^6, \left(\frac{3}{2}\right)^8$ 为例演绎上述方法。

- 找出其中的最大值，即 $\left(\frac{3}{2}\right)^8$，将其开方得到 $\left(\frac{3}{2}\right)^{\frac{8}{2}}$。由于此时 3 个数的值并没有全都相同，所以继续下一步。
- 找出其中的最大值，即 $\left(\frac{3}{2}\right)^6$，将其开方得到 $\left(\frac{3}{2}\right)^{\frac{6}{2}}$。由于此时 3 个数的值并没有全都相同，所以继续下一步。
- 找出其中的最大值，即 $\left(\frac{3}{2}\right)^4$，将其开方得到 $(\frac{3}{2})^{\frac{4}{2}}$。由于此时 3 个数的值并没有全都相同，所以继续下一步。
- 找出其中的最大值，即 $\left(\frac{3}{2}\right)^{\frac{8}{2}}$，它是由 $\left(\frac{3}{2}\right)^8$ 开平方得到的，于是我们接下来将其改为 $\left(\frac{3}{2}\right)^8$ 的三次方根，即 $\left(\frac{3}{2}\right)^{\frac{8}{3}}$。由于 3 个数的值并没有全都相同，所以继续下一步。
- 找出其中的最大值，即 $\left(\frac{3}{2}\right)^{\frac{6}{2}}$，它是由 $\left(\frac{3}{2}\right)^8$ 开平方得到的，于是我们接下来将其改为 $\left(\frac{3}{2}\right)^6$ 的三次方根，即 $\left(\frac{3}{2}\right)^{\frac{6}{3}}$。由于 3 个数的值并没有全都相同，所以继续下一步。
- 找出其中的最大值，即 $\left(\frac{3}{2}\right)^{\frac{8}{3}}$，它是由 $\left(\frac{3}{2}\right)^8$ 开三次方得到的，于是我们接下来将其改为 $\left(\frac{3}{2}\right)^8$ 的四次方根，即 $\left(\frac{3}{2}\right)^{\frac{8}{4}}$。由于此时 3 个数的值全都相同，所以停止操作，得到答案为 $\left(\frac{3}{2}\right)^2$。

不过该方法虽然思路简单，但用程序来实现并不轻松。

那还有别的方法可以求解吗?

有的。我们可以利用更相减损术。

根据上文分析，我们得到了 Q 的两个等式。

（1）$Q=\gcd\left(\left(\frac{u}{d}\right)^{c_1},\left(\frac{u}{d}\right)^{c_2},\ldots,\left(\frac{u}{d}\right)^{c_{n-1}}\right)$。

（2）$Q=\left(\frac{u}{d}\right)^{\gcd(c_1,c_2,\ldots,c_{n-1})}$。

将它们结合可得 $\gcd\left(\left(\frac{u}{d}\right)^{c_1},\left(\frac{u}{d}\right)^{c_2},\ldots,\left(\frac{u}{d}\right)^{c_{n-1}}\right)=\left(\frac{u}{d}\right)^{\gcd(c_1,c_2,\ldots,c_{n-1})}$。

为了方便表示，我们设 $\frac{u}{d}=p$，那么 $\gcd(p^{c_1},p^{c_2},\ldots,p^{c_{n-1}})=p^{\gcd(c_1,c_2,\ldots,c_{n-1})}$。

由于求解多个数的最大公约数可以分解为多次求解两个数的最大公约数，比如 $\gcd(a,b,c)=\gcd(a,\gcd(b,c))$，所以我们只需要分析如何求解两个分数的最大公约数即可。

下面我们以求解 p^{c_1},p^{c_2} 的最大公约数为例来介绍。

（1）已知

$$\gcd(p^{c_1},p^{c_2})=p^{\gcd(c_1,c_2)}.$$

（2）根据更相减损术 $\gcd(a,b)=\gcd(a,b-a)$ 可得

$$p^{\gcd(c_1,c_2)}=p^{\gcd(c_1,c_2-c_1)}=\gcd(p^{c_1},p^{c_2-c_1})=\gcd\left(p^{c_1},\frac{p^{c_2}}{p^{c_1}}\right)=\dots.$$

（3）让这个式子不断变化下去，直到 $p^{c_1}=p^{c_2}\left(\dfrac{p^{c_1}}{p^{c_2}}=1\right)$，那么

$$\gcd\left(p^{c_1},\frac{p^{c_2}}{p^{c_1}}\right)=\gcd(p^{c_1},1)=\gcd(p^{c_1},p^0)=p^{\gcd(c_1,0)}=p^{c_1}.$$

这样，我们就能成功求解出两个分数的最大公约数。接下来只要多次运用此方法，就可轻松得出答案。

提示

（1）$a_1,a_2,\dots,a_n$ 是从一个等比数列中随机抽取的，一个数可能被抽取多次，所以我们需要去除重复出现的数。

（2）在用更相减损术求解时，我们可以把 $\dfrac{u}{d}$ 的分子、分母分开计算，这并不会影响答案。因为答案为 $\left(\dfrac{u}{d}\right)^k$，它的分子和分母的幂次是相同的。

参考代码 4-5

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int n , cnt;
long long a[N] , u[N] , d[N];
long long gcd_sub(long long a , long long b){
  if(a < b) swap(a , b); // 更相减损术总是大减小（a,b 的底数是一样的）
  if(b == 1) return a;
  return gcd_sub(b , a / b);
}
signed main(){
  cin >> n;
  for(int i = 1 ; i <= n ; i ++) cin >> a[i];
  sort(a + 1 , a + 1 + n);
  for(int i = 2 ; i <= n ; i ++){
    if(a[i] == a[i - 1]) continue ; // 去重
    long long g = __gcd(a[i] , a[i - 1]);
    // 将 a[i] / a[i - 1] 化为最简分数后分别存储在 u[cnt] 和 d[cnt] 里
    u[++ cnt] = a[i] / g;
    d[cnt] = a[i - 1] / g;
  }
  long long x = u[1] , y = d[1];
```

```
23    for(int i = 2 ; i <= cnt ; i ++){
24      // 分子、分母分开处理
25      x = gcd_sub(x , u[i]);
26      y = gcd_sub(y , d[i]);
27    }
28    cout << x << "/" << y << '\n';
29    return 0;
30  }
```

4.6 巩固与练习

题目	难度	题目年份	组别
分数	★	2018 年省赛	• C/C++ A 组第 1 题 • Java A 组第 1 题
序列求和	★	2019 年国赛	• C/C++ A 组第 4 题 • Java B 组第 5 题；Java C 组第 5 题
乘积尾零	★	2018 年省赛	• C/C++ A 组第 3 题；C/C++ B 组第 3 题
包子凑数	★★	2017 年省赛	• C/C++ A 组第 8 题；C/C++ B 组第 8 题 • Java A 组第 8 题；Java B 组第 8 题
选素数	★★★	2022 年国赛	• C/C++ A 组第 7 题 • Java A 组第 8 题

第5章 字符串算法

CHAPTER 5

5.1 单词分析 (★)

题目信息

2020 年省赛 - 程序设计题

- ✧ C/C++ C 组第 7 题
- ✧ Java B 组第 7 题；Java C 组第 7 题
- ✧ Python 组第 7 题

【问题描述】

小蓝正在学习一门神奇的语言，这门语言中的单词都是由小写英文字母组成，有些单词很长，远远超过正常英文单词的长度。小蓝学了很长时间也记不住一些单词，他准备不再完全记忆这些单词，而是根据单词中哪个字母出现得最多来分辨单词。

现在，请你帮助小蓝，给了一个单词后，帮助他找到出现最多的字母和这个字母出现的次数。

【输入格式】

输入一行包含一个单词，单词只由小写英文字母组成。

对于所有的评测用例，输入的单词长度不超过 1000。

【输出格式】

输出两行，第一行包含一个英文字母，表示单词中出现得最多的字母是哪个。

如果有多个字母出现的次数相等，输出字典序最小的那个。

第二行包含一个整数，表示出现得最多的那个字母在单词中出现的次数。

【样例输入 1】

lanqiao

【样例输出 1】

a

2

【样例输入 2】

longlonglongistoolong

【样例输出 2】

o

6

懒人速读

> 给定一个只包含小写字母的字符串，要找出该字符串中出现次数最多的、字典序最小的字母，输出该字母及该字母出现的次数。

题目分析

核心考点

- ✧ 模拟
- ✧ ASCII

本题思路十分简单，只要根据题意，先将单词中的每个字母（只包含小写字母）出现的次数记录下来，然后再找出出现次数最多、字典序最小的字母即可。

该思路可分解为以下两个步骤。

（1）统计每个字母出现的次数。

（2）找出出现次数最多、字典序最小的字母。

显然，要想实现这两个步骤是有很多种方法的，比如我们可以采用最基础的方法 —— 定义 26 个变量，每个变量统计一个字母出现的次数，方法如下。

参考代码 5-1-1

```
定义变量:  cnt_a, cnt_b, cnt_c ... 分别对应字母 a,b,c ... 的出现次数
for(遍历单词){
  if (当前遍历到的字母为 a) cnt_a ++ ;
  if (当前遍历到的字母为 b) cnt_b ++ ;
  if (当前遍历到的字母为 c) cnt_c ++ ;
  ...
}
```

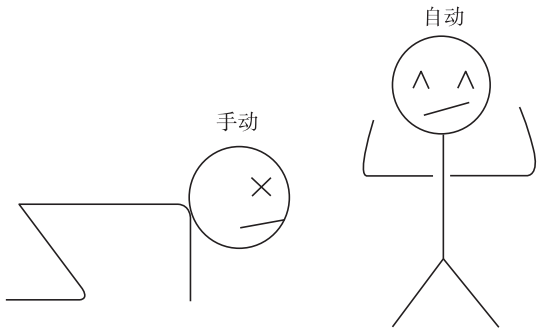
待遍历完单词（所有字母）后，再对定义的 26 个变量一一进行比较，以找出出现次数最多、字典序最小的字母。

不过该方法的代码量并不小。造成代码量不小的原因很明显 —— 有多条判断语句。

为什么要有这么多条判断语句呢?

因为每个字母都有统计自己出现次数的变量。我们要**手动**根据遍历到的字母的不同增加不同变量的值（将遍历到的字母对应的变量值 +1）。

若能让程序**自动**将字母和统计该字母出现次数的变量对应上，我们就可以省去不少代码。



那有没有办法能让程序自动将字母和统计该字母出现次数的变量对应上呢?

有的。我们可以用数组来解决，即开辟一个空间足够大的数组，一个字母对应一个数组下标，将不同字母出现的次数记录在不同下标对应的元素中。

这样，就可以将字母对应变量转换为字母对应数组下标（一个整数）。

那如何让程序自动将字母和该字母对应的数组下标对应上呢?

我们可以利用 **ASCII** 编码来实现。

ASCII 编码

在计算机中，所有的数据在存储和运算时都要使用二进制数表示，而具体用哪些二进制数字表示哪个符号，每个人都可以自行决定。为了避免混乱，有关标准化组织就出台了 **ASCII** 编码。**ASCII** 编码定义了二进制数与字符的对应规则。

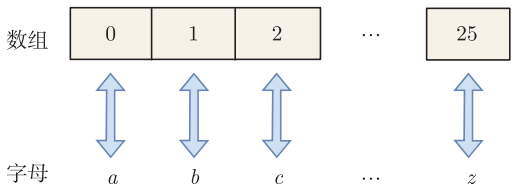
对于字符 $a \sim z$，它们的 **ASCII** 编码是**连续**的，对应的十进制数分别为 $97 \sim 122$，即 (char)'a' = (int)97。所以我们不用怎么处理就可以将 $a \sim z$ 与数组下标 $97 \sim 122$ 分别对应上。这样就减少了不少代码量，方法如下。

参考代码 5-1-2

```
定义数组： cnt[]
for(遍历单词){
  cnt[当前遍历到的字母] ++;
}
```

不过这样存在一个资源浪费的问题。因为 $a \sim z$ 对应的数组下标为 $97 \sim 122$，所以我们至少要开辟一个空间大于 122 的数组。但我们实际上使用的只有数组中下标为 $97 \sim 122$ 的元素，所以下标为 $0 \sim 96$ 的空间就浪费了。

对此，我们可以将 $a \sim z$ 对应的数组下标全都减去 97，即 $a \sim z$ 对应数 $0 \sim 25$，如下图所示。这样我们只要开辟一个空间为 26 的数组即可，方法如下。



参考代码 5-1-3

```
定义数组： cnt[26]

for(遍历单词){
  cnt[当前遍历到的字母 - 97] ++;
}
```

待统计完每个单词出现的次数后，遍历数组的第 1 至第 26 个元素，找出其中对应元素值最大、字典序最小的数组下标，然后根据字母 = 下标 + 97 即可求出答案。

参考代码 5-1-4 【时间复杂度为 $O(|S|)$（$|S|$ 表示 S 的长度）】

```cpp
#include<bits/stdc++.h>
using namespace std;
int cnt[26];
signed main(){
  string s;
  cin >> s;
  for(int i = 0 ; i < s.size() ; i ++) cnt[s[i] - 'a'] ++ ; // 'a' = 97
  char ans1 = 0 ; int ans2 = 0;
  for(int i = 0 ; i < 26 ; i ++){
    if(cnt[i] > ans2) ans2 = cnt[i] , ans1 = (i + 'a'); // 'a' = 97;
  }
  cout << ans1 << '\n' << ans2 << '\n';
  return 0;
}
```

5.2 人物相关性分析 (★★)

题目信息

2019 年省赛 - 程序设计题

- ✧ C/C++ C 组第 8 题
- ✧ Java B 组第 8 题；Java C 组第 8 题

【问题描述】

小明正在分析一本小说中的人物相关性。他想知道在小说中 Alice 和 Bob 有多少次同时出现。

更准确地说，小明定义 Alice 和 Bob “同时出现”的意思是在小说文本中 Alice 和 Bob 之间不超过 K 个字符。

例如以下文本：

This is a story about Alice and Bob. Alice wants to send a private message to Bob.

假设 $K = 20$，则 Alice 和 Bob 同时出现了 2 次，分别是“Alice and Bob”和“Bob. Alice”。前者 Alice 和 Bob 之间有 5 个字符，后者有 2 个字符。

注意:

- Alice 和 Bob 是大小写敏感的，alice 或 bob 等并不计算在内。
- Alice 和 Bob 应为单独的单词，前后可以有标点符号和空格，但是不能有字母。例如 Bobbi 并不算出现了 Bob。

【输入格式】

第一行包含一个整数 $K(1 \leqslant K \leqslant 10^6)$。

第二行包含一行字符串，只包含大小写字母、标点符号和空格。长度不超过 10^6。

【输出格式】

输出一个整数，表示 Alice 和 Bob 同时出现的次数。

【样例输入】

20

This is a story about Alice and Bob. Alice wants to send a private message to Bob.

【样例输出】

2

题目分析

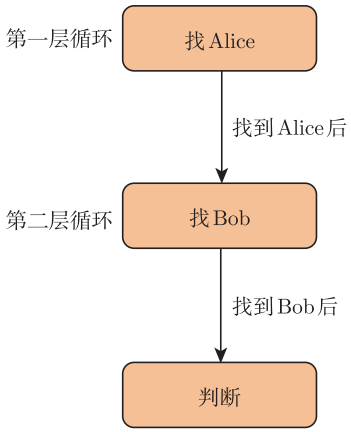
核心考点

- ✧ 枚举
- ✧ 计算贡献
- ✧ 二分
- ✧ 前缀和
- ✧ 尺取法

本题的题意清晰，只要按照要求找出所有的 Alice, Bob，然后求出这当中多少对 (Alice,Bob) 同时出现即可。

由于题目的求解目标十分明确，所以我们可以先使用一种绝大部分人都会想到的简单方法 —— 双层循环 + 判断来尝试对问题的求解。

双层循环 + 判断的原理如下页图所示。



- 第一层循环用来找 Alice，每找到一个 Alice，就进入第二层循环。
- 第二层循环用来找 Bob，每找到一个 Bob 就进行判断。
- 判断第一层循环找到的 Alice 和第二层循环找到的 Bob 是否同时出现。根据判断的结果对答案进行相应的更新

该方法的实现主要分为以下两步。

（1）循环找出 Alice, Bob。

（2）判断 Alice 和 Bob 是否同时出现。

循环大家都会写，但如何找出 Alice 和 Bob 呢? 找出后又要如何判断 Alice 和 Bob 是否同时出现呢?

1. 如何找出 Alice 和 Bob

提出这个问题主要是因为题干的两条重要信息。

（1）Alice 和 Bob 是大小写敏感的，alice 或 bob 等并不计算在内。

（2）Alice 和 Bob 应为单独的单词，前后可以有标点符号和空格，但是不能有字母。例如出现 Bobbi 并不算出现了 Bob。

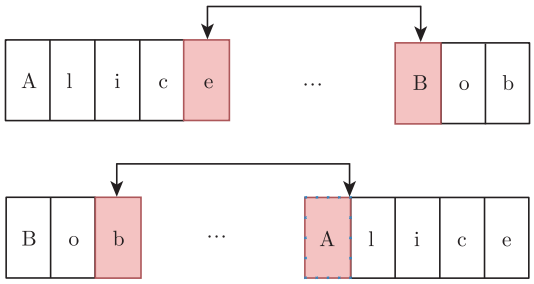
处理方法其实也很简单。

（1）对于 Alice，每当文本中有连续的 5 个字符分别为 A, l, i, c, e，且 A 没有上一个字符或 A 的上一个字符不为大小写字母、e 没有下一个字符或 e 的下一个字符不为大小写字母时，我们就当找到了 Alice。

（2）对于 Bob，每当文本中有连续的 3 个字符分别为 B, o, b，且 B 没有上一个字符或 B 的上一个字符不为大小写字母、b 没有下一个字符或 b 的下一个字符不为大小写字母时，我们就当找到了 Bob。

2. 如何判断 Alice 和 Bob 是否同时出现

当 Alice 与 Bob 同时出现时，它们的位置关系会有两种情况，如下页图所示。



（1）Alice 出现在 Bob 之前（此时 Alice 和 Bob 之间的字符个数相当于 Alice 中的 e 和 Bob 中的 B 之间的字符个数）。

（2）Alice 出现在 Bob 之后（此时 Alice 和 Bob 之间的字符个数相当于 Alice 中的 A 和 Bob 中的 b 之间的字符个数）。

针对这两种情况我们进行不同处理即可。

参考代码 5-2-1　【时间复杂度为 $O(n^2)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
bool check(char x){
    if(x >= 'a' && x <= 'z') return true;
    if(x >= 'A' && x <= 'Z') return true;
    if(x >= '0' && x <= '9') return true;
    return false;
}
vector<int>B , b;
signed main(){
    int k ; string s;
    cin >> k;
    cin.get();
    getline(cin , s); // 整行读入字符串
    int n = s.size();
    B.push_back(-1000000000) , b.push_back(-1000000000); // 边界处理
    for(int i = 0 ; i < n ; i ++){
        if(i + 2 < n && s.substr(i , 3) == "Bob"){
            if(i - 1 >= 0 && check(s[i - 1]) || i + 3 < n && check(s[i + 3])) continue ;
            B.push_back(i + 1) , b.push_back(i + 3);
        }
    }
    B.push_back(1000000000) , b.push_back(1000000000); // 边界处理
    long long ans = 0;
    for(int i = 0 ; i < n ; i ++){
        if(i + 4 < n && s.substr(i , 5) == "Alice"){
```

```
28          if(i - 1 >= 0 && check(s[i - 1]) || i + 5 < n && check(s[i + 5])) continue ;
29          int A = i + 1 , e = i + 5;
30          // Alice 在 Bob 前面
31          int p1 = upper_bound(B.begin() , B.end() , e + k) - B.begin() - 1; // 找到比e+k
                大的最小数的下标，那么该下标-1得到的就是小于等于e+k的最大数的下标
32          int p2 = lower_bound(B.begin() , B.end() , e) - B.begin();
33          ans += p1 - p2 + 1;
34          // Alice 在 Bob 后面
35          p1 = upper_bound(b.begin() , b.end() , A) - b.begin() - 1; // 找到比A大的最小数
                的下标，那么该下标-1得到的就是小于等于A的最大数的下标
36          p2 = lower_bound(b.begin() , b.end() , A - k) - b.begin();
37          if(b[p2] > A) continue ;
38          ans += p1 - p2 + 1;
39      }
40  }
41  cout << ans << '\n';
42  return 0;
43 }
```

复杂度优化

双层循环 + 判断的方法虽然十分简单，但本题的最大数据为 10^6，$O(n^2)$ 的做法显然是无法通过的。

那我们应该怎么办呢?

有以下两个选择。

1. 对双层循环 + 判断的方法进行优化

我们先来回顾一下双层循环 + 判断的每个步骤以及每个步骤需要的时间复杂度。

（1）找出 Alice，时间复杂度为 $O(n)$。

（2）找出 Bob，时间复杂度为 $O(n)$。

（3）判断 Alice 和 Bob 是否同时出现，时间复杂度为 $O(1)$。

事实上，第 (1) 步和第 (2) 步是两个不冲突的操作，即我们只需要一层循环或者两层**不嵌套**的循环即可找出所有的 Alice,Bob，而之所以使用**嵌套**循环，是为了便于第 3 步的判断。

什么意思?

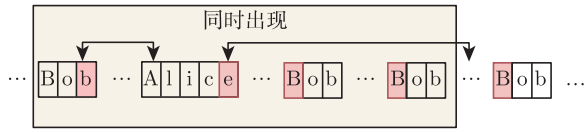
简单来说，第 (3) 步是针对于某一个 Alice 和某一个 Bob 进行判断，目的是判断这一对 (Alice,Bob) 能否为答案带来贡献。

由于每一对 (Alice, Bob) 都有可能为答案带来贡献，所以为了不计算错误，我们才使用**嵌套**的循环对每一对 (Alice,Bob) 都进行判断。但在极端数据下，(Alice,Bob) 的对数会是 n^2 级别的。因此，即使进行一次判断的时间复杂度仅为 $O(1)$，但想要通过判断来获取答案也需要进行 n^2 次判断。所以以判断的方式来求解答案并不可行。

2. *考虑其他方法*

除了十分暴力的双层循环 + 判断外，还有一种常见的解题套路 ——枚举 + 计算贡献。

对于文本中的任意一个 Alice，如果和它同时出现的 Bob 共有 x 个，那么它就能和这 x 个 Bob 组成共 x 对 (Alice,Bob)。由于这 x 对 (Alice,Bob) 都会对答案的值产生影响，且都和这个 Alice 相关，因此，我们可以称这个 Alice 对答案的贡献为 x，如下图所示。



显然，答案就是文本中所有 Alice 的贡献之和。

因此，枚举 + 计算贡献的整体思路为枚举文本中所有的 Alice，并计算它们对答案的贡献之和。

由于枚举文本中所有的 Alice 必然要消耗 $O(n)$ 的时间，所以，计算一个 Alice 对答案的贡献的方法必须要高效。那么，要怎么计算呢?

我们来分析一下。

对于一个 Alice，它对答案的贡献 = 与它同时出现的 Bob 的个数 = 与它相隔字符数不超过 k 的 Bob 个数。

如果再详细点，则 Alice 对答案的贡献可分为两部分。

（1）在它前面的、与 Alice 中的 A 相隔字符数不超过 k 的 Bob 个数。

（2）在它后面的、与 Alice 中的 e 相隔字符数不超过 k 的 Bob 个数。

这两部分的求解方法本质上是差不多的，所以我们只要能求出其中一部分，另一部分也自然就会求了。

下面我们就以第 (2) 部分为例尝试求解。

对于某个 Alice，我们会通过枚举来得到它，所以它的具体信息我们也都能得到，就不用过多地在它身上进行分析。

对于在该 Alice 后面的、与该 Alice 中 e 的相隔字符数不超过 k 的 Bob，我们一无所知，我们仅仅只有在 Alice 后面、相隔字符数不超过 k 这两个条件。

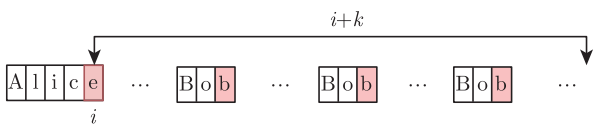
于是问题来了，仅凭这两个条件够吗?

够的。为什么?

我们接着往下分析。

假设 Alice 中的 e 在文本中的下标为 i，那么我们可以确定与该 Alice 同时出现的这些 Bob 中的 B 的下标一定会大于 i，也一定会小于 $i+k+1$（与 i 之间的间隔数不超过 k）。

这么看来，我们要求的解其实就是 $[i+1, i+k]$ 这个区间内 Bob 的个数，如下图所示。



怎么求区间 $[i+1, i+k]$ 内 Bob 的个数呢? 十分简单。

事实上，这个问题就等价于给了一个序列，序列中存在某种特殊数。现在有若干次询问，每次询问会给出一个区间，让求出该区间内特殊数的个数。

为什么等价呢? 我们来进行一个文字替换。

（1）序列 $\rightarrow$ 文本。

（2）某种特殊数 $\rightarrow$ Bob （准确来说是 Bob 中的字母 B）。

（3）若干次询问，每次询问给出一个区间 $\rightarrow$ 每枚举一个 Alice，就会给定一个区间 $[i+1, i+k]$（i 表示 Alice 中字母 e 的下标）。

（4）求出该区间内特殊数的个数 $\rightarrow$ 求出 $[i+1, i+k]$ 内 Bob 的个数。

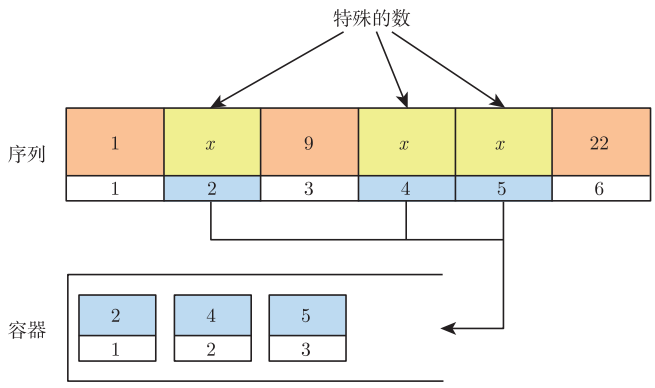
那要怎么解决这个问题呢? 方法颇多，下面列举几种。

1. 二分

按照从左到右的顺序将所有特殊数的下标存进一个容器当中，对于每次询问给出的区间（设为 $[L, R]$），二分找出容器中比 L 大的最小数的编号（容器内的编号），二分找出容器中比 R 小的最大数的编号（容器内的编号），那么两者相减的值再 $+1$ 即为所求。

每次查询的时间复杂度为 $O(\log n)$。

例如在下图中，查询区间为 [1,4]，分别进行如下操作。



1. 在容器中二分找出比1大的最小数，得到2，2在容器内的编号为1
2. 在容器中二分找出比4小的最大数，得到4，4在容器内的编号为2

答案为2−1+1=2

参考代码 5-2-2 【时间复杂度为 $O(n \log n)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
bool check(char x){
    if(x >= 'a' && x <= 'z') return true;
    if(x >= 'A' && x <= 'Z') return true;
    if(x >= '0' && x <= '9') return true;
```

```
    return false;
}
vector<int>B , b;
signed main(){
    int k ; string s;
    cin >> k;
    cin.get();
    getline(cin , s); // 整行读入字符串
    int n = s.size();
    B.push_back(-1000000000) , b.push_back(-1000000000); // 边界处理
    for(int i = 0 ; i < n ; i ++){
        if(i + 2 < n && s.substr(i , 3) == "Bob"){
            if(i - 1 >= 0 && check(s[i - 1]) || i + 3 < n && check(s[i + 3])) continue ;
            B.push_back(i + 1) , b.push_back(i + 3);
        }
    }
    B.push_back(1000000000) , b.push_back(1000000000); // 边界处理
    long long ans = 0;
    for(int i = 0 ; i < n ; i ++){
        if(i + 4 < n && s.substr(i , 5) == "Alice"){
            if(i - 1 >= 0 && check(s[i - 1]) || i + 5 < n && check(s[i + 5])) continue ;
            int A = i + 1 , e = i + 5;
            // Alice 在 Bob前面
            int p1 = upper_bound(B.begin() , B.end() , e + k) - B.begin() - 1; // 找到比e+k
                大的最小数的下标，那么该下标-1得到的就是小于等于e+k的最大数的下标
            int p2 = lower_bound(B.begin() , B.end() , e) - B.begin();
            ans += p1 - p2 + 1;
            // Alice 在 Bob 后面
            p1 = upper_bound(b.begin() , b.end() , A) - b.begin() - 1; // 找到比A大的最小数
                的下标，那么该下标-1得到的就是小于等于A的最大数的下标
            p2 = lower_bound(b.begin() , b.end() , A - k) - b.begin();
            if(b[p2] > A) continue ;
            ans += p1 - p2 + 1;
        }
    }
    cout << ans << '\n';
    return 0;
}
```

2. 前缀和

将特殊数的值视为 1，其他数的值视为 0，然后维护一个前缀和数组（命名为 sum[]）。这样的 sum[] 数组拥有一个特点，它每一项的值只会受到特殊数的影响（因为其他数的值被视

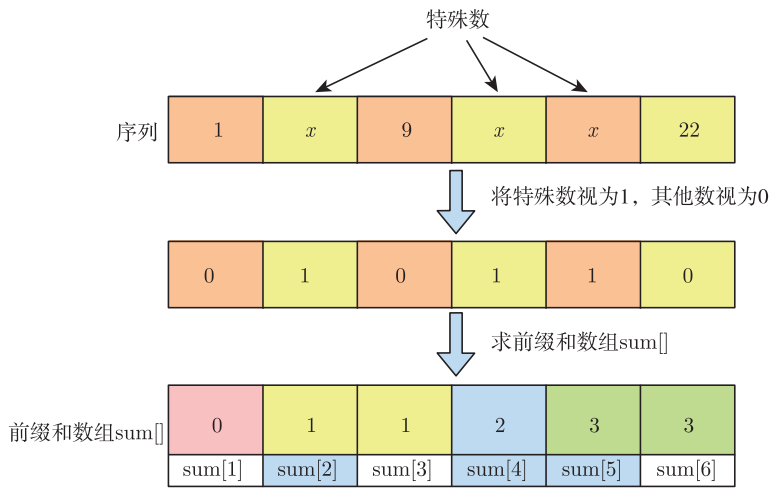
为了 0)。

一个特殊数对 sum[] 数组的影响为 1，所以在该前缀和数组中，第 i 项（sum[i]）表示的就是区间 $[1,i]$ 中特殊数的个数。那么对于每次询问给出的区间（设为 $[L,R]$），sum[R]$-$sum[$L-1$] 即为所求。

提示

sum[R] 表示区间 $[1,R]$ 特殊数的个数 = 区间 $[1,L]$ 特殊数的个数 + 区间 $[L+1,R]$ 特殊数的个数。

例如在下图中，查询区间为 [3,6]，分别进行如下操作。



1. 求出区间[1, 3−1]特殊数的个数：sum[3−1]=sum[2]=1
2. 求出区间[1, 6]特殊数的个数：sum[6]=3

答案：区间[3, 6]特殊数的个数=区间[1, 6]特殊数的个数−区间[1 ,2]特殊数的个数=sum[6]−sum[2]=3−1=2

参考代码 5-2-3 【时间复杂度为 $O(n)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
bool check(char x){
    if(x >= 'a' && x <= 'z') return true;
    if(x >= 'A' && x <= 'Z') return true;
    if(x >= '0' && x <= '9') return true;
    return false;
}
// 当 Alice 在 Bob 前面时，将所有 Bob 中的 B 视为 1，将其他字符视为 0，记录在数组 B[] 中，并
    用 sum1 表示数组 B[] 的前缀和
```

```
// 当 Alice 在 Bob 后面时，将所有 Bob 中的 b 视为 1，将其他字符视为 0，记录在数组 b[] 中，并用 sum2 表示数组 b[] 的前缀和
int B[N] , b[N] , sum1[N] , sum2[N];
signed main(){
    int k ; string s;
    cin >> k;
    cin.get();
    getline(cin , s); // 整行读入字符串
    int n = s.size();
    for(int i = 0 ; i < n ; i ++){
        if(i + 2 < n && s.substr(i , 3) == "Bob"){
            if(i - 1 >= 0 && check(s[i - 1]) || i + 3 < n && check(s[i + 3])) continue ;
            B[i + 1] = 1 , b[i + 3] = 1;
        }
    }
    for(int i = 1 ; i <= n ; i ++) sum1[i] = sum1[i - 1] + B[i] , sum2[i] = sum2[i - 1] + b[i];
    long long ans = 0;
    for(int i = 0 ; i < n ; i ++){
        if(i + 4 < n && s.substr(i , 5) == "Alice"){
            if(i - 1 >= 0 && check(s[i - 1]) || i + 5 < n && check(s[i + 5])) continue ;
            int A = i + 1 , e = i + 5;
            ans += sum1[min(n , e + k)/* e + k 可能大于 n , 因此要对 n 取 min 防止越界*/] - sum1[e + 1 - 1]; // Alice 在 Bob 前面
            ans += sum2[A - 1] - sum2[max(0 , A - 1 - k - 1)/* A - 1 - k - 1 可能小于 0 , 因此要对 0 取 max 防止越界*/]; // Alice 在 Bob 后面
        }
    }
    cout << ans << '\n';
    return 0;
}
```

3. 尺取法

本题也可用尺取法解决。

假设文本中共有 i 个 Alice、j 个 Bob。按照它们在文本中的前后位置关系，我们将它们分别打上编号，即 $\text{Alice}_1, \text{Alice}_2, \ldots, \text{Alice}_i$;$\text{Bob}_1, \text{Bob}_2, \ldots, \text{Bob}_j$。

Alice 与 Bob 同时出现有以下两种情况。

（1）Alice 在前、Bob 在后。

（2）Alice 在后、Bob 在前。

为了方便表述，下文我们分别称这两种情况为情况 (1)、情况 (2)。

对于 Alice_1：

（1）如果 Bob_1,Bob_2 的位置在 Alice_1 之前，却不能以情况 2 的方式与 Alice_1 同时出现，那么 Bob_1,Bob_2 同样无法与 $\text{Alice}_2, \text{Alice}_3, \ldots, \text{Alice}_i$ 以情况 2 的方式同时出现。

（2）如果 $\text{Bob}_3, \text{Bob}_4$ 的位置在 Alice_1 之前，能以情况 2 的方式与 Alice_1 同时出现，那么 Bob_3,Bob_4 有可能与 $\text{Alice}_2, \text{Alice}_3, \ldots, \text{Alice}_i$ 以情况 2 的方式同时出现。

简单来说，就是在 Alice_1 之前，却无法以情况 2 的方式与 Alice_1 同时出现的 Bob 也一定无法与 $\text{Alice}_2, \ldots, \text{Alice}_i$ 同时出现。

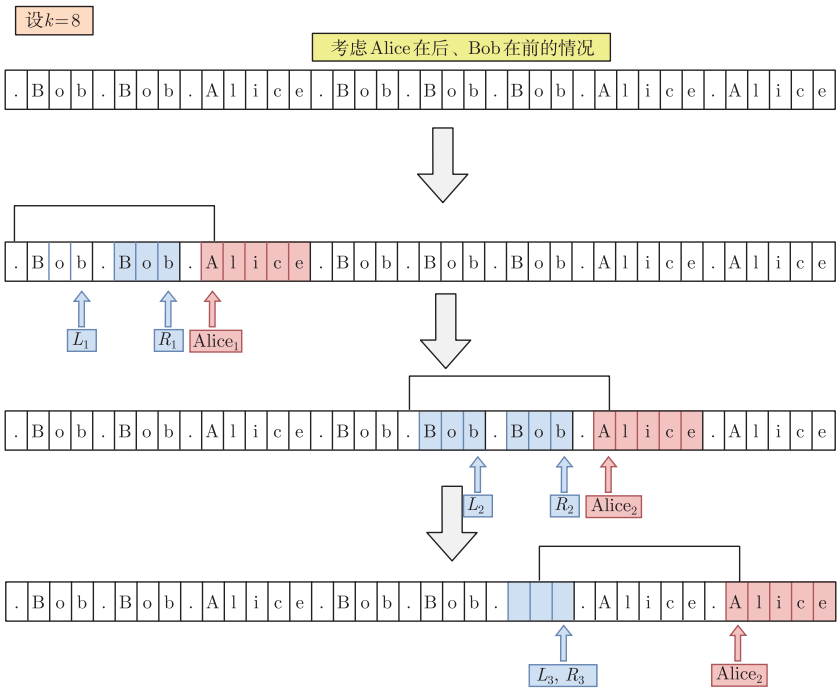
那么能与 $\text{Alice}_2, \ldots, \text{Alice}_i$ 同时出现的 Bob 要么在 Alice_1 之后，要么也一定能和 Alice_1 同时出现。所以有以下两种情况。

（1）如果我们用 L_1 表示能与 Alice_1 以情况 2 的方式同时出现的第一个 Bob，用 L_2, $L_3, \ldots, L_i$ 表示能与 $\text{Alice}_2, \text{Alice}_3, \ldots, \text{Alice}_i$ 以情况 2 的方式同时出现的第一个 Bob，则必然有 $L_1 \leqslant L_2 \leqslant \ldots \leqslant L_i$ 。

（2）如果我们用 R_1 表示能与 Alice_1 以情况 2 的方式同时出现的最后一个 Bob，用 $R_2, R_3, \ldots, R_i$ 表示能与 $\text{Alice}_2, \text{Alice}_3, \ldots, \text{Alice}_i$ 以情况 2 的方式同时出现的最后一个 Bob，则必然有 $R_1 \leqslant R_2 \leqslant \ldots \leqslant R_i$ 。

尺取法就是以此为核心，建立两指针，然后枚举 Alice，每次将两指针移动到当前枚举到的 Alice 所对应的 L 和 R 上，并计算该 Alice 的贡献（贡献 = 两指针之间的 Bob 的个数 $= R - L + 1$）。计算完当前 Alice 的贡献后，再枚举下一个 Alice，并将两指针移动到下一个 Alice 所对应的 L 和 R 上。

由于两指针在**整个**枚举的过程中都是在不断往右移动的，所以它们在**整个**枚举过程中最多移动 n 次，即移动指针的时间复杂度为 $O(n)$。



参考代码 5-2-4　【时间复杂度为 $O(n)$】

```cpp
#include<bits/stdc++.h>
using namespace std;
bool check(char x){
    if(x >= 'a' && x <= 'z') return true;
    if(x >= 'A' && x <= 'Z') return true;
    if(x >= '0' && x <= '9') return true;
    return false;
}
vector<int>A , B, e , b;
signed main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int k; string s;
    cin >> k;
    cin.get();
    getline(cin , s); // 整行读入字符串
    for(int i = 0 ; i < s.size() ; i ++){
        // 找 Alice，将找到的 Alice 中 A 的下标存储进容器 A中，将e的下标存储进容器 e 中
        if(i + 4 >= s.size() || s.substr(i , 5) != "Alice") continue ;
        if(i - 1 >= 0 && check(s[i-1])||i + 5 < s.size() && check(s[i + 5])) continue;
        A.push_back(i) ; e.push_back(i + 4);
    }
    for(int i = 0 ; i < s.size() ; i ++){
        // 找 Bob，将找到的 Bob 中 B 的下标存储进容器 B 中，将b 的下标存储进容器 b 中
        if(i + 2 >= s.size() || s.substr(i , 3) != "Bob") continue ;
        if(i - 1 >= 0 && check(s[i - 1]) || i + 3 < s.size()&&check(s[i + 3])) continue;
        B.push_back(i) ; b.push_back(i + 2);
    }
    long long ans = 0;

    // Alice 在后、 Bob 在前的情况
    int pl = 0 , pr = 0; // pl 所指向的第一个能和当前枚举到的 Alice 同时出现的 Bob，pr 所指向的则是最后一个
    for(int i = 0 ; i < A.size() ; i ++){
        // 若 pl 所指向的 Bob 和当前枚举到的 Alice的距离大于k， 则令 pl不断指向下一个Bob，直到pl指向的 Bob 和 Alice 的距离不超过 k
        while(pl < b.size() && b[pl] + k < A[i]) pl ++;

        // pr 指向的是最后一个 ，pl 是第一个，所以 pr 一定在 pl 的后面，因此 pr 可以直接从 pl 开始移动
        // b[pr] + k >= b[pl] + k > A[i]
```

```
39          pr = max(pr , pl);
40          // 若 pr 所指向的 Bob 的枚举小于当前枚举到的 Alice，则令 pr 不断指向下一个 Bob，直到
                pr 指向的 Bob 的位置大于 Alice，那么此时 pr - 1 即为最后一个能和当前 Alice 同时
                出现的 Bob
41          while(pr < b.size() && b[pr] < A[i]) pr ++ ;
42          ans += (pr - 1) - pl + 1;
43      }
44      // Alice 在前、Bob 在后的情况
45      pl = 0 , pr = 0;
46      for(int i = 0 ; i < e.size() ; i ++){
47          while(pl < B.size() && B[pl] < e[i]) pl ++ ;
48          pr = max(pr , pl);
49          while(pr < B.size() && B[pr] - k <= e[i]) pr ++ ;
50          ans += (pr - 1) - pl + 1;
51      }
52      cout << ans << '\n';
53      return 0;
54  }
```

5.3 子串分值和 (★★)

题目信息

2020 年省赛 - 程序设计题

- ✧ C/C++ B 组第 8 题；C/C++ C 组第 10 题
- ✧ Java B 组第 9 题

【问题描述】

对于一个字符串 S，我们定义 S 的分值 $f(S)$ 为 S 中出现的不同的字符个数。例如 $f(aba) = 2f(abc) = 3, f(aaa) = 1$。

现在给定一个字符串 $S[0...n1]$（长度为 n），请你计算对于所有 S 的非空子串 $S[i...j](0 \leqslant i \leqslant j < n)$，$f(S[i...j])$ 的和是多少。

【输入格式】

输入一行包含一个由小写字母组成的字符串 S。

其中，$1 \leqslant n \leqslant 10^5$。

【输出格式】

输出一个整数表示答案。

【样例输入】

ababc

【样例输出】

28

【评测用例规模与约定】

对于 20% 评测用例，$1 \leqslant n \leqslant 10$；

对于 40% 评测用例，$1 \leqslant n \leqslant 100$；

对于 50% 评测用例，$1 \leqslant n \leqslant 1000$；

对于 60% 评测用例，$1 \leqslant n \leqslant 10000$；

对于所有评测用例，$1 \leqslant n \leqslant 100000$。

题目分析

核心考点

- ✧ 枚举
- ✧ 逆向思维
- ✧ 计算贡献

本题包含多个得分点。下面就让我们根据能拿分值的不同，尝试不同的解法吧。

方式 1：能拿 40% 分数的做法

题意是让我们求解字符串 S 的所有非空子串的分值和，虽然看起来并不简单，但要想拿 40% 的分数，我们仅需考虑 $n \leqslant 100$ 的情况。

既然如此，我们能否直接按照题意进行模拟，即枚举子串，判断子串中每个字符的贡献，以求出答案呢？简单尝试一下。

依据枚举子串，判断子串中每个字符的贡献的思路，我们需要分以下 3 个步骤来实现。

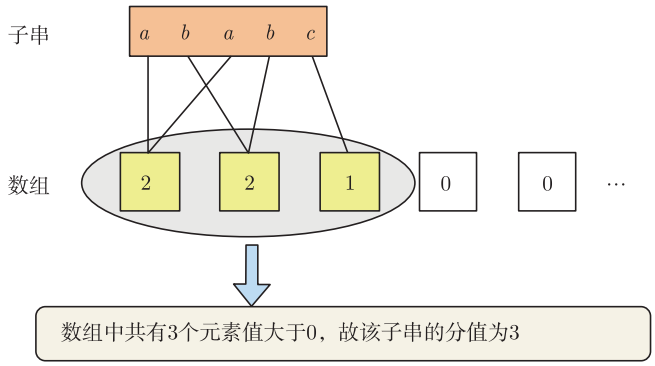
（1）枚举所有非空子串。

（2）求解枚举到的子串的分值。

（3）统计子串的分值和。

对于第（1）步，实现起来是十分简单的，我们仅需要使用双重循环：第一重循环用于确定子串的左端点，第二重循环用于确定子串的右端点。这样，我们就可确定好子串的左右端点，以达到确定好子串的目的。时间复杂度为 $O(n^2)$。

对于第（2）步，我们可以对每个子串开辟一个空间大小为 26 的数组，然后遍历子串，并用数组的每一位分别记录子串中每种字符（$a \sim z$）出现的次数，这样数组中值大于 0 的元素个数即为子串的分值，如下图所示。时间复杂度为 $O(n)$。



对于第（3）步，我们每求解完一个子串的分值后将该分值统一添加到一个变量上即可（该变量的值即为所有子串的分值和）。时间复杂度为 $O(1)$。

3 个步骤一起实现的总的时间复杂度为 $O(n^3)$。由于 $n \leqslant 100$，所以直接按题意模拟求解的方法可行。

参考代码 5-3-1　【时间复杂度为 $O(n^3)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int ans , cnt[26];
char s[N];
signed main(){
  cin >> s + 1;
  int n = strlen(s + 1);
  for(int l = 1 ; l <= n ; l ++){
      for(int r = l ; r <= n ; r ++){
          for(int i = 0 ; i <= 25 ; i ++) cnt[i] = 0;
          for(int i = l ; i <= r ; i ++) cnt[s[i] - 'a'] ++ ;
          int res = 0;
          for(int i = 0 ; i <= 25 ; i ++) if(cnt[i] > 0) res ++ ;
          ans += res;
        }
  }
  cout << ans << '\n';
  return 0;
}
```

方式 2：能拿 60% 分数的做法

显然，当 $n \leqslant 1000$ 时，$O(n^3)$ 的时间复杂度是完全无法解决的。那么，不妨让我们来试试优化这 3 个步骤。

（1）枚举所有非空子串 —— 时间复杂度 $O(n^2)$。

（2）求解枚举到的子串的分值 —— 时间复杂度 $O(n)$。

（3）统计子串的分值和 —— 时间复杂度 $O(1)$。

我们可以用最“朴素”的优化手段来分析每个步骤之间的联系。

在这 3 个步骤中，由于第三步的时间复杂度为 $O(1)$，已经达到最优，所以我们着重分析第（1）步和第（2）步之间的联系。

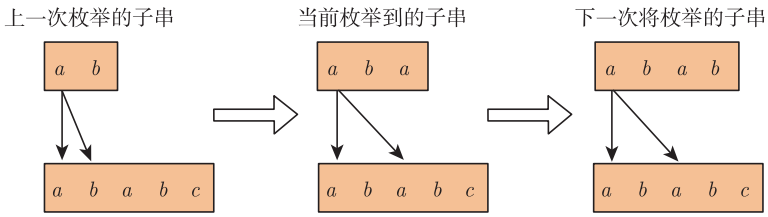
对于第（1）步，由于一个长度为 n 的字符串个数等于 C_n^2（从 n 个字符中任选一个作为左端点，再任选一个作为右端点），是 n^2 级别的，因此以 $O(n^2)$ 的时间复杂度枚举是没有问题的。

对于第（2）步，由于我们并不清楚当前枚举到的子串的信息，因此我们需要**从头到尾**计算当前子串的分值。“**从头到尾**”这 4 个字就相当于每次枚举到子串时，我们都把它当作一个**完全未知**的字符串处理。

但它真的是**完全未知**的吗？并不是。

回顾一下我们枚举非空子串的方式：先枚举确定左端点，再枚举确定右端点。

这种枚举方式的好处是可以使得我们当前枚举到的子串对比上一次枚举到的子串只多一个字符，如下图所示。



因此，对于一个子串，我们并不需要从头到尾重新计算它的分值，而是可以继承上一个子串的分值，再对比上一个子串多出来的一个字符进行计算。这样，我们每次计算的时间复杂度就为 $O(1)$。注意要开 long long 来存储答案。

参考代码 5-3-2　【时间复杂度为 $O(n^2)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e4 + 10;
long long ans; // 注意，此时int 可能存不下所有子串的分值和
int cnt[26];
char s[N];
signed main(){
    cin >> s + 1;
    int n = strlen(s + 1);
    for(int l = 1 ; l <= n ; l ++){
        for(int i = 0 ; i <= 25 ; i ++) cnt[i] = 0;
        int res = 0;
```

```
13          for(int r = l ; r <= n ; r ++){
14              if(cnt[s[r] - 'a'] == 0) res ++ ;
15              cnt[s[r] - 'a'] ++ ;
16              ans += res;
17          }
18      }
19      cout << ans << '\n';
20      return 0;
21  }
```

方式 3：满分做法

由于长度为 n 的字符串的子串数量是 n^2 级别的，所以要想通过枚举所有子串来计算答案肯定是不行的。

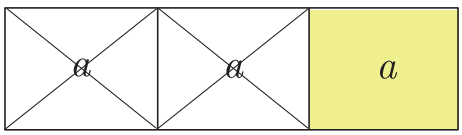
那怎么办呢?

我们可以采用逆向思维（更换枚举对象），也就是说，既然枚举子串，判断子串中每个字符的贡献的思路不行，我们就反过来尝试枚举字符，计算字符能对多少子串产生贡献，即考虑 S 的每个字符对答案的贡献。

但要怎么计算一个字符能对多少个子串产生贡献呢?

我们来分析一下。

一个子串的贡献等于该子串不同的字符个数，也就是对于字符串 aaa 而言，3 个 a 中只有一个 a 可以对其产生贡献。那么，我们不妨将第 1 个、第 2 个 a 的贡献抹去，认为 aaa 中只有第 3 个 a 产生了贡献，如下图所示，即当一个字符 char 在字符串中出现了多次时，我们只计算**最靠右**的 char 对答案的贡献（其他 char 在该字符串的贡献都被抹去）。

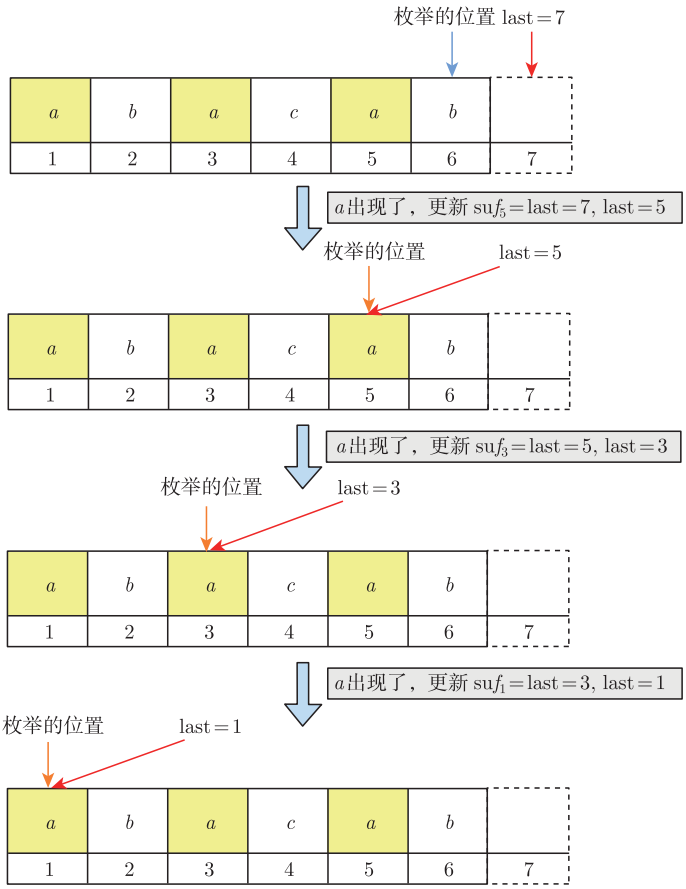


于是，一个字符能产生贡献的区间就为 [1 ~ 该字符下一次出现的位置 −1]。我们可以定义一个 suf 数组，其中 suf_i 表示第 i 个字符下一次出现的位置。

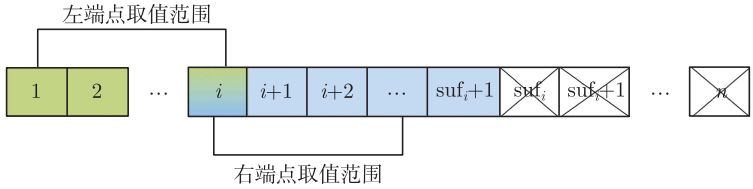
suf_i 怎么求呢?

十分简单。

我们考虑用一个变量（如 last）记录第 i 个字符上一次出现的位置。当第 i 个字符再度出现时，$\text{suf}_i = \text{last}$，同时更新 $\text{last} = i$，如下页图所示。



那么要想让第 i 个字符 S_i 能对答案产生贡献，则包含它的子串的左端点的取值范围必须为 $[1,i]$ 、右端点的取值范围必须为 $[i+1,\text{suf}_{S_i}-1]$（若右端点的取值大于等于 suf_{S_i}，则 S_i 对该子串的贡献将被抹除）。



在限制了左、右端点的取值范围后所能构造出的子串的个数（第 i 个字符对答案的贡献）就为左端点的取值范围 $(1\sim i)\times$ 右端点的取值范围 $i\sim \text{suf}_i-1=(i-1+1)\times(\text{suf}_i-1-i+1)=i\times(\text{suf}_i-i)$。

答案等于所有字符产生的贡献，即 $\text{ans}=\sum_{i=1}^{n} i\times(\text{suf}_i-i)$（注意要开 `long long` 来存储答案）。

参考代码 5-3-3　【时间复杂度为 $O(n)$】

```
1  #include<bits/stdc++.h>
2  using namespace std;
```

```cpp
const int N = 1e5 + 10;
long long ans;
char s[N];
int suf[N] , last[30];
signed main(){
    cin >> s + 1;
    int n = strlen(s + 1);
    for(int j = 0 ; j <= 25 ; j ++) last[j] = n + 1;
    for(int i = n ; i >= 1 ; i --){
        suf[i] = last[s[i] - 'a'];
        last[s[i] - 'a'] = i;
    }
    for(int i = 1 ; i <= n ; i ++){
        // 注意 i * (suf[i] - i) 也有可能超出 int 的范围
        ans += 1ll * i * (suf[i] - i);
    }
    cout << ans << '\n';
    return 0;
}
```

5.4 字串排序 (★★★★)

题目信息

2020 年省赛 - 程序设计题

- ✧ C/C++ A 组第 10 题；C/C++ B 组第 10 题
- ✧ Java A 组第 10 题

【问题描述】

小蓝最近学习了一些排序算法，其中冒泡排序让他印象深刻。

在冒泡排序中，每次只能交换相邻的两个元素。

小蓝发现，如果对一个字符串中的字符排序，只允许交换相邻的两个字符，在所有可能的排序方案中，冒泡排序的总交换次数是最少的。

例如，对于字符串 ‘lan’ 排序，只需要 1 次交换。对于字符串 ‘qiao’ 排序，总共需要 4 次交换。

小蓝的幸运数字是 V，他想找到一个只包含小写英文字母的字符串，对这个串中的字符进行冒泡排序，正好需要 V 次交换。请帮助小蓝找一个这样的字符串。如果可能找到多个，请告诉小蓝最短的那个。如果最短的仍然有多个，请告诉小蓝字典序最小的那个。请注意字符串中可以包含相同的字符。

【输入格式】

输入一行包含一个整数 V $(1 \leqslant V \leqslant 10^4)$，为小蓝的幸运数字。

【输出格式】

输出一个字符串，为所求的答案。

【样例输入 1】

4

【样例输出 1】

bbaa

【样例输入 2】

100

【样例输出 2】

jihgfeeddccbbaa

【评测用例规模与约定】

对于 30% 评测用例，$1 \leqslant V \leqslant 20$；

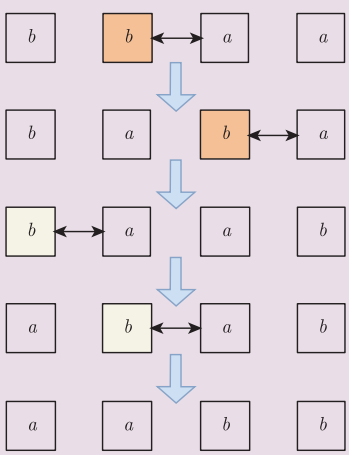
对于所有评测用例，$1 \leqslant V \leqslant 10000$。

懒人速读

> 给定一个常数 V，请构造出一个只包含小写英文字母的字符串，使得该字符串满足以下条件。
>
> (1) 冒泡排序（从小到大）的交换次数为 V。
>
> (2) 若存在多个这样的字符串，则选择其中长度最短的。
>
> (3) 若最短的仍然存在多个，则选择其中字典序最小的。

举例说明

下图演示了对字符串 *bbaa* 进行冒泡排序的过程。



题目分析

核心考点

- ✧ DFS
- ✧ 冒泡排序 & 逆序对
- ✧ 思维分析

这是一个字符串构造问题。

根据题意，我们构造的字符串需依次满足以下 3 个约束条件。

（1）字符串冒泡排序的交换次数为 V。

（2）字符串的长度最短。

（3）字符串的字典序最小。

在这 3 个条件中，后两个是我们相对熟悉的约束条件，但可能有人对第一个约束条件所涉及的冒泡排序的交换次数感到十分陌生。什么是冒泡排序的交换次数呢?

对此，我们先来简单回顾一下冒泡排序。

冒泡排序称得上是最基础的排序算法，它是基于交换的排序。

冒泡排序的步骤如下。

（1）从数组的第一个元素开始，将相邻的两个元素进行比较，直到最后两个元素完成比较。在比较过程中，如果前面的元素比后面的元素大，就**交换**它们的位置。待整个过程完成后，数组中最后一个元素自然就是最大值，也就完成了一轮比较。

（2）除了最后一个元素，将剩余的元素按照第一步的方法进行两两比较，这样就可以将数组中第二大的元素交换到倒数第二个位置上。

（3）重复上述步骤，将值第三大的元素交换到倒数第三个位置，将值第四大的元素交换到倒数第四个位置。

根据这些步骤可得，冒泡排序的交换次数即为在整个比较的过程中，前一个元素大于后一个元素的情况数。

那么，我们要怎么求冒泡排序的交换次数呢?

显然，可以用最“朴素”的方法——模拟一遍冒泡排序的过程，以统计交换次数。

我们也可以深入分析冒泡排序的性质，来看看是否有更好的办法。

冒泡排序的性质：在冒泡排序的过程中，只有相邻的两个元素才会进行交换。

那么对于数组（字符串）中的某个元素 x，如果在 x 之前有比 x 大的元素（如 y），则 x 就一定会和 y 进行一次交换。

为什么呢?

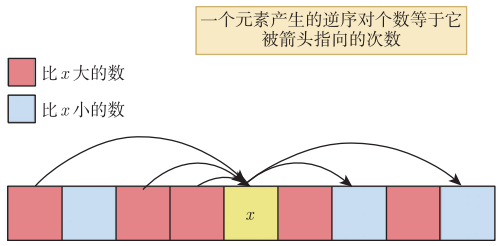
因为在冒泡排序结束后，数组是有序的，即 y 一定会在 x 的后面 —— y 一定会被交换到 x 的后面。由于只有相邻的两个元素才能进行交换，所以 x 一定会和 y 进行一次交换。

同理，如果在 x 之后有比 x 小的元素（如 a），那么 x 也一定会和 a 进行一次交换。

对于这样的 (x,y)、(x,a)，我们称之为逆序对。只要出现了一个逆序对，我们就必然要进行一次交换。所以，冒泡排序的交换次数就是数组（字符串）中的逆序对个数。

根据逆序对的定义可得，对于元素 x，由它产生的逆序对的个数 $=$ 在 x 之前比 x 大的数的个数 $+$ 在 x 之后比 x 小的数的个数。

不过为了避免 (x,y)、(y,x) 被重复计算两次，我们可以认为由 x 产生的逆序对个数 $=$ 在 x 之前比 x 大的数的个数。在 x 之后比 x 小的数的个数不被计算在 x 产生的逆序对个数里面，而归纳于在 x 之后比 x 小的数上，如下图所示。



有了逆序对的概念后，我们不妨来思考一个问题，即在长度固定的情况下，如何构造数组可以使得逆序对的个数最多呢?

简单分析一下。

对于一个长度为 n 的数组，它产生的逆序对个数可分为两个独立的部分。

（1）前 $n-1$ 个元素产生的逆序对个数；

（2）第 n 个元素产生的逆序对个数。

要想使该数组的逆序对个数最多，就要保证这两个独立的部分分别产生的逆序对个数最多。

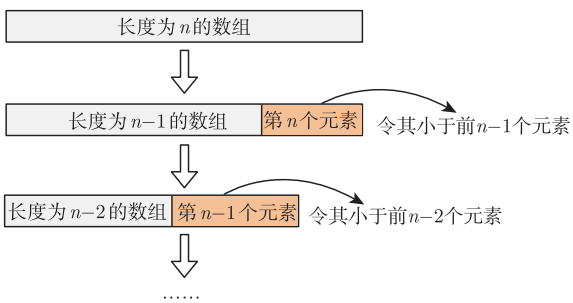
已知第 n 个元素产生的逆序对的个数即前 $n-1$ 个元素中比它大的数的个数，那么为了使第 n 个元素产生的逆序对的个数最多，就要令其小于前 $n-1$ 个元素。

至于前 $n-1$ 个元素产生的逆序对个数，我们也可以将其分为两个独立的部分：

（1）前 $n-2$ 个元素产生的逆序对个数；

（2）第 $n-1$ 个元素产生的逆序对个数。

同理，要想 $n-1$ 个元素产生的逆序对个数最多，就要让第 $n-1$ 个元素小于前 $n-2$ 个元素，然后继续将前 $n-2$ 个元素分为两个部分，如下图所示。



不难发现，当一个数组是**严格递减序列**时，其可产生的逆序对个数最多。

此外，如上文所说，每出现一个逆序对后，冒泡排序的交换次数就会 $+1$。那么要是反过来，每减少一个逆序对后，冒泡排序的交换次数是不是也会 -1 呢?

就逆序对的定义而言，是的。

于是我们可以得出，如果某个数组（字符串）冒泡排序的交换次数为 $x(x>0)$，那我们就一定可以通过改变该数组（字符串）中某两个元素的位置（减少逆序对个数）使得数组（字符串）冒泡排序的交换次数变为 $x-1$。当然，我们也可以继续改变某两个元素的位置，使得数组（字符串）冒泡排序的交换次数变为 $x-2$、$x-3$、...、0。

分析完冒泡排序的性质后，我们来讨论一下如何解题。

方式 1：能拿 30% 分数的做法

因为 $V \leqslant 20$，范围特别小。为了减少思维量，我们可以直接用 DFS 来暴力搜索答案。

先来考虑一下 DFS 的搜索边界。

（1）字符串冒泡排序的交换次数为 V—— 当搜索过程中构造出的字符串冒泡排序的交换次数大于 V 时，停止搜索。

（2）字符串的长度最短 —— 随机构造一个字符串 **gfedcba**，其逆序对个数 $=0+1+2+3+4+5+6=21$。由于 $V \leqslant 20$，所以答案对应的字符串的最短长度一定不超过 **gfedcba** 的长度，即当搜索过程中构造出的字符串的长度大于 7 时停止搜索。

（3）字符串的字典序最小。因为 **gfedcba** 的逆序对个数为 21，所以我们只需使用字符 $a \sim g$ 构造。

再来讨论一下 DFS 过程中的一些细节问题。

（1）如何计算搜索过程中构造出的字符串冒泡排序的交换次数?

（2）什么情况下更新答案?

答案如下。

（1）虽然求逆序对有更好的办法，但为了简化 DFS 的过程，模拟冒泡排序并统计交换次数即可。

（2）当构造的字符串的交换次数等于 V 时，有以下两种情况。

- 若其长度小于答案长度，则更新答案。
- 若其长度等于答案长度，且字典序小于答案，则更新答案。

综上，我们即可轻松写出 DFS 求解答案，其中时间复杂度分别包含搜索的复杂度和模拟冒泡排序的复杂度。

参考代码 5-4-1　【时间复杂度为 $O(7^7 \times 7^2)$】

```
#include<bits/stdc++.h>
using namespace std;
int V;
string ans = "gfedcba";
void dfs(string s){
```

```
  int cnt = 0 , len = s.size();
  if(len > 7) return ; // 最大长度不会超过 7
  string t = s;
  // 模拟冒泡排序
  for(int i = 0 ; i < len - 1 ; i ++){
    for(int j = 0 ; j < len - i - 1 ; j ++){
      if(t[j] > t[j + 1]) {
        cnt ++ ;
        swap(t[j] , t[j + 1]);
        if(cnt > V) return ;
      }
    }
  }
  if(s.size() > ans.size()) return ;
  if(cnt == V){
    if(s.size() == ans.size()) ans = min(ans , s);
    else ans = s;
    return ;
  }
  for(int i = 0 ; i <= 6 ; i ++) dfs(s + (char)(i + 'a')); // 1 ~ 6 分别表示字符 a ~ g
}
signed main(){
  cin >> V;
  dfs("");
  cout << ans << '\n';
  return 0;
}
```

方式 2：满分做法

已知我们构造的字符串需依次满足以下 3 个条件

（1）字符串冒泡排序的交换次数为 V。

（2）字符串的长度最短。

（3）字符串的字典序最小。

在 30% 分数的做法中，由于 V 的数据范围很小，所以我们可以大范围搜索字符串，并从中轻松找出满足以上 3 个条件的答案。

但在 $V \leqslant 10000$ 时，我们还能这么做吗？显然不能。

要想通过搜索字符串找出答案，无疑等同于大海捞针。所以，我们只能分析满足这 3 个条件的字符串的特点，以此来锁定答案。

在这 3 个条件中，由于 V 是题目给定的，所以条件 1 是已经明确了的。那么按照优先级，我们需要先处理条件 2，即确定答案对应的字符串的最短长度。

1. 求字符串的最短长度

如果按照常规的逻辑思考，我们会先求出所有交换次数等于 V 的字符串，然后再从这些字符串中找出最短长度（这个最短长度即为所求）。

但问题在于，我们根本无法求出所有交换次数等于 V 的字符串，甚至当 V 的数值大了之后，想求一个交换次数等于 V 的字符串也并不简单。

那怎么办呢?

别忘了，我们还可以用上前面推导出的逆序对的性质。

逆序对性质

如果某个数组（字符串）冒泡排序的交换次数为 $x(x>0)$，那就一定可以通过改变该数组（字符串）中某两个元素的位置（减少逆序对个数）使得数组（字符串）冒泡排序的交换次数变为 $x-1$、$x-2$、$x-3$、……、0。

不妨让我们随机构造一个长度为 len 的字符串 S，并求出它的逆序对个数。那么如果 S 的逆序对个数大于等于 V，则我们要求的最短长度一定不会超过 len（至少在最坏的情况下，我们可以通过改变 S 的字符顺序以构造出长度为 len，且逆序对个数为 V 的字符串）。

不过随机构造的意义并不大，因为如果我们随机构造的字符串的逆序对个数小于 N，那就无法确定最短长度的范围。

因此，我们需要进行一些有意义的构造，比如构造一个长度为 len、逆序对个数最多的字符串 S。这样当 S 的逆序对个数大于等于 V 时，我们就能确定我们要求的最短长度的范围为 $1\sim$ len；当 S 的逆序对个数小于 V 时，我们就能确定我们要求的最短长度一定大于 len。

我们可以定义 $\max_i$ 来表示长度为 i 的字符串的最大逆序对个数。这样在所有的 $\max_{\text{len}} \geqslant V$ 中，最小的 len 就是我们要的最短长度。

那么问题来了，$\max_i$ 要怎么求呢?

根据前面的分析我们得知，在长度相同的情况下，逆序对越多，交换的次数就越多。为保证逆序对尽可能多，我们构造的字符串就要尽量是**严格递减序列**。

于是当长度 len $=1$ 时，我们构造的字符串要形如 **a**，逆序对个数为 0，交换次数为 0。

当长度 len $=2$ 时，我们构造的字符串要形如 **ba**，逆序对个数为 1，交换次数为 1。

当长度 len $=3$ 时，我们构造的字符串要形如 **cba**，逆序对个数为 3，交换次数为 3。

……

当长度 len $=26$ 时，我们构造的字符串要为 **zyxwvutsrqponmlkjihgfedcba**，其逆序对个数为 325，交换次数为 325。

此时又出现了一个问题，即题目给定的 V 的最大取值为 10000，而 325 是远远不及 10000 的，所以按照步骤，我们接下来要构造长度大于 26 的、逆序对个数最多的字符串。但根据题意，我们能使用的字符只有 $a\sim z$，而在长度为 26 时我们已经使用了 $a\sim z$ 中的所有字符，即当长度大于 26 时，我们是无法构造出**严格递减序列**的。

这时候我们就要考虑，当字符串不是**严格递减序列**时，我们要如何构造使其交换次数最多?

先来一个子问题：假设我们现在能使用的字符只有 **a,b,c**，要求我们构造出一个长度为 7，且交换次数最多的字符串该怎么办?

根据逆序对的性质可得，每当往字符串中的某个位置插入一个新字符后所能产生的逆序对个数 = 在该位置之前比该字符大的字符的数量 + 在该位置之后比该字符小的字符的数量。

由于我们插入字符的位置是任意的，所以插入新字符后所能产生的逆序对个数最多会等于字符串长度 − 该字符的数量，即每次插入字符数量最少的字符所能产生的逆序对个数是最多的。

每次插入字符数量最少的字符相当于让**字符串被所有字符平分**。所以当字符串长度为 len 时，字符数量要么为 $\left\lfloor\frac{\text{len}}{26}\right\rfloor+1$，要么为 $\left\lfloor\frac{\text{len}}{26}\right\rfloor$（其中字符数量为 $\left\lfloor\frac{\text{len}}{26}\right\rfloor+1$ 的字符有 len%26 个，字符数量为 $\left\lfloor\frac{\text{len}}{26}\right\rfloor$ 的字符有 26−len%26 个，它们产生的逆序对个数分别为 $\text{len}-\left\lfloor\frac{\text{len}}{26}\right\rfloor+1$、$\text{len}-\left\lfloor\frac{\text{len}}{26}\right\rfloor$）。

于是我们可得如下结果。

- 子问题的答案为 **ccbbaaa** 或 **ccbbbaa** 或 **cccbbaa**。
- 长度为 len 的字符串的最大逆序对个数为

$$\frac{(\text{len}-(\text{len}/26+1))\times(\text{len}/26+1)\times(\text{len}\%26)+(\text{len}-\text{len}/26)\times(\text{len}/26)\times(26-\text{len}\%26)}{2}$$

除 2 是因为形如 (x,y)、(y,x) 的逆序对被计算了两次。

得到 $\max_{\text{len}}$ 的计算方式后我们就可以通过从小到大枚举以找出 $\max_{\text{len}}\geqslant V$ 的最小 len。

2. 已知长度求最小字典序

在解决了字符串的长度问题后，我们接着来求解字符串的最小字典序。

为了方便表示，我们设最短长度为 len。

因为字典序是从头到尾比较字符大小的，所以为了保证字典序最小，我们要从字符串第 1 个位置开始一个一个构造。

怎么构造呢?

自然要结合约束条件制定规则。

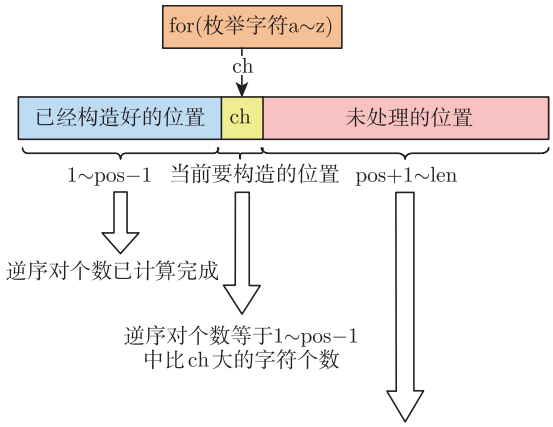
- 构造完当前位置的字符后，必须保证构造后续位置的字符时，至少存在一种能令字符串逆序对个数大于等于 V 的方法（约束条件 1）。
- 能使用小的字符就使用小的字符（约束条件 3）。

于是，对于每个位置，我们就可从小到大枚举字符 ch($\text{ch}\in[a\sim z]$)，并判断当前位置的字符为 ch 时是否可以构造出长度为 len，且逆序对个数大于等于 V 的字符串。

该怎么判断呢?

我们来分析一下。

假设我们当前要构造的位置为 pos，那么存在的情况如下页图所示。



需要插入 len−pos 个字符。

若插入的字符为 x ，则产生的逆序对个数 = (1~pos 中比 x 大的字符的个数)+
(pos+1~len 中比 x 大的字符的个数)+(pos+1~len 中比 x 小的字符的个数)
= (1~ pos 中比 x 大的字符的个数}+
(pos+1~len中的字符数−pos+1~len中等于 x 的字符的个数)

- $1 \sim (\text{pos}-1)$ 位置的字符肯定已经构造完成（因为我们是从字符串第 1 个位置开始逐一构造的），设它们产生的逆序对个数为 pre。
- 当前位置选择字符 ch 可以新增的逆序对个数为 $1 \sim (\text{pos}-1)$ 中比 ch 大的字符的个数（记为 add1）。
- 对于 $(\text{pos}+1) \sim \text{len}$ 的位置，我们总共要添加 $\text{len}-\text{pos}$ 个字符，且要保证这些字符能够产生最多的逆序对。我们可以“暴力”点**一个个**插入字符（**注意，插入并不一定是按顺序插入，可能先在第 $\text{pos}+2$ 的位置插入字符，再在第 $\text{pos}+1$ 的位置插入字符**）来保证产生的逆序对个数最多，即每次插入都选择能产生最多逆序对的字符，并统计插入后能产生的总逆序对个数（记为 add2）。

如果 $\text{pre}+\text{add1}+\text{add2} \geqslant V$，我们就可以在第 pos 个位置放置字符 ch 来继续构造第 $\text{pos}+1$ 个位置的字符；反之，我们就判断第 pos 个位置的字符为 $\text{ch}+1$ 时是否可以构造出长度为 len，且逆序对个数大于等于 V 的字符串，当 $V=10000$ 时，$\text{len}=145$，所以时间复杂度是没有问题的。

参考代码 5-4-2 【时间复杂度为 $O(26\text{len}^2)$】

```
#include<bits/stdc++.h>
using namespace std;
int V , len , pre , cnt[27] , sum[27];
int get_max(int len){ // 计算长度为 len 时的最大逆序对个数
  return ((len - (len / 26 + 1)) * (len / 26 + 1) * (len % 26) + (26 - len % 26) * (len /
      26) * (len - len / 26)) / 2;
}
// 当前位置上放置的是第 x 个字母，后面还有 n 个位置
bool check(int x , int n){
  memset(cnt , 0 , sizeof(cnt));
```

```
10    int add1 = 0; // 当前位置放置 x 后新增的逆序对个数
11    int add2 = 0; // 在后续位置放置字母最多可以新增的逆序对个数
12    // 统计已放置好的部分（在 x 之前）比 x 大的数的字符个数，即放置 x 后新增的逆序对个数 (add1)
13    for(int j = 26 ; j >= x + 1 ; j --) add1 += sum[j];
14    sum[x] ++ ; // 在当前位置放置 x，固 x 的个数增加 1
15    for(int k = 1 ; k <= n ; k ++){
16        int ma = -1; // 在第 k+pos 个位置上放置字母后最多能新增的逆序对个数
17        int p = 0;
18        int num = 0; // [1,pos] 中大于 j 的字母个数
19        for(int j = 25 ; j >= 0 ; j --){
20            // 插入第 j 个字符产生的逆序对个数 = [1,pos] 中比 j 大的字符个数,+ [pos+1,len] 中的
                  字符数 - [pos+1,len] 中字符 j 的个数
21            if(k - 1 - cnt[j] + num > ma){ // k - 1 表示即将插入第 j 个字符时 [pos+1,len] 中的
                  所有字符的个数// cnt[j] 表示即将插入第 j 个字符时 [pos+1,len] 中字符 j 的个数
22                ma = k - 1 - cnt[j] + num;
23                p = j;
24            }
25            num += sum[j];
26        }
27        add2 += ma , cnt[p] ++;
28    }
29    if(pre + add1 + add2 >= V) {
30        pre += add1;
31        return true;
32    }
33    else {
34        sum[x] -- ;
35        return false;
36    }
37  }
38  signed main(){
39    string ans = "";
40    cin >> V;
41    for(int i = 1 ; ; i ++) {
42        if(get_max(i) >= V){
43            len = i;
44            break ;
45        }
46    }
47    // 构造自断续最小的字符串：依次在第i个位置上放置字符
48    for(int i = 1 ; i <= len ; i ++){
49        for(int j = 0 ; j <= 25 ; j ++){ // 在位置i上试探性地放置第j个字母
```

```
50          if(check(j , len - i)){
51              ans += char(j + 'a');
52              break ;
53          }
54      }
55   }
56   cout << ans << '\n';
57   return 0;
58 }
```

5.5 巩固与练习

题目	难度	题目年份	组别
航班时间	★★	2018 年省赛	• C/C++ A 组第 6 题 • Java A 组第 6 题
子串分值	★★	2020 年省赛	• C/C++ A 组第 8 题 • Java A 组第 8 题；Java C 组第 10 题
切开字符串	★★★	2015 年国赛	• C/C++ A 组第 5 题；C/C++ C 组第 6 题 • Java A 组第 5 题

第6章 动态规划

CHAPTER 6

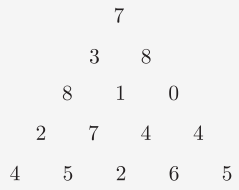
6.1 数字三角形 (★★)

题目信息

2020 年省赛 - 程序设计题

- C/C++ C 组第 8 题
- Java B 组第 8 题；Java C 组第 8 题
- Python 组第 8 题

【问题描述】



上图给出了一个数字三角形。从三角形的顶部到底部有很多条路径。对于每条路径，把路径上面的数加起来可以得到一个和，你的任务就是找到最大的和。

路径上的每一步只能从一个数走到下一层和它最近的左边的那个数或者右边的那个数。此外，向左下走的次数与向右下走的次数相差不能超过 1。

【输入格式】

输入的第一行包含一个整数 N $(1 \leqslant N \leqslant 100)$，表示三角形的行数。

下面的 N 行给出数字三角形。数字三角形上的数都是 0 至 100 之间的整数。

【输出格式】

输出一个整数表示答案。

【样例输入】

```
    5
    7
    3 8
    8 1 0
    2 7 4 4
    4 5 2 6 5
【样例输出】
    27
```

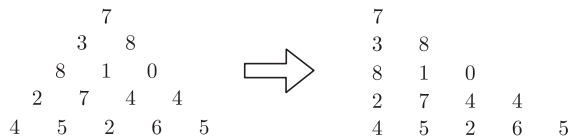
题目分析

核心考点

- ✧ 思维分析
- ✧ DFS
- ✧ 动态规划

本题是十分经典的动态规划题。

为了方便存储与操作，可以将题目描述中的等腰三角形转换为直角三角形，如下图所示。



将其转换为直角三角形后，就可以用一个二维数组（$a[][]$）来存储它。这样三角形第 i 行的第 j 个数字就可以通过 $a[i][j]$ 来表示。同时原来向左下走就变为了向下走，本题下文解析所说的“向下走”即为“向左下走”，向右下走还是向右下走。

在处理完样例输入后，我们来尝试对题目进行求解。

由于“向左下走的次数与向右下走的次数相差不能超过 1”这个限制条件看上去比较复杂，所以我们可将该题分解为两个子问题: 第一个子问题是不考虑限制条件，只解决如何走的问题；第二个子问题是加上这个限制条件找到走到底部的最大路径和，从而完整解答出本题。对于每个子问题，都分别用 DFS 和动态规划两种方式来解题, 大家可进一步体验这两种方式的特点。

解决子问题 1：DFS 模式

题目要求我们从三角形的顶部开始走，而顶部只有一个数字 $a[1][1]$，所以所走路径的起点一定是 $a[1][1]$。

由于我们每一步只能向下走或者向右下走，因此 $a[1][1]$ 的下一步只有以下两种可能。

（1）走向 $a[2][1]$。

（2）走向 $a[2][2]$。

具体要走向哪个呢？我们并不好确定。

我们能得知的是，如果设 $\mathrm{dfs}(i,j)$ 表示从 $a[i][j]$ 走到底部的最大路径和，那么我们一定会走向 $\max(\mathrm{dfs}(2,1),\mathrm{dfs}(2,2))$ 所对应的路径，即从 $a[1][1]$ 走到底部的最大路径和：

$$\mathrm{dfs}(1,1)=\max(\mathrm{dfs}(2,1),\mathrm{dfs}(2,2))+a[1][1].$$

在该式子中，$\mathrm{dfs}(1,1)$ 是我们要求解的，$a[1][1]$ 是已知的，$\mathrm{dfs}(2,1)$、$\mathrm{dfs}(2,2)$ 是未知的。显然，只要存在未知数，就无法求解 $\mathrm{dfs}(1,1)$ 的值。因此，必须先求解 $\mathrm{dfs}(2,1)$ 和 $\mathrm{dfs}(2,2)$。

但要怎么求解它们呢？

我们以求解 $\mathrm{dfs}(i,j)$ 为例。

当 $a[i][j](i<n)$ 不在底部时，每一步只能向下走或者向右下走，因此 $a[i][j]$ 的下一步只有两种可能。

（1）走向 $a[i+1][j]$。

（2）走向 $a[i+1][j+1]$。

当 $a[i][j](i=n)$ 在底部时，我们将无法再走任何一步。

由此可得

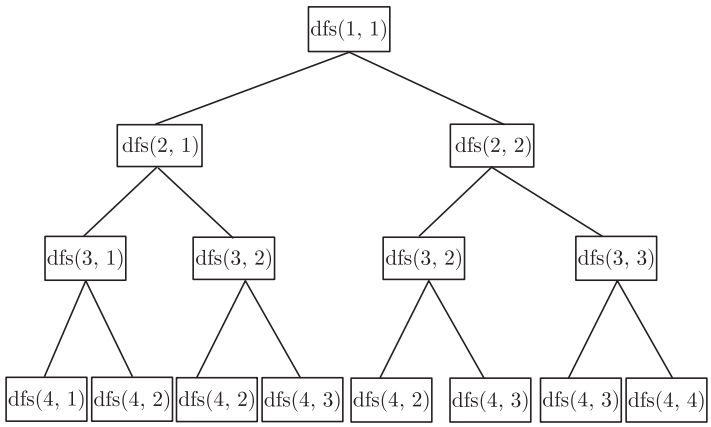
$$\mathrm{dfs}(i,j)=\begin{cases}\max(\mathrm{dfs}(i+1,j),\mathrm{dfs}(i+1,j+1))+a[i][j] & i<n\\ a[i][j] & i=n.\end{cases}$$

参考代码 6-1-1　【时间复杂度为 $O(2^n)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int n , a[N][N];
int dfs(int i , int j){
   if(i == n) return a[i][j]; // 走到底部无法再走了，直接返回
   return max(dfs(i + 1 , j) , dfs(i + 1 , j + 1)) + a[i][j];
}
signed main(){
   cin >> n;
   for(int i = 1 ; i <= n ; i ++)
     for(int j = 1 ; j <= i ; j ++)
       cin >> a[i][j];
   cout << dfs(1 , 1) << '\n';
   return 0;
}
```

在上方代码中，每次 dfs 都几乎调用了自己两次，所以代码的时间复杂度约为 $O(2^n)$。

如果一个三角形的行数为 4，则程序的递归过程将如下图所示。



显然，$O(2^n)$ 的时间复杂度是十分低效的，无法帮助我们顺利解出本题。至于复杂度低效的原因，从上图中不难发现是因为在递归的过程中做了过多重复的计算，比如在上图中，$\text{dfs}(3,2)$ 就重复计算了一次。

那么，我们要如何避免递归时的重复计算呢?

可以采用一个简单且高效的方法——记忆化，即定义一个二维数组 res[][]，用 res$[i][j]$ 保存 $\text{dfs}(i,j)$ 的计算结果。当再次需要 $\text{dfs}(i,j)$ 的计算结果时，直接返回 res$[i][j]$ 即可，无须继续递归下去。

使用了记忆化的方法优化后，最多只需进行约 n^2 次递归。

参考代码 6-1-2　【时间复杂度为 $O(n^2)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int n , a[N][N] , res[N][N];
int dfs(int i , int j){
    if(res[i][j]) return res[i][j]; // res[i][j] != 0 说明其记录着 dfs(i, j) 的值，返回从下往上找到达（i, j）时的最大值
    if(i == n) return a[i][j];
    return res[i][j] = max(dfs(i + 1 , j) , dfs(i + 1 , j + 1)) + a[i][j];
}
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++)
        for(int j = 1 ; j <= i ; j ++)
            cin >> a[i][j];
    cout << dfs(1 , 1) << '\n';
    return 0;
}
```

解决子问题 1：动态规划模式

与 DFS 不同的是，动态规划通常是采用递推的方式从上到下来求解问题。

在本题中，我们从三角形的顶部 $(1,1)$ 走到三角形的某个位置 (i,j) 的方法颇多，根据不同的走法，所得到的路径和可能会有所差异。

由于本题需要求解的是从三角形顶部 $(1,1)$ 走到底部的最大路径和，所以在不同方法带来的不同路径和中，我们只需要关注最大路径和即可。

于是，我们可以定义一个数组 dp[][]，其中 dp$[i][j]$ 用以表示从三角形顶部 (11) 走到 (ij) 的所有路径和中的最大值。

当走到底部时，所处的位置可能有 $(n,1)$、$(n,2)$、…、(n,n)，它们对应的最大路径和分别为 dp$[n][1]$、dp$[n][2]$、…、dp$[n][n]$。我们要的是这当中的最大路径和，即 $\max\limits_{j=1}^{n}$ dp$[n][j]$。

起初，处于三角形的顶部 $(1,1)$。因为从 $(1,1)$ 走到 $(1,1)$ 的方法仅有一种，所以我们可得 dp$[1][1] = a[1][1]$。

在有了 dp$[1][1]$ 的值后，我们便可开始从上到下求解所有 dp$[i][j]$。

对于位置 (i,j)，我们只有以下两种到达方法。

（1）从位置 $(i-1,j)$ 向下走一步。

（2）从位置 $(i-1,j-1)$ 向右下走一步。

无论我们选择哪种，从 $(1,1)$ 到 (i,j) 的路径与 $(1,1)$ 到 $(i-1,j)$ 或 $(i-1,j-1)$ 的路径都只会有一步之差。所以两种方法到达 (i,j) 的最大路径和分别如下。

（1）dp$[i-1][j] + a[i][j]$。

（2）dp$[i-1][j-1] + a[i][j]$。

为了使到达位置 (i,j) 的路径和最大，我们需要从这两种方法中选择较大的一种，即

$$\text{dp}[i][j] = \max(\text{dp}[i-1][j], \text{dp}[i-1][j-1]) + a[i][j].$$

这样，我们便完成了 dp 方程的转移。

参考代码 6-1-3　【时间复杂度为 $O(n^2)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int n , a[N][N] , dp[N][N];
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++)
        for(int j = 1 ; j <= i ; j ++)
            cin >> a[i][j];
    dp[1][1] = a[1][1];
    for(int i = 2 ; i <= n ; i ++)
        for(int j = 1 ; j <= i ; j ++)
```

```
            dp[i][j] = max(dp[i - 1][j] , dp[i - 1][j - 1]) + a[i][j];
    int ans = 0;
    for(int j = 1 ; j <= n ; j ++) ans = max(ans , dp[n][j]);
    cout << ans << '\n';
    return 0;
}
```

提示

以上情况都是在不考虑“向左下走的次数与向右下走的次数相差不能超过 1”这个限制条件下分析的，接下来我们来思考加上向左下走的次数与向右下走的次数相差不能超过 1 这个限制条件之后该如何处理，即解决子问题 2。

为了方便读者理解，接下来我们分别用动态规划和 DFS 两种方法对本题进行讲解。

解决子问题 2：动态规划模式

我们可以采用最简单的方法：从顶部向底部走的过程中，额外添加一个状态 —— 向下走的次数，即定义 $\mathrm{dp}[i][j][k]$ 表示从 $(1,1)$ 走到 (i,j) 一共向下走了 k 次的最大和。

根据向下走和向右下走的次数相差不能超过 1 的条件，那么从第 1 行到第 n 行一共要走 $n-1$ 步，由此可得：

- 当 $n-1$ 为奇数（n 为偶数）时，向下走的次数可以为 $\dfrac{n-1}{2}$，也可以为 $\dfrac{n-1}{2}+1$；
- 当 $n-1$ 为偶数（n 为奇数）时，向下走的次数只能为 $\dfrac{n-1}{2}$。

那么答案就可表示为

$$\mathrm{ans} = \begin{cases} \max\limits_{j=1}^{n}\left(\mathrm{dp}[n][j]\left[\dfrac{n-1}{2}\right], \mathrm{dp}[n][j]\left[\dfrac{n-1}{2}+1\right]\right) & n\%2==0 \\ \max\limits_{j=1}^{n}\left(\mathrm{dp}[n][j]\left[\dfrac{n-1}{2}\right]\right) & n\%2==1. \end{cases}$$

对于位置 (i,j)，它可以由位置 $(i-1,j)$ 向下走了一步得到，也可以由位置 $(i-1,j-1)$ 向右下走了一步得到，于是 dp 转移方程为

$$\mathrm{dp}[i][j][k] = \max(\mathrm{dp}[i-1][j][k-1], \mathrm{dp}[i-1][j-1][k]) + a[i][j].$$

参考代码 6-1-4　【时间复杂度为 $O(n^3)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int n , a[N][N] , dp[N][N][N];
signed main(){
    memset(dp , -0x3f , sizeof(dp)); // 某些情况可能并不存在，如 dp[2][1][0]：走到位置
        (2,1) 时共向下走了 0 次。 为了防止这种情况被转移，需初始化 dp 数组
```

```
    cin >> n;
    for(int i = 1 ; i <= n ; i ++)
        for(int j = 1 ; j <= i ; j ++)
            cin >> a[i][j];
    dp[1][1][0] = a[1][1];
    for(int i = 2 ; i <= n ; i ++){
        for(int j = 1 ; j <= i ; j ++){
            for(int k = 0 ; k <= (n - 1) ; k ++){
                if(!k) dp[i][j][k] = dp[i - 1][j - 1][k] + a[i][j];
                else dp[i][j][k] = max(dp[i - 1][j - 1][k], dp[i - 1][j][k - 1])+a[i][j];
            }
        }
    }
    int ma = 0;
    if((n - 1) & 1) for(int j = 1 ; j <= n ; j ++) ma = max(ma , max(dp[n][j][(n - 1) /
        2], dp[n][j][(n - 1) / 2 + 1]));
    else for(int j = 1 ; j <= n ; j ++) ma = max(ma , dp[n][j][(n - 1) / 2]);
    cout << ma << '\n';
    return 0;
}
```

虽说运用上述的方法已经可以完成本题了，但我们其实不额外添加一个状态也可以解决。

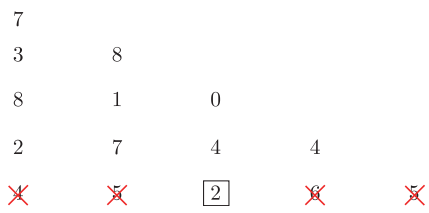
由上可得以下结论。

- 当 $n-1$ 为奇数（n 为偶数）时，向下走的次数可以为 $\frac{n-1}{2}$ 次，也可以为 $\frac{n-1}{2}+1$ 次。
- 当 $n-1$ 为偶数（n 为奇数）时，向下走的次数只能为 $\frac{n-1}{2}$。

基于向下走的次数、向右下走的次数限制不难发现以下情况。

- 当 $n-1$ 为奇数（n 为偶数）时，无论中间的路线是什么样的，最后的位置只有两种可能：$\left(n, 1+\frac{n-1}{2}\right)$、$\left(n, 1+\frac{n-1}{2}+1\right)$。
- 当 n 为偶数时，无论中间的路线是什么样的，最后的位置只有一种可能：$\left(n, 1+\frac{n-1}{2}\right)$。

所以我们并不需要考虑从顶部走到底部时向下走的次数，只需要保证最后的位置正确即可。



用 dp[i][j] 表示从 $(1,1)$ 走到 (i,j) 的最大和并对其进行转移，那么答案就可以表示为

$$\text{ans} = \begin{cases} \max\limits_{j=1}^{n}\left(\text{dp}[n]\left[\dfrac{n-1}{2}\right], \text{dp}[n]\left[\dfrac{n-1}{2}+1\right]\right) & n\%2 == 0 \\ \max\limits_{j=1}^{n}\left(\text{dp}[n]\left[\dfrac{n-1}{2}\right]\right) & n\%2 == 1. \end{cases}$$

参考代码 6-1-5 【时间复杂度为 $O(n^2)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int n , a[N][N] , dp[N][N];
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++){
        for(int j = 1 ; j <= i ; j ++){
            cin >> a[i][j];
        }
    }
    dp[1][1] = a[1][1];
    for(int i = 2 ; i <= n ; i ++){
        for(int j = 1 ; j <= i ; j ++){
            dp[i][j] = max(dp[i - 1][j - 1] , dp[i - 1][j]) + a[i][j];
        }
    }
    // 根据 n - 1 的奇偶性输出：在满足向下走的次数与向右下走的次数相差不能超过 1 的条件下可能
        会到达的位置对应的 dp 值
    if((n - 1) & 1) cout << max(dp[n][1 + (n-1)/2], dp[n][1 + (n - 1) / 2 + 1]) << '\n';
    else cout << dp[n][1 + (n - 1) / 2] << '\n';
    return 0;
}
```

解决子问题 2：DFS 模式

DFS 的处理方法和动态规划的类似，有以下两种方法：

（1）额外添加一个状态使得答案满足条件；

（2）保证最后的位置正确使得答案满足条件。

下面给出第二种方法的参考代码。

参考代码 6-1-6

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
```

```cpp
int n , a[N][N] , res[N][N];
int dfs(int i , int j){
    if(res[i][j]) return res[i][j];
    if(i == n) {
        if(n % 2 && j == n / 2 + 1) return a[i][j];
        if(n % 2 == 0 && (j == n / 2 || j == n / 2 + 1)) return a[i][j];
        return -10000000; // 位置不正确时，返回一个极大的负数
    }
    return res[i][j] = max(dfs(i + 1 , j) , dfs(i + 1 , j + 1)) + a[i][j];
}
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++)
        for(int j = 1 ; j <= i ; j ++)
            cin >> a[i][j];
    cout << dfs(1 , 1) << '\n';
    return 0;
}
```

6.2　砝码称重 (★★★)

题目信息

2021 年省赛 - 程序设计题

- ✧ C/C++ A 组第 6 题；C/C++ B 组第 7 题；C/C++ C 组第 7 题
- ✧ Java A 组第 6 题；Java B 组第 7 题；Java C 组第 7 题

【问题描述】

你有一架天平和 N 个砝码，这 N 个砝码重量依次是 $W_1, W_2, \ldots, W_N$。

请你计算一共可以称出多少种不同的重量。注意砝码可以放在天平两边。

【输入格式】

输入的第一行包含一个整数 N。

第二行包含 N 个整数：$W_1, W_2, W_3, \ldots, W_N$。

【输出格式】

输出一个整数表示答案。

【样例输入】

3

1 4 6

【样例输出】

10

【样例说明】

能称出的 10 种重量是 123456791011。

$1 = 1$

$2 = 6 - 4$(天平一边放 6，另一边放 4)

$3 = 4 - 1$

$4 = 4$

$5 = 6 - 1$

$6 = 6$

$7 = 1 + 6$

$9 = 4 + 6 - 1$

$10 = 4 + 6$

$11 = 1 + 4 + 6$

【评测用例规模与规定】

对于 50% 的评测用例，$1 \leqslant N \leqslant 15$。

对于所有评测用例，$1 \leqslant N \leqslant 100$，$N$ 个砝码总重不超过 100000。

题目分析

核心考点

- ✧ 思维分析；DFS
- ✧ 动态规划；背包问题
- ✧ 偏移量

本题是道有限制的选择问题、背包问题的变形题。

在本题中，题目给定了 1 个天平及 n 个砝码，每个砝码都有自己的重量。天平存在以下 3 种状态。

（1）平衡。

（2）向左倾斜。

（3）向右倾斜。

这 3 种状态分别可称出的重量：

（1）0（忽略不计）；

（2）左侧的砝码重量 − 右侧的砝码重量；

（3）右侧的砝码重量 − 左侧的砝码重量。

显然，天平的状态只会受到两侧砝码的重量影响。对于每个砝码，它都有以下 3 种处理方式。

（1）放在天平的左侧。

（2）放在天平的右侧。

（3）两侧都不放。

那么，n 个砝码就有 3^n 种处理方式（情况）。

在 50% 的评测用例中，$1 \leqslant n \leqslant 15$。当 $n = 15$ 时，$3^{15} = 14348907$，数据规模为 10^7 左右。

因此，我们可以使用直接 dfs “暴力”搜索出放置 n 个砝码的所有情况，并统计这些情况能够称出的不同重量的个数。

参考代码 6-2-1　【时间复杂度为 $O(3^n)$】

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10 , M = 1e5 + 10;
int n , ans , w[N] , vis[M]; // vis[x] = 1 表示可以称出重量 x
void dfs(int i , int left , int right){
    if(i > n){
        vis[max(left , right) - min(left , right)] = 1;
        return ;
    }
    // 将第 i 个砝码放置在左边
    dfs(i + 1 , left + w[i] , right);
    // 将第 i 个砝码放置在右边
    dfs(i + 1 , left , right + w[i]);
    // 两边都不放
    dfs(i + 1 , left , right);
}
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++) cin >> w[i];
    dfs(1 , 0 , 0);
    for(int i = 1 ; i < M ; i ++) if(vis[i]) ans ++ ;
    cout << ans << '\n';
    return 0;
}
```

可当 $1 \leqslant n \leqslant 100$ 时，还能用 dfs 搜索出所有情况吗？显然不行，因为当 $n = 100$ 时，3^{100} 会是个天文数字，计算机在有限的时间内是不可能搜索出所有情况的。

那怎么办呢？

考虑用动态规划解决。

1. 设计状态数组

按照常规的步骤，我们会设计一个状态数组 dp[]，其中 dp[i] 表示用前 i 个砝码所能称出的不同重量的个数。

这么设计看似合理，答案也可以用 dp[n] 轻松表示，但略加思考后不难发现其存在一个致命问题，即在状态转移的过程中，无法处理相同的重量被重复计算的情况。

如何解决这个致命问题呢? 我们只要将重量也设计在状态数组中，即设计一个 boolean 类型的二维数组 dp[][]，其中 dp[i][j] = true 表示前 i 个砝码能通过天平称出重量 j，dp[i][j] = false 则表示前 i 个砝码不能通过天平称出重量 j。

由于 n 个砝码的总重量不超过 100000，所以我们只要在求解完整个数组后，枚举 dp[n][1 ∼ 100000]，统计其中值为 true 的元素个数，即可得出 n 个砝码所能称出的不同重量的个数。

到这里貌似没有什么问题。那么接下来，我们就来讨论一下如何求解 dp[][] 数组。

2. 初始状态

起初，我们未在天平两侧放置任何砝码，天平它会处于一种平衡的状态。我们可以认为此时天平称的重量为 0，即 dp[0][0] = true。

3. 推导转移方程

事实上，天平所称出的重量总是会由重的一侧减去轻的一侧，即对于第 i 个砝码，若我们将其放入天平较重的一侧，则它将使天平称出的重量增加 w_i；若将该砝码放入天平较轻的一侧，则它将使天平称出的重量减少 w_i。例如，在我们处理完前 $i-1$ 个砝码后，天平所称出的重量为 j，那么在处理完第 i 个砝码后，天平所能称出的重量将会根据对第 i 个砝码不同的处理方式得到 3 种不同结果分别如下。

（1）$j + w_i$：将第 i 个砝码放在较重的一侧。

（2）$j - w_i$：将第 i 个砝码放在较轻的一侧。

（3）j：两侧都不放。

因此，我们可推导出以下 3 种状态转移式：

$$(\text{dp}[i-1][j] = \text{ true }) \rightarrow \begin{cases} \text{dp}[i][j+w[i]] = \text{ true} \\ \text{dp}[i][j-w[i]] = \text{ true} \\ \text{dp}[i][j] = \text{ true }. \end{cases}$$

若将 3 个状态转移式整合成一个，可得

$$\text{dp}[i][j] = \text{dp}[i-1][j-w[i]] \mid \text{dp}[i-1][j+w[i]] \mid \text{dp}[i-1][j].$$

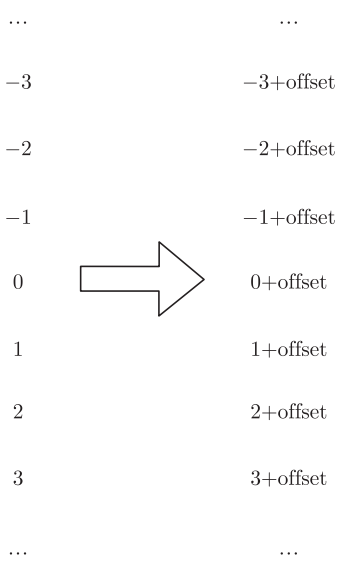
完成状态转移式的推导。

值得注意的是，在进行状态转移的过程中，可能会出现 $j < w[i]$ 的情况（$j - w[i] < 0$）。若不进行处理，就会导致数组的越界，进而导致答案错误。

因此，我们可以为所有重量添加一个偏移量 offset（对于重量 x，用 $x+\text{offset}$ 来表示它），如下图所示，使得 $\forall j - w[i] + \text{offset} \geqslant 0$。

提示

添加上 offset 后，dp$[i][j]$ 的含义为前 i 个砝码能否通过天平称出重量 $j - \text{offset}$。



参考代码 6-2-2　【时间复杂度为 $O(n \times \sum_{i=1}^{n} w_i)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10 , M = 1e5 + 10 , offset = 1e5;
int n , ans , w[N] , dp[N][2 * M];
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++) cin >> w[i];
    dp[0][0 + offset] = true;
    for(int i = 1 ; i <= n ; i ++){
        for(int j = 0 ; j < M + offset ; j ++){
            if(j - w[i] >= 0) dp[i][j] |= dp[i - 1][j - w[i]];
            dp[i][j] |= dp[i - 1][j + w[i]] | dp[i - 1][j];
        }
    }
    for(int i = 1 + offset ; i < M + offset ; i ++) if(dp[n][i]) ans ++ ;
    cout << ans << '\n';
    return 0;
}
```

6.3 括号序列 (★★★★)

题目信息

2021 年省赛 - 程序设计题

- ✧ C/C++ A 组第 9 题；C/C++ B 组第 10 题；C/C++ C 组第 10 题
- ✧ Java B 组第 10 题
- ✧ Python 组第 10 题

【问题描述】

给定一个括号序列，要求尽可能少地添加若干括号使得括号序列变得合法，当添加完成后，会产生不同的添加结果，请问有多少种本质不同的添加结果。

两个结果是本质不同的是指存在某个位置一个结果是左括号，而另一个是右括号。

例如，对于括号序列 (((，只需要添加两个括号就能让其合法，有以下几种不同的添加结果：()()()、()(())、(())()、(()()) 和 ((())。

【输入格式】

输入一行包含一个字符串 s，表示给定的括号序列，序列中只有左括号和右括号。

【输出格式】

输出一个整数表示答案，答案可能很大，请输出答案除以 1000000007 (即 10^9+7) 的余数。

【样例输入】

(((

【样例输出】

5

【评测用例规模与规定】

对于 40% 的评测用例，$|s| \leqslant 200$。

对于所有评测用例，$1 \leqslant |s| \leqslant 5000$。

题目分析

核心考点

- ✧ 思维分析
- ✧ DFS
- ✧ 动态规划

在开始解题之前，我们先分析两个问题。

问题 1. 什么情况下必须要添加左括号，什么情况下必须添加右括号?

问题 2. 我们添加的左括号会和我们添加的右括号匹配吗?

对于问题 1，我们可从以下两方面作进一步思考：

（1）当一个右括号之前已经没有可以与其相匹配的左括号时，我们必须要添加左括号。例如 ( ))，从**左往右**看，第二个右括号之前已经没有左括号可以与其匹配（第一个左括号和第一个右括号匹配），这时候我们就必须添加左括号。

（2）当一个左括号之后已经没有可以与其相匹配的右括号时，我们必须要添加右括号。例如 (( )，从**右往左**看，第二个左括号之后已经没有右括号可以与其匹配（第一个右括号和第一个左括号匹配），这时候我们就必须添加右括号。

对于问题 2，答案是“不会”。

我们可以使用反证法来证明：

（1）假设**操作完成后**的括号序列为 XXX (XXX) XXX，(、) 为我们添加的互相匹配的左括号和右括号，X 表示原序列的括号，或是我们添加的与原序列括号相匹配的括号。

（2）那么该序列 XXX (XXX) XXX 若是一个合法序列，则将(、)去掉后的序列 XXX XXX XXX 也会是个合法序列（对于一个合法的括号序列，去掉任意一对匹配的左右括号，它也依旧合法）。

（3）所以如果我们添加的左右括号相互匹配，它们将起不到什么作用，只会徒然增长序列的长度。而题目要求我们的序列长度尽可能短，所以我们添加的左括号不会和我们添加的右括号匹配（即我们**添加的左括号**只会受**原序列中的括号**影响，**添加的右括号**只会受**原序列中的括号**影响）。

因此，我们可以将**左括号的添加方案**和**右括号的添加方案**分开计算，然后再将其合并为**答案**（因为左括号和右括号的添加是互不影响的、独立的，所以**答案 = 左括号的添加方案数 × 右括号的添加方案数**）。

下面分别给出 3 种解题方式, 分别是暴力做法、能拿 40% 分数的做法、满分做法。大家仔细体验这 3 种解题方式的特点, 其中“能拿 40% 分数的做法”给出的解题方式, 虽然拿不了满分, 但在实际竞赛中, 也可以视情况使用。

方式 1：暴力做法

本题作为 2021 年蓝桥杯省赛压轴题之一，相信绝大部分人在做的时候都会一头雾水，甚至在已知上面信息的情况下，也还是会找不到切入点。

不过没有关系，蓝桥杯毕竟是个根据通过的数据量来给分的比赛，所以我们并不需要在第一步就思考出它的满分做法。

那第一步该做什么呢? 暴力!

有一个做题技巧：在面对一道完全没有头绪的题目时，请不要傻等着浪费时间，可以先

尝试用暴力的方法解题，然后再在暴力的基础上思考如何优化复杂度。这样不仅能帮助梳理题目的约束，同时也可以发现题目的隐含性质。

前面我们分析出**添加的左右括号**是互不影响的，所以接下来让我们先来尝试下如何用暴力求解的**左括号的添加方案数**。

首先确定一种暴力方法，比如在每个右括号前添加若干个左括号从而产生若干个新的括号序列，再从中计算最短的合法括号序列的个数。

假设原序列中有 cnt 个右括号，以这 cnt 个右括号为**分界线**，一共可以划分出 cnt + 1 个区域：区域 1) 区域 2) 区域 3) ...) 区域 cnt+1。显然我们只能在第 1 ∼ cnt 个区域添加左括号，因为若是在第 cnt + 1 区域添加左括号，将不会有右括号可以与它匹配。

那每个区域要添加多少个左括号呢? 加多少个左括号都可以。因此我们可以枚举每个区域要添加的括号数。

不过既然要枚举，就得设置一个枚举上限。很显然，我们**添加的左括号数**不可能大于**原序列中的右括号数**，所以可以设置每个区域的上限为原序列中的右括号数 cnt。

设置完上限后，我们就可以从第一个区域开始依次枚举要添加的左括号数。

当第 cnt 个区域枚举结束后，将会得到一个全新的括号序列。根据枚举的不同，我们也将得到多个全新的、不同的括号序列。我们再从这些新的括号序列中计算长度最短的、合法的序列个数，即可得到答案。

可问题又来了，如何判断一个括号序列是否合法呢?

这个简单。我们添加左括号的目的是为了让**原序列中的每个右括号**之前都有对应的左括号与其相匹配，所以只要新的括号序列中的每个右括号之前有对应的左括号可以与其相匹配即为合法序列。

上述整个过程可以使用 DFS 来实现。

参考代码 6-3-1　【时间复杂度为 $O(\text{len} \times \text{cnt}^{\text{cnt}})$】

```
#include<bits/stdc++.h>
using namespace std;
int n , cnt , ans , min_len = 0x3f3f3f3f;
string s;
bool check(string s){
    int left = 0;
    for(int i = 0 ; i < s.size() ; i ++){
        if(s[i] == '(') left ++ ; // 未匹配的左括号个数+1
        else left -- ; // 拿走一个左括号与该右括号匹配，左括号个数-1
        if(left < 0) return false; // 该右括号之前没有左括号可以与其匹配
    }
    return true;
}
void dfs(int p , string new_s){
    if(p == n){ // 枚举结束
```

```
        if(check(new_s) == true){ // 判断这个新的括号序列是否合法
            if(new_s.size() == min_len) ans ++ ; // 如果长度和最小长度相同，则答案个数+1
            else if(new_s.size() < min_len) min_len = new_s.size() , ans = 1; // 如果长度比最小长度还短，则作为新的最小长度
        }
        return ;
    }
    if(s[p] == '(') dfs(p + 1 , new_s + '(');
    else{
        for(int i = 0 ; i <= cnt ; i ++){ // 如果是右括号,则在该右括号前枚举要添加的左括号个数i
            string add = "";
            for(int j = 1 ; j <= i ; j ++) add += '(';
            dfs(p + 1 , new_s + add + ')');
        }
    }
}
signed main(){
    cin >> s;
    n = s.size();
    for(int i = 0 ; i < s.size(); i ++) if(s[i] == ')') cnt ++ ;
    dfs(0 , "");
    cout << ans << '\n';
    return 0;
}
```

以上的暴力做法对于第 1 ∼ cnt 个区域都从 0 ∼ cnt 枚举了要添加的左括号个数，这无疑产生了大量不合法的括号序列，那要如何减少不合法的括号数呢？可以从每个区域的枚举上限入手。

我们令每个区域的枚举上限都为 cnt 的理由是 “**添加的左括号数**不可能大于**原序列中的右括号数**”。这里**添加的左括号数**指的是所有区域的总添加数，而不是单一区域的添加数。

因此，我们可以设置一个动态的枚举上限 total 来简单优化一下枚举的次数（详见下方代码）。

参考代码 6-3-2

```
void dfs(int p , string new_s , int total){ // total 表示还可以添加的左括号数。在开始 DFS 之前，total = cnt
    if(p == n){ // 枚举结束
        if(check(new_s) == true){ // 判断这个新的括号序列是否合法
            if(new_s.size() == min_len) ans ++ ; // 如果长度和最小长度相同，则答案个数+1
            else if(new_s.size() < min_len) min_len = new_s.size() , ans = 1; // 如果长度比最小长度还短，则作为新的最小长度
```

```
6         }
7         return ;
8     }
9     if(s[p] == '(') dfs(p + 1 , new_s + '(' , total);
10    else{
11        for(int i = 0 ; i <= total ; i ++){ // 如果是右括号,则在该右括号前枚举要添加的左括号
              个数i
12            string add = "";
13            for(int j = 1 ; j <= i ; j ++) add += '(';
14            dfs(p + 1 , new_s + add + ')' , total - i); // 当前区域添加了 i 个左括号，那后面
                  的区域可以添加的左括号总数为 total - i
15        }
16    }
17 }
```

与优化前相比，虽然上述方法能一定程度上减少一些不合法的括号序列，但产生的不合法括号序列依旧很多，这并不能起到很有效的作用。

那还能进行哪些优化呢？我们重新思考这句话：**添加的左括号数**不可能大于**原序列中的右括号数**。既然如此，那到底要添加多少左括号呢？

回顾一下，我们添加左括号的目的是让**原序列中的每个右括号**的前面都有对应的左括号与其相匹配。也就是说，只有在原序列中的某个右括号之前没有可以与其相匹配的左括号时，我们才有必要添加左括号。

例如，对于括号序列) (( )))，它的第 1 个右括号前没有可以与其相匹配的左括号，所以我们至少要添加 1 个左括号；第 2 ∼ 3 个右括号前都有左括号可以与其相匹配，所以至少要添加的左括号个数为 0；第 4 个右括号前没有可以与其相匹配的左括号，所以我们至少要添加 1 个左括号。

有了这个条件，我们就可以制定贪心策略：**只有当必须要添加左括号时才添加左括号**，这样就可以保证我们添加的括号总数最小（记添加的最小总数为 need），相关代码参考完整代码的第 26∼31 行。

这样我们就可以将枚举上限设置为 need（need ⩽ cnt），从而减少不合法括号序列的产生。不过很显然这优化力度还远远不够，甚至在最坏的情况下 (need = cnt) 起不到任何作用。

那还有什么办法可以优化呢？能不能一次性避免所有不合法括号序列的产生呢？

答案自然是“能”。只要保证产生的括号序列都是合法的即可避免不合法括号序列的产生。

那如何保证产生的括号序列都是合法的呢？只要保证**每个右括号**之前都有可以与之匹配的左括号即可。

于是我们就可以在 DFS 过程中记录未匹配的左括号个数，当某个右括号之前未匹配的左括号个数为 0 时，我们就必须添加左括号（从 1 开始枚举添加的左括号个数）。这样就能

保证产生的括号序列都是合法的，也就不需要用 check() 函数来判断序列的合法性。完整代码如下。

参考代码 6-3-3

```
#include<bits/stdc++.h>
using namespace std;
int n , cnt , ans , min_len = 0x3f3f3f3f;
string s;
void dfs(int p , string new_s , int total , int left){
    if(p == n){ // 枚举结束
        if(new_s.size() == min_len) ans ++ ; // 如果长度和最小长度相同，则答案个数+1
        else if(new_s.size() < min_len) min_len = new_s.size() , ans = 1; // 如果长度比最小长度还短，则作为新的最小长度
        return ;
    }
    if(s[p] == '(') dfs(p + 1 , new_s + '(' , total , left + 1); // 未匹配的左括号个数+1
    else{
        int st = 0; // 枚举起点
        if(left == 0) st = 1; // 该右括号之前未匹配的左括号个数为0，那么为了保证序列合法，它之前至少要添加一个左括号
        for(int i = st ; i <= total ; i ++){ // 如果是右括号，则在该右括号前枚举要添加的左括号个数i，从st开始枚举
            string add = "";
            for(int j = 1 ; j <= i ; j ++) add += '(';
            dfs(p + 1 , new_s + add + ')' , total - i , left + i - 1); // 需要从未匹配的左括号中拿出一个和当前右括号匹配，所以要-1
        }
    }
}
signed main(){
    cin >> s;
    n = s.size();
    for(int i = 0 ; i < s.size(); i ++) if(s[i] == ')') cnt ++ ;
    int need = 0, left = 0;// need 表示最少需要添加的左括号数，left 表示未匹配的左括号个数
    for(int i = 0 ; i < s.size() ; i ++){
        if(s[i] == '(') left ++ ; // 未匹配的左括号个数 + 1
        else left --; // 拿走一个左括号与该右括号匹配，左括号个数-1
        if(left < 0) need ++, left = 0;// 该右括号之前没有左括号可以与其匹配，此时就必须添加左括号
    }
    dfs(0 , "" , need , 0);
    cout << ans << '\n';
    return 0;
```

```
35 }
```

以上就是暴力求解**左括号添加方案数**的过程，**右括号添加方案数**的求解类似，就不过多阐述。

暴力的优化到此结束。

可见，采用暴力解法虽然可以避免所有不合法括号序列的产生，但产生出的合法括号序列的数量也相当多，因此我们无法通过“暴力”来解决本题。

方式 2：能拿 40% 分数的做法

1. 求原括号序列中第 $i(1 \leqslant i \leqslant \text{cnt})$ 个右括号之前的左括号数

我们虽然求解了 need（最少需要添加的左括号数），但这个 need 是对于整个括号序列的。而对于原括号序列的某个右括号，我们并不知道它之前必须要添加多少个左括号。

于是我们可以定义原括号序列中第 i 个右括号之前的左括号数为 num[i]，那么第 i 个右括号之前需要**添加的左括号数**就等于 $i - \text{num}[i]$。

num[i] 的求解如下。

参考代码 6-3-4

```
1 int left = 0 , cnt = 0;
2 for(int i = 1 ; i <= n ; i ++){
3     if(s[i] == ')') num[++ cnt] = left;
4     if(s[i] == '(') left ++; // left 表示原序列区间[1, @$i$@]的左括号个数
5 }
```

2. 计算方案数

通过前面的步骤，我们将问题转换成了“往 cnt 个区域中添加 need 个左括号的合法方案数” 的求解。

假设现在有 i 个区域（右括号），一共往这个 i 个区域中添加了 j 个左括号，要想得出其中的合法方案数，该怎么计算呢?

我们可以先定义 dp[i][j] 来表示往前 i 个区域中共添加了 j 个左括号后的所有合法方案数，而往前 i 个区域中添加了 j 个左括号相当于往前 $i-1$ 个区域中添加了 k（$0 \leqslant k \leqslant j$）个左括号，并往第 i 个区域中添加了 $j-k$ 个左括号。由此，我们可以就可推出 dp[i][j] 的状态转移式。

提示

$\mathrm{dp}[i][j]$ 表示往前 i 个区域中添加了 j 个左括号的所有合法方案数，而 $\mathrm{dp}[i-1][0\sim j]$ 表示往前 $i-1$ 个区域中添加了 $0\sim j$ 个左括号的所有合法方案数。

$\mathrm{dp}[i][j]$ 和 $\mathrm{dp}[i-1][0\sim j]$ 的区别就在于 $\mathrm{dp}[i][j]$ 多了第 i 个区域，所以 $\mathrm{dp}[i][j]$ 可以由以下情况得到。

（1）$\mathrm{dp}[i-1][0]$ 往区域 i 添加了 j 个左括号得到。

（2）$\mathrm{dp}[i-1][1]$ 往区域 i 添加了 $j-1$ 个左括号得到。

（3）.....

（4）$\mathrm{dp}[i-1][j]$ 往区域 i 添加了 0 个左括号得到。

即

$$\mathrm{dp}[i][j]=\mathrm{dp}[i-1][0]+\mathrm{dp}[i-1][1]+\ldots\mathrm{dp}[i-1][j]$$

不过要想求解 $\mathrm{dp}[i][j]$ ，只有状态转移式还不够，我们还需确定其初状态（即确定 $i=1$ 时 $\mathrm{dp}[1][j]$ 对应的值）。

$i=1$ 时的特殊情况我们可以分为以下两部分讨论。

（1）$j\in(1\sim\mathrm{need})$。显然，前 1 个区域（右括号）添加了 1, 2, 3, ..., need 个左括号的方案数都为 1（need 表示最少需要添加的左括号个数），所以 $\mathrm{dp}[1][1\sim\mathrm{need}]=1$。

（2）$j=0$。这部分又可细分为两种情况。

- 第 1 个区域**包含**原序列的左括号，那么 $\mathrm{dp}[1][0]=1$，即前 1 个区域（右括号）添加了 0 个左括号的方案数为 1。
- 第 1 个区域**不包含**原序列的左括号，那么 $j=0$ 是不合法的，该情况不能作为一个方案（对于一个合法序列的任意前缀，左括号的个数一定要大于等于右括号），所以 $\mathrm{dp}[1][0]=0$ 。

提示

由于我们要的是合法序列，所以当右括号的个数为 i 时，它前面的左括号的个数必然大于等于 i，即添加的左括号数 j + 原序列的左括号数 $\mathrm{num}[i]\geqslant i$。

接下来，我们就可以完成 $\mathrm{dp}[i][j]$ 的求解。

参考代码 6-3-5

```
1  for(int i = 1 ; i <= need ; i ++) dp[1][i] = 1;
2  if(num[1] > 0) dp[1][0] = 1;
3
4  for(int i = 2 ; i <= cnt ; i ++){
5   // 1. j+num[i]>=i
```

```
 // 2.最少添加的数量为 need
   for(int j = i - num[i] ; j <= need ; j ++){
      for(int k = 0 ; k <= j ; k ++){
         dp[i][j] += dp[i - 1][k];
         dp[i][j] %= mod;
   }
  }
}
```

提示

答案可用 dp[cnt][need] 表示，即前 cnt 个区域添加了 need 个左括号的所有合法方案数。

至此，添加左括号的方案数就计算完成了。

对于右括号的添加方案，我们可以将原括号序列反转，并令 (为)、) 为 (，然后按照求左括号的方案数的方法再求一遍即可。

参考代码 6-3-6　【时间复杂度为 $O(N^3)$】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int N = 5e3 + 10, mod = 1e9 + 7;
string s;
int dp[N][N], num[N];
int calc(int need, bool flag)
{
   memset(dp, 0, sizeof(dp));
   memset(num, 0, sizeof(num));
   // flag = 1 表示统计添加左括号的方案数; flag = 0 表示统计添加右括号的方案数（翻转）
   if(!flag) {
      reverse(s.begin(), s.end());
      for(int i = 0 ; i < s.size() ; i ++) {
         if(s[i] == ')') s[i] = '(';
         else s[i] = ')';
      }
   }
   int left = 0, cnt = 0;
   for(int i = 0 ; i < s.size() ; i ++) {
      if(s[i] == ')') num[++ cnt] = left; // cnt 为右括号的编号
      if(s[i] == '(') left ++; // left 表示原序列区间[1,i]的左括号的个数
   }
   // 区域1 有左括号时 dp[1][0] = 1, 否则 dp[1][0] = 0
   if(num[1] > 0) dp[1][0] = 1;
```

```
    for(int i = 1 ; i <= cnt ; i ++) dp[1][i] = 1;
    for(int i = 2 ; i <= cnt ; i ++) {
        // 1. j+num[i]>=i
        // 2.最少添加的数量为 need
        for(int j = max(0ll , i - num[i]) ; j <= need ; j ++) {
            for(int k = 0 ; k <= j ; k ++) {
                dp[i][j] += dp[i - 1][k];
                dp[i][j] %= mod;
            }
        }
    }
    return dp[cnt][need];
}
signed main()
{
    ios::sync_with_stdio(false);
    cin.tie(0) , cout.tie(0);
    cin >> s;
    int need = 0, tmp = 0;// need 表示最少需要添加的左括号数，tmp表示序列前缀和
    for(int i = 0 ; i < s.size() ; i ++) {
        if(s[i] == '(') tmp ++ ;
        else tmp --;
        if(tmp < 0) need ++, tmp ++;// tmp < 0 表示右括号数大于左括号数，此时需要添加左括号
            使得 左括号数>=右括号数
    }
    int need2 = tmp;
    cout << calc(need, 1) * calc(need2, 0) % mod << '\n';
    return 0;
}
```

方式 3：满分做法

要想拿满分需要在“方式 2：能拿 40% 分数做法”的步骤上作进一步优化。

在上述步骤中，我们得到 $\mathrm{dp}[i][j] = \mathrm{dp}[i-1][0] + \mathrm{dp}[i-1][1] + \ldots + \mathrm{dp}[i-1][j]$。

如果接触过“完全背包”，那就应该联想到用完全背包的前缀和来优化：

已知

$$\begin{aligned}\mathrm{dp}[i][j] &= (\mathrm{dp}[i-1][0] + \mathrm{dp}[i-1][1] + \ldots + \mathrm{dp}[i-1][j] + \mathrm{dp}[i-1][j+1]) \\ &\quad - \mathrm{dp}[i-1][j+1] \\ &= \mathrm{dp}[i][j+1] - \mathrm{dp}[i-1][j+1]\end{aligned}$$

移项得

$$\begin{aligned}\mathrm{dp}[i][j+1] =& \mathrm{dp}[i][j]+\mathrm{dp}[i-1][j+1], j \in (i-\mathrm{num}[i], \mathrm{cnt}),\\ & j+1 \in (i-\mathrm{num}[i]-1, \mathrm{cnt}-1)\end{aligned}$$

再令 $j=j-1$，得

$$\mathrm{dp}[i][j]=\mathrm{dp}[i][j-1]+\mathrm{dp}[i-1][j]$$

这样就可以去掉一层循环，复杂度为 $O(N^2)$。

不过需要注意，这是前缀和优化，也就是要使 $\mathrm{dp}[i][j]=\mathrm{dp}[i-1][j]+\mathrm{dp}[i][j-1]$，须满足所有的 $\mathrm{dp}[i][j]=\sum_{k=0}^{j}\mathrm{dp}[i-1][k]$ 。

而我们在求解 $\mathrm{dp}[i][j]$ 时，有以下两种情况。

（1）当 $j \in [(i-\mathrm{num}[i]) \sim \mathrm{cnt}]$ 时，$\mathrm{dp}[i][j]=\sum_{k=0}^{j}\mathrm{dp}[i-1][k]$。

（2）当 $j \in [0, i-\mathrm{num}[i]-1]$ 时，$\mathrm{dp}[i][j]=0 \neq \sum_{k=0}^{j}\mathrm{dp}[i-1][k]$。

所以 $\mathrm{dp}[i][j]=\mathrm{dp}[i-1][j]+\mathrm{dp}[i][j-1]$ 的转移并不完全合法。

那怎么办呢？我们可以用类似的方法来优化。

定义 pre[] 数组。

- 当 $j \in [0 \sim (i-\mathrm{num}[i]-1)]$ 时，$\mathrm{pre}[j]=0$。
- 当 $j \in [(i-\mathrm{num}[i]) \sim \mathrm{need}]$ 时，$\mathrm{pre}[j]=\mathrm{pre}[j-1]+\mathrm{dp}[i-1][j]$。
- 那么当 $j \in [(i-\mathrm{num}[i]) \sim \mathrm{need}]$ 时，$\mathrm{dp}[i][j]=\mathrm{pre}[j]$。

这样时间复杂度就优化到了 $O(N^2)$。

参考代码 6-3-7　【时间复杂度为 $O(N^2)$】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int N = 5e3 + 10, mod = 1e9 + 7;
string s;
int dp[N][N], num[N], pre[N] , nex[N];
// 计算方案数
int calc(int need, bool flag)
{
    if(!need) return 1; // 不需要添加括号也是一种方案
    memset(dp, 0, sizeof(dp));
    memset(num, 0, sizeof(num));
    memset(pre , 0 , sizeof(pre));
    memset(nex , 0 , sizeof(nex));
    // flag = 1 表示统计添加左括号的方案数; flag = 0 表示统计添加右括号的方案数（翻转）
    if(!flag) {
```

```
17          reverse(s.begin(), s.end());
18          for(int i = 0 ; i < s.size() ; i ++) {
19              if(s[i] == ')') s[i] = '(';
20              else s[i] = ')';
21          }
22      }
23      int left = 0, cnt = 0;
24      for(int i = 0 ; i < s.size() ; i ++) {
25          if(s[i] == ')') num[++ cnt] = left; // cnt 为右括号的编号
26          if(s[i] == '(') left ++; // left 表示原序列区间[1,i]的左括号的个数
27      }
28      // 区域1 有左括号时 dp[1][0] = 1, 否则 dp[1][0] = 0
29      if(num[1] > 0) dp[1][0] = 1 , pre[0] = 1;
30      for(int i = 1 ; i <= cnt ; i ++) dp[1][i] = 1 , pre[i] = pre[i - 1] + 1;
31      for(int i = 2 ; i <= cnt ; i ++){
32          for(int j = 0 ; j <= need ; j ++) nex[j] = 0;
33          for(int j = max(0ll , i - num[i]) ; j <= need ; j ++){
34              dp[i][j] = pre[j];
35              if(j - 1 < 0) nex[j] = dp[i][j];
36              else nex[j] = (nex[j - 1] + dp[i][j]) % mod;
37          }
38          for(int j = 0 ; j <= cnt ; j ++) pre[j] = nex[j];// 先用 nex[] 存储 dp[i-1] 的信
                息，再将 nex[] 的值赋给 pre[]，这样下次循环 pre[] 存的值就是 dp[i-1] 的值
39      }
40      return dp[cnt][need];
41  }
42  signed main()
43  {
44      ios::sync_with_stdio(false);
45      cin.tie(0) , cout.tie(0);
46      cin >> s;
47      int need = 0, pre = 0;// need 表示最少需要添加的左括号数，pre表示序列前缀和
48      for(int i = 0 ; i < s.size() ; i ++) {
49          if(s[i] == '(') pre ++ ;
50          else pre --;
51          if(pre < 0) need ++, pre ++;// pre < 0 表示右括号数大于左括号数，此时需要添加左括号
                使得左括号数>=右括号数
52      }
53      int need2 = pre;
54      cout << calc(need, 1) * calc(need2, 0) % mod << '\n';
55      return 0;
56  }
```

6.4 异或三角 (★★★★)

题目信息

2021 年国赛 - 程序设计题

- ✧ C/C++ A 组第 9 题；C/C++ B 组第 10 题
- ✧ Java A 组第 9 题；Java B 组第 10 题

【问题描述】

给定 T 个数 $n_1, n_2, \ldots, n_T$，对每个 n_i 请求出有多少组 a, b, c 满足：

（1）$1 \leqslant a, b, c \leqslant n_i$；

（2）$a \oplus b \oplus c = 0$，其中 $\oplus$ 表示二进制按位异或；

（3）长度为 a, b, c 的三条边能组成一个三角形。

【输入格式】

输入的第一行包含一个整数 T。

接下来 T 行每行一个整数，分别表示 $n_1, n_2, \ldots, n_T$。

【输出格式】

输出 T 行，每行包含一个整数，表示对应的答案。

【样例输入】

```
2
6
114514
```

【样例输出】

```
6
11223848130
```

【评测用例规模与规定】

对于 10% 的评测用例，$T = 1, 1 \leqslant n_i \leqslant 200$；

对于 20% 的评测用例，$T = 1, 1 \leqslant n_i \leqslant 2000$；

对于 50% 的评测用例，$T = 1, 1 \leqslant n_i \leqslant 2^{20}$；

对于 60% 的评测用例，$1 \leqslant T \leqslant 100000, 1 \leqslant n_i \leqslant 2^{20}$；

对于所有评测用例，$1 \leqslant T \leqslant 100000, 1 \leqslant n_i \leqslant 2^{30}$。

题目分析

核心考点

- ✧ 思维分析；动态规划
- ✧ 位运算
- ✧ (二维) 数位 dp

题目对于我们要求解的 a, b, c 设定了两个十分重要的限制条件。

- 限制条件 1: $a \oplus b \oplus c = 0$。
- 限制条件 2: 长度为 a, b, c 的三条边能组成一个三角形。

这两个条件不难解读。

根据异或运算的归零律（相同数的异或值为 0）可得

- $(a \oplus b) = c$。
- $(b \oplus c) = a$。
- $(a \oplus c) = b$。

根据三角形的不等式定理（两边之和大于第三边）可得

- $a + b > c$。
- $a + c > b$。
- $b + c > a$。

解读完两个限制条件后，我们进入解题环节。

提示

本题包含多个得分点，不同得分点的分值 (得分比例) 不同。下面将根据不同的得分比例尝试不同的解法。

方式 1: 能拿 10% 分数的做法

对于 10% 分数做法的测试数据，$n \leqslant 200$，其数据范围是极小的。

因此，我们可以从“暴力”方向入手，直接使用 3 个循环分别枚举 a, b, c，并在整个枚举过程中判断并统计有多少组 a, b, c 满足题目的两个限制条件即可。

参考代码 6-4-1　【时间复杂度为 $O(Tn^3)$（$T = 1$）】

```
#include<bits/stdc++.h>
using namespace std;
signed main(){
    int T = 1;
    cin >> T;
    while(T --){
        int n , ans = 0;
```

```
        cin >> n;
        for(int a = 1 ; a <= n ; a ++)
            for(int b = 1 ; b <= n ; b ++)
                for(int c = 1 ; c <= n ; c ++)
                    if((a ^ b ^ c) == 0 && a + b > c && a + c > b && b + c > a)
                        ans ++ ;
        cout << ans << '\n';
    }
    return 0;
}
```

方式 2: 能拿 20% 分数的做法

当 n 的数据规模达到 2000 时，$n^3 = 2000^3 = 8 \times 10^9$。此时，对 3 个数进行暴力枚举的做法显然是不可行的。

但是，这并不要紧。

根据对限制条件 1 的解读，我们可得到 a, b, c 中的任意一个数的值都可由另外两个数异或得到。因此，对于 20% 分数做法的测试数据，我们还是可以轻松地通过枚举两个数（如 a, b）以得到第三个数（c）来判断并统计有多少组 a, b, c 满足题目的两个限制条件。

参考代码 6-4-2　【时间复杂度为 $O(Tn^2)$（$T = 1$）】

```
#include<bits/stdc++.h>
using namespace std;
signed main(){
    int T = 1;
    cin >> T;
    while(T --){
        int n , ans = 0;
        cin >> n;
        for(int a = 1 ; a <= n ; a ++)
            for(int b = 1 ; b <= n ; b ++){
                int c = (a ^ b); //a ^ b ^ c = 0 , 因为相同数的异或值为 0, 所以 c = (a ^ b)
                if(a + b > c && a + c > b && b + c > a)
                    ans ++ ;
            }
        cout << ans << '\n';
    }
    return 0;
}
```

方式 3：能拿 50% 分数的做法

当 n 的数据规模达到 2^{20} 时，我们最多就只能从 a,b,c 3 个数中选择一个数进行枚举，并且选择的是 3 个数中的最大数。

提示

在两边之和大于第三边这个条件当中，如果第三边是其中值最大的一条边，那么这 3 条边的关系一定满足任意两边之和大于第三边，即这 3 个数一定能构成三角形；但如果第三边不是其中值最大的一条边，那么就无法确定这 3 条边的关系是否满足任意两边之和大于第三边即这 3 个数不一定能构成三角形。

例如，假设 $a>b>c$。那么这 3 个数必然满足以下条件。

- $a+b>c$。
- $a+c>b$。

要想确定 a,b,c 能否组成三角形，只需要求出 $b+c$ 与 a（另外两个数与最大数）的大小关系即可。但如果求的是 $a+b>c$ 或 $a+c>b$，那么是无法确定 a,b,c 是否能组成三角形的。因此，我们应枚举 3 个数中的最大数。

可 a,b,c 3 个数中的最大数是 a、是 b，还是 c 呢?

简单来说，a,b,c 谁都可以是最大数。

例如，有 3 个数分别为 $3,1,1$，那么以下 3 种情况都满足条件。

（1）$a=3,b=1,c=1$。

（2）$a=1,b=3,c=1$。

（3）$a=1,b=1,c=3$。

为了方便处理，我们可以假设 a 作为最大值（b,c 大小关系不确定）的满足条件（$a\oplus b\oplus c=0$、$b+c>a$）的组数为 F。

由于 b 作为最大值、c 作为最大值和 a 作为最大值无异，因此 b, c 作为最大值的满足条件的组数也同样为 F，答案等于 $3\times F$ 。

可要如何求 F 呢?

我们一步一步来。

首先 a 的值是可以通过枚举确定的。而在确定了 a 的值后，我们要做的就是找出满足 $1\leqslant b,c\leqslant a$，$b+c>a$ 的 (b,c) 的个数。

怎么找呢?

自然要结合题目给定的限制条件来找。

由于限制条件涉及位运算（异或），并且我们要求解的是在一个范围内满足某条件的数的个数，因此不难想到利用二进制下的数位 dp 来确定这个数。

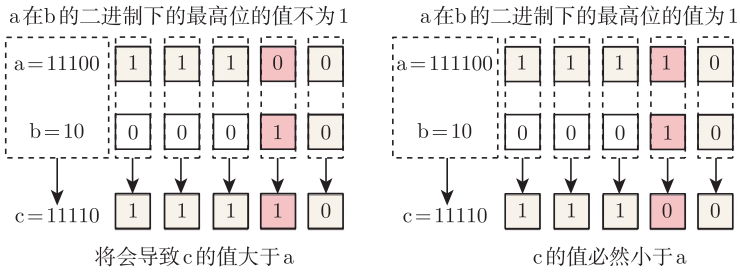
假定我们要确定的数为 b，那么在数位 dp 的过程中需要满足的条件如下。

（1）$b<a$：令数位 dp 的上界为 a。

（2）$c<a$：要使 $c<a$，则 a 在 b 的二进制下的最高位的值必须为 1，如下页图所示。

举例说明

> 若 $a = 11100$，则 b 可取的值有 $100, 1100, 1000, 101, 110, \ldots$，而不可取的值有 $1, 10, 11, \ldots$。



(3) $b + c > a = b \oplus c$：

要使 $b + c > b \oplus c$，则 b, c 在二进制下至少有一位的值都为 1。

举例说明

> 若 $b = 11100$，则 c 可取的值有 $100, 101, 110, 111, 1000 \ldots$，不可取的值有 $1, 10, 11, \ldots$。c 是我们通过 a, b 计算来的，所以要使 b, c 在二进制下至少有一位的值都为 1，a, b 就必然在某位上存在 a 的值为 0、b 的值为 1 的情况。

结合以上条件，我们可以定义 dp[len][limit][ok1][ok2]，其含义分别如下。

- len：当前处理到 b 数位上的第 len 位。
- limit：b 是否达到了上限。
- ok1：a 在 b 的二进制下的最高位是否为 1 。
- ok2：a, b 是否在某一位存在 $a = 0, b = 1$ 的情况。

根据以上定义和分析进行数位统计、转移，即可得到答案。

参考代码 6-4-3 【时间复杂度为 $O(2^{20}\log)$】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
int n , dp[21][2][2][2] , num[22];
int dfs(int len , bool limit , bool zero , bool ok1 , bool ok2){
    if(!len) return ok1 && ok2 ? 1 : 0;
    if(~dp[len][limit][ok1][ok2] && !zero) return dp[len][limit][ok1][ok2];
    int up = limit ? num[len] : 1 , res = 0;
    for(int i = 0 ; i <= up ; i ++){
      if(zero && i && !num[len]) continue ;
      res+= dfs(len - 1 , limit && i == up , zero && !i, ok1 || (zero && i && num[len]),
          ok2 || (i && !num[len]));
    }
```

```
    return dp[len][limit][ok1][ok2] = res;
}
int solve(int n){
    memset(dp , -1 , sizeof(dp));
    int len = 0;
    while(n) num[++ len] = n & 1 , n >>= 1;
    return dfs(len , 1 , 1 , 0 , 0);
}
signed main(){
    int T = 1;
    cin >> T;
    while(T --){
      cin >> n;
      int res = 0;
      for(int i = 1 ; i <= n ; i ++) res += solve(i);
      cout << res * 3 << '\n';
    }
    return 0;
}
```

方式 4: 能拿 60% 分数的做法

和 50% 分数做法的唯一区别在于，60% 分数做法的测试数据包含多组查询。不过对于每组查询，我们的步骤都是从 $1 \sim n$ 中枚举 a，然后再对 b 进行数位 dp 以求出对应的方案。

因此，我们可以进行一个预处理操作，即从 $1 \sim 2^{20}$ 枚举 a，枚举的同时对 b 进行数位 dp 得到对应的方案数，然后将每次枚举对应的方案数存储起来（第 i 次枚举存储的方案数即为 $n = i$ 时的答案）。

这样在多组查询的时候，就能通过 $O(1)$ 的时间复杂度得到答案。

参考代码 6-4-4　【时间复杂度为 $O(2^{20}\log + T)$】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
int n , dp[21][2][2][2] , num[22] , ans[1 << 21];
int dfs(int len , bool limit , bool zero , bool ok1 , bool ok2){
    if(!len) return ok1 && ok2 ? 1 : 0;
    if(~dp[len][limit][ok1][ok2]) return dp[len][limit][ok1][ok2];
    int up = limit ? num[len] : 1 , res = 0;
    for(int i = 0 ; i <= up ; i ++){
      if(zero && i && !num[len]) continue ;
      res += dfs(len - 1, limit && i == up , zero && !i, ok1 || (zero && i && num[len]),
          ok2 || (i && !num[len]));
```

```
    }
    return dp[len][limit][ok1][ok2] = res;
}
int solve(int n){
    memset(dp , -1 , sizeof(dp));
    int len = 0;
    while(n) num[++ len] = n & 1 , n >>= 1;
    return dfs(len , 1 , 1 , 0 , 0);
}
signed main(){
    int T = 1 , res = 0;
    // 预处理，将 n = i 的答案记录在 ans[i] 中
    for(int i = 1 ; i <= (1 << 20) ; i ++) res += solve(i) , ans[i] = res * 3;
    cin >> T;
    while(T --){
      cin >> n;
      cout << ans[n] << '\n';
    }
    return 0;
 }
```

方式 5：100% 满分做法

对于 50%、60% 分数做法的测试数据，因为 a 的最大范围为 $2^{20} = 1048576$，所以我们可以通过枚举来确定 a。但在满分做法的测试数据中，n 的最大范围为 2^{30} 次方，显然，枚举是不可能的了。

那怎么办呢?

回顾一下我们先前对于各个得分点的处理步骤。

（1）10% 分数：暴力枚举 3 个数并统计答案。

（2）20% 分数：无法枚举 3 个数，转换成枚举两个数，通过两个数的异或计算得到第三个数并统计答案。

（3）50% 分数、60% 分数：无法枚举两个数，转换成枚举一个数、数位 dp 一个数，通过两个数的异或计算得到第三个数并统计答案。

既然此前我们在无法枚举一个数时，使用数位 dp 来代替处理，那么现在无法枚举 a 时，是不是也能用数位 dp 来代替处理呢?

下面来尝试一下。

还是以 a 为最大值，先对其进行处理。对于 a，由于它是第一个要确定的数，因此我们仅须对它增设一个限制条件：$a < n$（将其上界定为 n 即可）。

接下来考虑 b （此时 a 已确定）。

- $b < a$：将 b 的上界设为 a。

- $c < a$：处理方式在 50% 分数的做法中已讨论过。
- $b + c > a = b \oplus c$：处理方式在 50% 分数的做法中已讨论过。

于是，我们就可以定义 dp[len][limit1][limit2][ok1][ok2]，其含义如下。

- len：当前进行到二进制下的第 len 位。
- limit1：a 是否达到上界 (limit1 = 0 表示未到达，反之表示达到)。
- limit2：b 是否达到上界 (limit2 = 0 表示未到达，反之表示达到)。
- ok1：是否满足 $c < a$（ok1 = 1 表示满足，反之表示未满足）。
- ok2：是否满足 $b + c > a = b \oplus c$（ok2 = 1 表示满足，反之表示未满足）。

其转移的核心参考如下代码。

参考代码 6-4-5

```
int up1 = limit1 ? num[len] : 1 , res = 0;
for(int i = 0 ; i <= up1 ; i ++){
   int up2 = limit2 ? i : 1;
   for(int j = 0 ; j <= up2 ; j ++){
      if(!ok1 && !i && j) continue ;
      res += dfs(len - 1 , limit1 && i == up1 , limit2 && j == up2 , ok1 || (j == i && j
           == 1) , ok2 || (j == 1 && j != i));
   }
}
```

参考代码 6-4-6　【时间复杂度为 $O(T \log^2)$（常数较大，请务必进行卡常数优化）】

```
#pragma GCC target ("avx2,fma")
#pragma GCC optimize ("O3")
#pragma GCC optimize ("unroll-loops")
#include<bits/stdc++.h>
#define int long long
using namespace std;

template<typename T>void read(T &res){bool flag=false;char ch;while(!isdigit(ch=getchar()
    ))(ch=='-')&&(flag=true);
for(res=ch-48;isdigit(ch=getchar());res=(res<<1)+(res<<3)+ch - 48);flag&&(res=-res);}

template<typename T>void Out(T x){if(x<0)putchar('-'),x=-x;if(x>9)Out(x/10);putchar(x%10+
    '0');}

int n , dp[32][2][2][2][2] , num[32];
inline int dfs(int len , bool limit1 , bool limit2 , bool ok1 , bool ok2){
   if(!len) return ok2 ? 1 : 0;
   if(~dp[len][limit1][limit2][ok1][ok2]) return dp[len][limit1][limit2][ok1][ok2];
   int up1 = limit1 ? num[len] : 1 , res = 0;
   for(int i = 0 ; i <= up1 ; i ++){
```

```
19          int up2 = limit2 ? i : 1;
20          for(int j = 0 ; j <= up2 ; j ++){
21              if(!ok1 && !i && j) continue ;
22              res += dfs(len - 1 , limit1 && i == up1 , limit2 && j == up2 , ok1 || (j == i
                    && j == 1) , ok2 || (j == 1 && j != i));
23          }
24      }
25      return dp[len][limit1][limit2][ok1][ok2] = res;
26  }
27  inline int solve(int n){
28      memset(dp , -1 , sizeof(dp));
29      int len = 0;
30      while(n) num[++ len] = n & 1 , n >>= 1;
31      return dfs(len , 1 , 1 , 0 , 0);
32  }
33  signed main(){
34      int T = 1;
35      read(T);
36      while(T --){
37          read(n);
38          Out(solve(n) * 3) , putchar('\n');
39      }
40      return 0;
41  }
```

6.5 组合数问题 (★★★★★)

题目信息

2019 年国赛 - 程序设计题

- ✧ C/C++ A 组第 9 题；C/C++ B 组第 10 题
- ✧ Java A 组第 9 题；Java B 组第 10 题

【问题描述】

给 n, m, k，求有多少对 (i, j) 满足 $1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \min(i, m)$ 且 $\binom{j}{i} \equiv 0 \pmod{k}$，$k$ 是质数。其中 $\binom{j}{i}$ 是组合数，表示从 i 个不同的数中选出 j 个组成一个集合的方案数。

【输入格式】

第一行两个数 t,k，其中 t 代表该测试点包含 t 组询问，k 的意思与上文中相同。

接下来 t 行每行两个整数 n,m，表示一组询问。

【输出格式】

输出 t 行，每行一个整数表示对应的答案。由于答案可能很大，请输出答案除以 10^9+7 的余数。

【样例输入】

1 2

3 3

【样例输出】

1

【评测用例规模与规定】

对于评测用例 $1,2$，$t\leqslant 1,n,m\leqslant 2000,k\leqslant 100$；

对于评测用例 $3,4$，$t\leqslant 10^5,n,m\leqslant 2000,k\leqslant 100$；

对于评测用例 $5,6,7$，$t\leqslant 100,n,m\leqslant 10^{18},k\leqslant 100$；

对于评测用例 $8,9,10$，$t\leqslant 10^5,n,m\leqslant 10^{18},k\leqslant 10^8$。

题目分析

核心考点

✧ 思维分析　✧ 逆元　✧ 数位 dp

✧ 组合数　✧ （二维）前缀和

✧ 卢卡斯定理

本题是道与组合数相关的题。在开始解题之前，我们先来简单回顾一下组合数的定义、公式及性质。

组合数的定义：从 n 个不同元素中任取 $m(m\leqslant n)$ 个元素组成一个集合称为从 n 个不同元素中取出 m 个元素的一个组合；从 n 个不同元素中取出 $m(m\leqslant n)$ 个元素的所有组合的个数称为从 n 个不同元素中取出 m 个元素的组合数，用 C_n^m 来表示。

组合数的公式：

$$C_n^m=\frac{n!}{m!\,(n-m)!}.$$

关于该公式，我们需要注意一点：$n!$、$m!$、$(n-m)!$ 都可能是规模极大的数。在用代码模拟公式计算时，如果我们先求解 $n!$、$m!$、$(n-m)!$ 的值再求解 $\dfrac{n!}{m!\,(n-m)!}$ 就可能出现整数溢出的情况。

对此，我们可以进行一些小小的优化：

（1）将公式转换成 $C_n^m = \dfrac{n \times (n-1) \times \ldots \times n - m + 1}{1 \times 2 \times \ldots \times m}$；

（2）在计算的时候 “边乘边除” 以避免溢出。

不过在许多问题中，C_n^m 本身的值就已经是溢出的，比如 C_{2000}^{1000} 的值就远大于 C++、Java 整型变量所能存储的范围。这时候无论我们怎么优化，都无法解决其溢出的问题。

好在存在这类问题的题目都会让我们 mod 上一个质数（在整型变量所能存储的范围内）然后输出。于是我们可以根据模运算的性质，来保证计算的过程使用到的变量以及结果都不会溢出。

但此时又有两点需要注意：

（1）除法是不满足模运算性质的，需要用上**逆元**来解决除法的模运算；

（2）模数不能小于 m。

组合数有 2 个性质：互补性与恒等式。

（1）互补性：

$$C_n^m = C_n^{n-m}.$$

（2）恒等式：

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}.$$

关于恒等式，我们可以用动态规划的思想来理解，即 C_n^m 表示从前 n 个不同元素中取出 m 个元素的方案数。那么对于第 n 个元素，我们有两种选择，分别是取和不取。

若取，则 C_n^m 可由从前 $n-1$ 个不同元素中取出 $m-1$ 个元素，并取第 n 元素（C_{n-1}^{m-1}）转移得到；若不取，则 C_n^m 可由从 $n-1$ 个元素中取出 m 个元素，不取第 n 个元素（C_{n-1}^m）转移得到。

因此可得转移方程：

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}.$$

我们可以用数组 $C[n][m]$ 来表示 C_n^m，并递推求出 $1 \leqslant i \leqslant n, 0 \leqslant j \leqslant m$ 的所有 $C[n][m]$。

回归本题，题意十分明确，即给定 t 个询问，每个询问给定两个整数 n, m，求出满足 $C_j^i \bmod k = 0$ （$1 \leqslant i \leqslant n, 0 \leqslant j \leqslant m$）的组合数的个数。

因为涉及模运算，所以若是想用公式法来求解组合数，就必须注意以下两点。

（1）除法改用乘法逆元。

（2）模数必须小于 j。

显然，本题的任何一个测试数据都无法保证模数 k 一定小于 j，因此弃用公式法。

除了公式法，我们还能使用动态规划（组合数的恒等式）来求解，该方法并没有什么限制。

有了组合数的求解方法，我们就可以从最浅显的算法出发，由浅到深慢慢研究。

暴力枚举

最浅显的算法自然与暴力枚举有关。

按照动态规划求解组合数的方法，我们可以用 $O(n^2)$ 的复杂度预处理出某个范围内的所有组合数。这样，在后续要求解某个 C_i^j 时，就可以以 $O(1)$ 的复杂度得到。

于是我们可以根据每个询问给出的 n, m 暴力枚举 $i, j(1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \min(i, m))$，并找出满足 $C_i^j \bmod k = 0$ 的 C_i^j 的个数（答案）。

参考代码 6-5-1　【时间复杂度为 $O(t \times n \times m)$，可通过 20% 的测试数据】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 2e3 + 10 , mod = 1e9 + 7;
int n , m , k;
long long C[N][N];
signed main(){
    int T = 1;
    cin >> T >> k;
    // 预处理得到 C[i][j]
    for(int i = 0 ; i < N ; i ++) {
        C[i][0] = 1;
        for(int j = 1 ; j <= i ; j ++) C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % k;
    }
    while(T --){
        cin >> n >> m;
        int ans = 0;
        // 暴力枚举 i , j
        for(int i = 1 ; i <= n ; i ++)
            for(int j = 0 ; j <= min(i , m) ; j ++)
                if(C[i][j] % k == 0) ans ++ ;
        cout << ans % mod << '\n';
    }
    return 0;
}
```

暴力枚举虽然编码量小、思想简单，但也存在着明显的弊端，即单次询问的计算量太大、询问和询问之间没有关联。若只进行一次询问，自然是没有什么问题。但若是进行多次询问，则其计算量将难以想象。这也是为什么暴力枚举只能通过 20% 的测试数据。

接下来让我们尝试优化。

由于用公式表示 n, m 范围之内的组合数并不利于分析，因此，我们可以选用图表的形式来呈现这些组合数，如下表所示。

	1	**2**	$\cdots$	$\boldsymbol{n}$
0	C_1^0	C_2^0		C_n^0
1	C_1^1	C_2^1		C_n^1
2		C_2^2		C_n^2
$\cdots$				$\cdots$
$\boldsymbol{m}$				C_n^m

不难发现，对于每个询问所涉及的组合数 $C_i^j(1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \min(i,m))$ 都位于图表中一个长为 m、宽为 n 的矩形中，而其中 $\bmod k = 0$ 的组合数的个数就是询问的答案。

显然，只有当 $C_i^j \bmod k = 0$ 时，它才能对答案产生贡献。

那么，不如让我们将图表中所有 $\bmod k = 0$ 的 C_i^j 视为 1，所有 $\bmod k \neq 0$ 的 C_i^j 视为 0 来重构图表。这样，矩形中所有数的和就等于 $\bmod k = 0$ 的 C_i^j 的个数，即我们要求的答案。我们只要能快速求出矩形中所有数的和就能快速求出答案，以解决多次询问计算量大的问题。

可我们如何快速求出矩形中所有数的和呢? 暴力枚举? 显然不行。那怎么办呢?

别忘了，我们有个处理矩形之和的利器 —— **二维前缀和**。

定义 $\text{sum}[i][j]$ 表示左上角为 $(1,1)$、右下角为 (i,j) 的矩形的元素和，那么有

$$\text{sum}[i][j] = \text{sum}[i-1][j] + \text{sum}[i][j-1] - \text{sum}[i-1][j-1] + (C[i][j] \bmod k == 0).$$

根据该式子，我们可以预处理求解出所有的 $\text{sum}[i][j]$ 。之后对于每个询问给定的 n, m，直接输出 $\text{sum}[n][m]$ 即可。

参考代码 6-5-2　【时间复杂度为 $O(n^2+t)$，可通过 40% 的测试数据】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 2e3 + 10 , mod = 1e9 + 7;
int n , m , k , sum[N][N];
long long C[N][N];
signed main(){
    int T = 1;
    cin >> T >> k;
    for(int i = 0 ; i < N ; i ++) {
        C[i][0] = 1; // 初始化
        for(int j = 1 ; j <= i ; j ++)
            C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % k ;
    }
    for(int i = 1 ; i < N ; i ++)
        for(int j = 1 ; j < N ; j ++){
            // 注意当 j > i 时，值为 (i, j) 对应的组合数没有意义，将其值重构为 0
            if(C[i][j] % k != 0 || j > i) C[i][j] = 0;
            else C[i][j] = 1;
            sum[i][j] = sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1] + (C[i][j]);
                sum[i][j] = (sum[i][j] + mod) % mod;
        }

    while(T --){
        cin >> n >> m;
        m = min(n , m); // m 大于 n 时没有意义
        cout << sum[n][m] << '\n';
```

```
27      }
28      return 0;
29  }
```

上述代码虽已经将每次查询的复杂度优化至 $O(1)$，但遗憾的是，我们在预处理时需要 $O(n^2)$ 的复杂度计算组合数、前缀和。本题 n, m 的规模最高可达 10^{18}，$O(n^2)$ 的复杂度是远远无法计算出结果的。

如果连最基础的组合数都无法计算，又怎么更进一步分析呢?

到这里就不得不提及一个与组合数相关的数论定理 —— **卢卡斯定理**。

卢卡斯定理是用于处理组合数取模（模数必须为质数）的定理，它在求解组合数的领域中有着重要地位，也广为人知。它能够在 n, m 很大时，以 $O(\log_k(n) \times k)$ 的时间复杂度快速求解 $C_n^m \bmod k$。

虽说卢卡斯定理的功能十分强大，但许多人只记下了它的实现代码来作为模板使用，却没有深入研究其原理。

在本题中，若只把卢卡斯定理当作一个求解组合数的工具是无法解题的，因为卢卡斯定理一次只能求解一对 (n, m) 对应的组合数的值，而不能一次求解出所有满足 $1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \min(i, j)$ 的 $C_i^j \bmod k$ 的值。因此，我们至少要了解卢卡斯定理是如何求解 C_n^m 的。

事实上，卢卡斯定理求解组合数的本质就是一个 k 进制拆分的过程，即

$$C_n^m \equiv \prod C_{n_i}^{m_i} (\bmod k).$$

其中 n_i, m_i 分别为 n, m 在 k 进制下第 i 位的值。

这是个连乘的式子。对于乘法我们知道只要有一项的值为 0，结果就为 0。所以只要有任意的 $C_{n_i}^{m_i} \bmod k = 0$，$C_n^m \bmod k$ 的值就为 0。

不过值得一提的是，n_i, m_i 是 n，m 在 k 进制下第 i 位的值，即 $0 \leqslant n_i, m_i < k$。而 $C_{n_i}^{m_i} = \dfrac{n_i!}{m_i!(n_i - m_i)!} = \dfrac{1 \times 2 \times \ldots \times n_i}{(1 \times 2 \times \ldots \times m_i)(1 \times 2 \times \ldots \times (n_i - m_i))}$，$k$ 为质数，所以 $C_{n_i}^{m_i}$ 是一定不存在 k 这个因子的。

既然如此，$C_{n_i}^{m_i} \bmod k$ 是不是一定不为 0 呢?

准确来说，并不是。

当 $m_i > n_i$ 时，$C_{n_i}^{m_i}$ 是没有意义的，而在这种情况下，我们可以认为 $C_{n_i}^{m_i}$ 的值为 0。

因此，要使 $C_n^m \bmod = 0$，就至少需要存在一对 n_i, m_i 满足 $n_i < m_i$。

于是本题就可以转换为有多少对 (i, j) 满足 $1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \min(i, m)$ 且 i 在 k 进制下至少有一位的值小于 j 在 k 进制下该位的值。

这是个经典的二维数位 dp 问题，即同时对两个数进行数位统计。

我们可以先将 n, m 转换为 k 进制数，随后定义 dp[len][limit1][limit2][ok1][ok2] ，其含义分别如下。

- len：当前处理到数位上的第 len 位。
- limit1：i 是否达到了上限。

- limit2：j 是否达到了上限。
- ok1：i 是否大于 j。
- ok2：i 是否有某一位（在 k 进制下）的值小于 j 在该位的值。

dp 数组的前三维为数位 dp 固定写法，不过多讨论，我们重点来看看后两维如何转移。

（1）ok1 的类型为 boolean，初始时 ok1 = false。由于我们是从高位往低位进行数位统计的，所以只要在某数位下 ok2 = false（未出现 $j > i$ 的情况），并且 i 这一位的数值大于 j 这一位的数值，就令 ok1 = true 。

（2）在状态转移的过程中，只要任意一位出现 i 对应的数值大于 j 对应的数值，就令 ok2 = true。

根据以上定义和分析进行数位的统计、转移，即可得到答案。

时间复杂度为 $O(tk^2 \log_k^2 n)$，可通过编号 $1, 2, 5, 6, 7$ 的测试数据。结合参考代码 3 可通过 70% 的测试数据。

参考代码 6-5-3

```
#include<bits/stdc++.h>
using namespace std;
const int mod = 1e9 + 7;
int k , num1[65] , num2[65];
long long n , m , dp[65][2][2][2][2];
long long dfs(int len , bool limit1 , bool limit2 , bool ok1 , bool ok2){
    if(!len) return ok1 && ok2;
    int up1 = limit1 ? num1[len] : k - 1;
    int up2 = limit2 ? num2[len] : k - 1;
    if(~dp[len][limit1][limit2][ok1][ok2]) return dp[len][limit1][limit2][ok1][ok2];
    long long res = 0;
    for(int i_ = 0 ; i_ <= up1 ; i_ ++){ // i 在 k 进制下第 len 位的取值
        for(int j_ = 0 ; j_ <= up2 ; j_ ++){ // j 在 k 进制下第 len 位的取值
            res += dfs(len - 1 , limit1 && i_ == up1 , limit2 && j_ == up2 , ok1 || (!ok2
                && i_ > j_) , ok2 || j_ > i_);
            res %= mod;
        }
    }
    return dp[len][limit1][limit2][ok1][ok2] = res;
}
long long solve(){
    memset(dp , -1 , sizeof(dp));
    memset(num1 , 0 , sizeof(num1));
    memset(num2 , 0 , sizeof(num2));
    m = min(n , m); // m 大于 n 时没有意义
    int cnt1 = 0 , cnt2 = 0;
    while(n) num1[++ cnt1] = n % k , n /= k; // num1 存储 n 在 k 进制下的每一位
```

```
    while(m) num2[++ cnt2] = m % k , m /= k; // num2 存储 m 在 k 进制下的每一位
    return dfs(cnt1 , 1 , 1 , 0 , 0);
}
signed main(){
    int T = 1;
    cin >> T >> k;
    while(T --){
        cin >> n >> m;
        cout << solve() << '\n';
    }
    return 0;
}
```

虽然从卢卡斯定理的本质、数位 dp 的角度切入，已经能够很好地解决 n, m 数据量极大时的情况，但其时间复杂度却深受模数 k 的影响。当模数 k 也极大时，该方法就存在局限了。

既然如此，有没有更优的做法呢?

自然是有的。

当n, m, k **极大时**，且 $m > n$ 时，其对答案的贡献与 $m = n$ 时的贡献没有差别，所以我们令 $m = \min(n, m)$。

由于 k 很大时，用记忆化搜索来实现数位 dp 并不是一个好的选择。因此，我们要尝试用递推实现。

我们的任务是计算出有多少对 (i, j) 满足 $1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \min(i, m)$ 且 i 在 k 进制下至少有一位的值小于 j 在 k 进制下该位的值，但若是用递推的方式来求解将十分困难。

为什么?

假设某对满足条件的 (i, j) 在 k 进制下的位数都是 3 位（从低到高），那么这对 (i, j) 的情况就可能有如下几种。

（1）i 在 k 进制下的第 1 位小于 j、第 2 位小于 j、第 3 位大于 j。

（2）i 在 k 进制下的第 1 位小于 j、第 2 位等于 j、第 3 位大于 j。

（3）i 在 k 进制下的第 1 位小于 j、第 2 位大于 j、第 3 位等于 j。

（4）i 在 k 进制下的第 1 位小于 j、第 2 位大于 j、第 3 位大于 j。

（5）i 在 k 进制下的第 1 位等于 j、第 2 位小于 j、第 3 位大于 j。

（6）……

提示

以上所有情况都建立在 $i \geqslant j$ 的基础上。

若 i 在 k 进制下的第 1 位小于 j 在 k 进制下的第 1 位，则 i 在 k 进制下的第 2、3 位中至少要有 1 位大于 j 以保证 $i \geqslant j$（低位会影响高位）。

在 i, j 对应的 k 进制的位数都仅有 3 位时都存在这么多种情况，可想而知，当它们对应

的 k 进制的位数不止 3 位时，要想对它们进行分析、处理是非常难实现的。

那怎么办呢?

所谓正难则反，不妨让我们采用逆向思维来分析一下该问题的补问题，即计算有多少对 (i,j) 满足 $1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \min(i,m)$ 且 i 在 k 进制下每一位的值都大于等于 j 在 k 进制下对应位的值。

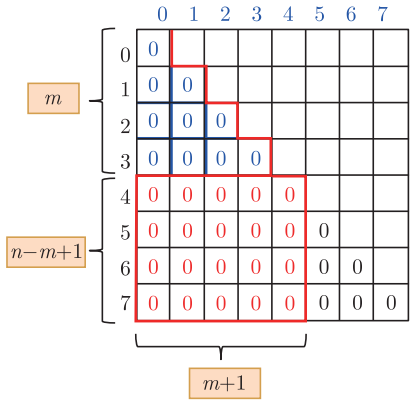
对于补问题而言，如果某对满足（补问题）条件的 (i,j) 在 k 进制下的位数都是 3 位（从低到高），那么这对 (i,j) 的情况就只会有 1 种，即 i 在 k 进制下的第 1 位大于等于 j、第 2 位大于等于 j、第 3 位大于等于 j。

提示

此时满足条件的 (i,j) 在 k 进制下任意一位的关系只有 $i>j$ 或 $i=j$，因此低位不会影响高位，可以将 “大于”“等于” 合并。

显然，补问题是要远远简单于原问题的，因此我们可以从补问题入手，求出补问题的答案（记为 cnt），再根据原问题的答案（记为 ans）= 所有（满足 $1 \leqslant i \leqslant n, 0 \leqslant j \leqslant \min(i,m)$）组合数的个数（记为 tot）– 补问题的答案（cnt）来进行求解。

但我们要如何求出所有组合数的个数 tot 呢? 以 $n=7, m=4$ 为例来求解，如下图所示。



通过上图我们不难发现，tot 即为上图中 m 部分的面积 + $(n-m+1)$ 部分的面积。

由于 n, m 是已知的，因此我们可以直接用公式计算出整体面积，即

$$\text{tot} = \frac{m \times (m+1)}{2} + (n-m+1) \times (m+1).$$

求出了 tot 后，要如何计算 cnt 呢?

正如上文所说,我们可以用递推的方式取代记忆化搜索来实现数位 dp,即定义 dp[len][limit]（limit $\in (0,1,2,3)$），其含义分别如下。

- dp[len][0]：在 k 进制下，i,j 未达到上界（i,j 第 len 位的取值都不受限制），满足 $x \leqslant i, y \leqslant j$ 且 x 的每一位都大于等于 y 的 (x,y) 的对数。

- dp[len][1]：在 k 进制下，i 的前 len 位都达到了上界，j 未达到上界（i 的第 len 位的取值受到限制，j 的第 len 位的取值不受限制），满足 $x \leqslant i, y \leqslant j$ 且 x 的每一位都大于等于 y 的 (x, y) 的对数。
- dp[len][2]：在 k 进制下，j 的前 len 位都达到了上界，i 未达到上界（i 的第 len 位的取值不受限制，j 的第 len 位的取值受到限制），满足 $x \leqslant i, y \leqslant j$ 且 x 的每一位都大于等于 y 的 (x, y) 的对数。
- dp[len][3]：在 k 进制下，i, j 的前 len 位都到达了上界（i, j 的第 len 位的取值都受到限制），满足 $x \leqslant i, y \leqslant j$ 且 x 的每一位都大于等于 y 的 (x, y) 的对数。

那么有 cnt = dp[1][0] + dp[1][1] + dp[1][2] + dp[1][3]（从高位往低位枚举）。

定义完递推方程式后，我们来思考一下如何进行状态转移。

根据 limit 的不同取值，我们可以引入 3 个函数，分别对应 limit $= 0$, limit $= 1$, limit $= 2$ 的状态转移（limit $= 3$ 时仅存在一种情况，无须额外引入函数）。

（1）calc1(i, j)：表示 x 取值范围为 $[0, i]$，y 取值范围为 $[0, j]$，且 $x \geqslant y$ 的数量（与求 tot 的方法相同）。

（2）calc2(i, j)：表示 $x = i$，y 取值范围为 $[0, j]$，且 $x \geqslant y$ 的数量（数量 $= \min(i, j) + 1$）。

（3）calc3(i, j)：表示 i 取值范围为 $[0, i]$，$y = j$，且 $x \geqslant y$ 的数量（数量 $= i - j + 1$）。

同时，我们可以分别用 $a[]$, $b[]$ 存储 n, m 在 k 进制下每一位的值，那么可以得到如下式子。

（1）
$$\begin{aligned} \text{dp[len][0]} =& \text{dp[len} + 1][0] \times \text{cal1}(k - 1, k - 1) + \text{dp[len} + 1][1] \\ & \times \text{calc1}(a[\text{len}] - 1, k - 1) + \text{dp[len} + 1][2] \\ & \times \text{calc1}(k - 1, b[\text{len}] - 1) + \text{dp[len} + 1][3] \\ & \times \text{calc1}(a[\text{len}] - 1, b[\text{len}] - 1). \end{aligned}$$

提示

i, j 在 k 进制下的第 len 位取值不受限制这一状态可以由以下状态转移而来。

- i, j 在 k 进制下第 len $+ 1$ 位取值不受限制的状态转移而来。
- i 在 k 进制下第 len $+ 1$ 位取值受限制、j 在 k 进制下第 len $+ 1$ 位取值不受限制的状态转移而来。但为了使 i 在第 len 位的取值不受限制，其在第 len 位的取值不能为 a[len]。
- i 在 k 进制下第 len $+ 1$ 位取值不受限制、j 在 k 进制下第 len $+ 1$ 位取值受限制的状态转移而来。但为了使 j 在第 len 位的取值不受限制，其在第 len 位的取值不能为 b[len]。
- i 在 k 进制下第 len $+ 1$ 位取值受限制、j 在 k 进制下第 len $+ 1$ 位取值受限制的状态转移而来。但为了使 i, j 在第 len 位的取值不受限制，i 在第 len 位的取值不能为 a[len]、j 在第 len 的取值不能为 b[len]。

（2）$\text{dp[len][1]} = \text{dp[len} + 1][1] \times \text{calc2}(a[\text{len}], k - 1) + \text{dp[len} + 1][3] \times \text{calc2}(a[\text{len}], b[\text{len}] - 1).$

提示

i 在 k 进制下的第 len 位取值受限制、j 在 k 进制下的第 len 位取值不受限制这一状态，可以由以下状态转移而来。

- i 在 k 进制下第 $\text{len}+1$ 位取值受限制、j 在 k 进制下第 $\text{len}+1$ 位取值不受限制的状态转移而来。但为了使 i 在第 len 位的取值仍然受限制，其在第 len 位的取值必须为 $a[\text{len}]$。
- i 在 k 进制下第 $\text{len}+1$ 位取值受限制、j 在 k 进制下第 $\text{len}+1$ 位取值受限制的状态转移而来。但为了使 i 在第 len 位的取值仍然受限制，其在第 len 位的取值必须为 $a[\text{len}]$。

（3）$\text{dp}[\text{len}][2] = \text{dp}[\text{len}+1][2] \times \text{calc3}(k-1, b[\text{len}]) + \text{dp}[\text{len}+1][3] \times \text{calc3}(a[\text{len}]-1, b[\text{len}])$.

提示

i 在 k 进制下的第 len 位取值不受限制、j 在 k 进制下的第 len 位取值受限制这一状态，可以由以下状态转移而来。

- i 在 k 进制下第 $\text{len}+1$ 位取值不受限制、j 在 k 进制下第 $\text{len}+1$ 位取值受限制的状态转移而来。但为了使 j 在第 len 位的取值仍然受限制，其在第 len 位的取值必须为 $b[\text{len}]$。
- i 在 k 进制下第 $\text{len}+1$ 位取值受限制、j 在 k 进制下第 $\text{len}+1$ 位取值受限制的状态转移而来。但为了使 j 在第 len 位的取值仍然受限制，其在第 len 位的取值必须为 $b[\text{len}]$。

（4）$\text{dp}[\text{len}][3] = \text{dp}[\text{len}+1][3]$ & $(a[\text{len}] >= b[\text{len}])$.

提示

i 在 k 进制下的第 len 位取值受限制、j 在 k 进制下的第 len 位取值受限制这一状态只能由 i 在 k 进制下第 $\text{len}+1$ 位取值受限制、j 在 k 进制下第 $\text{len}+1$ 位取值受限制的状态转移而来。但为了使 i, j 在第 len 位的取值仍然受限制，i 在第 len 位的取值必须为 $a[\text{len}]$，j 在第 len 位的取值必须为 $b[\text{len}]$。

参考代码 6-5-4　【时间复杂度为 $O(t\log_k(n))$，可通过所有测试数据】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int mod = 1e9 + 7 , inv2 = 5e8 + 4; // 2 的逆元
const int N = 65;
int n , m , k , dp[N][4];
int a[N] , b[N];
int calc1(int n , int m){
```

```
9     if(n < 0 || m < 0) return 0;
10    if(n < m) return (n % mod + 2) * (n % mod + 1) % mod * inv2 % mod;
11    return ((m % mod + 2) * (m % mod + 1) % mod * inv2 % mod + (n % mod - m % mod + mod)
          % mod * (m % mod + 1) % mod + mod) % mod;
12 }
13 int calc2(int n , int m){
14    return (min(n , m) + 1 + mod) % mod;
15 }
16 int calc3(int n,int m){
17    if(n < m) return 0;
18    return (n - m + 1 + mod) % mod;
19 }
20 signed main(){
21    int T = 1;
22    cin >> T >> k;
23    while(T --){
24       memset(dp , 0 , sizeof(dp));
25       for(int i = 0 ; i < N ; i ++) a[i] = b[i] = 0;
26       cin >> n >> m;
27       m = min(n , m);
28       int tot = calc1(n , m) , len1 = 0 , len2 = 0;
29       while(n) a[++ len1] = n % k , n /= k; // n 在 k 进制下的每一项
30       while(m) b[++ len2] = m % k , m /= k; // m 在 k 进制下的每一项
31       dp[len1 + 1][3] = 1;
32       for(int i = len1 ; i >= 1 ; i --){
33         // dp[i][0 ~ 3] 的状态转移
34          dp[i][0] = dp[i + 1][0] * calc1(k - 1, k - 1) % mod + dp[i + 1][1] * calc1(a[i]
                - 1, k - 1) % mod +
35                     dp[i + 1][2] * calc1(k - 1 , b[i] - 1) % mod + dp[i + 1][3] * calc1(
                           a[i] - 1 , b[i] - 1)%mod;
36          dp[i][1] = dp[i + 1][1] * calc2(a[i] , k - 1) % mod + dp[i + 1][3] * calc2(a[i]
                , b[i] - 1) % mod;
37          dp[i][2] = dp[i + 1][2] * calc3(k - 1 , b[i]) % mod + dp[i + 1][3] * calc3(a[i]
                - 1, b[i]) % mod;
38          dp[i][3] = dp[i + 1][3] & (a[i] >= b[i]);
39          dp[i][0] %= mod , dp[i][1] %= mod , dp[i][2] %= mod;
40       }
41       // 原答案 = 所有组合数的个数 - 补问题的答案
42       cout << (tot-(dp[1][0] + dp[1][1] + dp[1][2] + dp[1][3])% mod + mod)% mod <<'\n';
43    }
44    return 0;
45 }
```

6.6 巩固与练习

题目	难度	题目年份	组别
最小权值	★★	2021 年国赛	• C/C++ A 组第 4 题；C/C++ B 组第 4 题；C/C++ C 组第 5 题 • Java A 组第 3 题；Java B 组第 4 题；Java C 组第 5 题 • Python 组第 4 题
回路计数	★★	2021 年省赛	• C/C++ A 组第 5 题 • Java A 组第 5 题 • Python 组第 5 题
左孩子右兄弟	★★★	2021 年省赛	• C/C++ A 组第 8 题；C/C++ C 组第 9 题 • Java A 组第 7 题；Java C 组第 9 题 • Python 组第 98 题
二进制问题	★★★	2021 年国赛	• C/C++ B 组第 8 题；C/C++ C 组第 10 题 • Java A 组第 6 题 • Python 组第 9 题
质数行者	★★★★	2020 年国赛	• C/C++ B 组第 10 题 • Java B 组第 10 题

第7章 数据结构

CHAPTER 7

7.1 修改数组 (★★★)

题目信息

2019 年省赛 - 程序设计题

- ✧ C/C++ A 组第 8 题
- ✧ Java A 组第 8 题

【问题描述】

给定一个长度为 N 的数组 $A = [A_1, A_2, \ldots, A_N]$，数组中有可能有重复出现的整数。

现在小明要按以下方法将其修改为没有重复整数的数组。小明会依次修改 $A_2, A_3, \ldots, A_N$。

当修改 A_i 时，小明会检查 A_i 是否在 $A_1 \sim A_i - 1$ 中出现过。如果出现过，则小明会给 A_i 加上 1；如果新的 A_i 仍在之前出现过，小明会持续给 A_i 加 1，直到 A_i 没有在 $A_1 \sim A_i - 1$ 中出现过。

当 A_N 也经过上述修改之后，显然 A 数组中就没有重复的整数了。

现在给定初始的 A 数组，请你计算出最终的 A 数组。

【输入格式】

第一行包含一个整数 N。

第二行包含 N 个整数 $A_1, A_2, \ldots, A_N$。

【输出格式】

输出 N 个整数，依次是最终的 $A_1, A_2, \ldots, A_N$。

【样例输入】

5

2 1 1 3 4

【样例输出】

2 1 3 4 5

【评测用例规模与规定】

对于 80% 的评测用例，$1 \leqslant N \leqslant 10000$。

对于所有评测用例，$1 \leqslant N \leqslant 100000, 1 \leqslant A_i \leqslant 1000000$。

题目分析

核心考点

- ✧ 思维分析
- ✧ 树状数组
- ✧ 二分
- ✧ 并查集

本题的题意较为清晰，我们的任务也较为明确，即从前往后依次处理数组中的每个元素。每当数组中的某一元素 A_i 与 $A_1 \sim A_{i-1}$ 出现相同值时，就令 A_i 的值不断加 1，直至 A_i 不等同于 $A_1 \sim A_{i-1}$ 的任意一项。

单从任务的描述上来看，题目是十分简单的。如果忽略数据范围，可能绝大部分人都会想到一种极其简单的做法：双层循环 + 判重来尝试对问题的求解。

双层循环 + 判重这一方法的解题步骤大致如下。

（1）初始化

定义两个数组 vis[ ]、A[ ]，其中 vis[i] 用来判断值为 i 的数是否出现过（若出现过，则 vis[i] = 1，反之 vis[i] = 0），$A[i]$ 用来存储题目给定的第 i 个元素。默认情况下，vis[ ] 中任意一项的值均为 0。

（2）双层循环 + 判重

- 第一层循环用来遍历数组 A 的每一个元素。对于当前所遍历到的元素（设为 x），如果 vis[x] = 0（此前 x 并未出现过），则令 vis[x] = 1（表示数值 x 已出现过），继续遍历；反之，让 x 进入第二层循环。
- 在第二层循环中，我们令 x 的值不断 +1，直至 vis[x] = 0 后，令 vis[x] = 1 并跳出循环。

按照以上描述编写并成功运行代码，即可完成本题最基础的求解。

参考代码 7-1-1　【时间复杂度为 $O(n^2)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int n , a[N] , vis[N];
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++) cin >> a[i];
    for(int i = 1 ; i <= n ; i ++) {
        if(!vis[a[i]]) vis[a[i]] = 1;
```

```
10        else {
11            while(vis[a[i]]) a[i] ++ ; // 如果 a[i] 对应的值出现过，就让它不断+1，直至对应的
                  值未出现过
12            vis[a[i]] = 1;
13        }
14    }
15    for(int i = 1 ; i <= n ; i ++) cout << a[i] << " \n"[i == n];
16    return 0;
17 }
```

由于 $O(n^2)$ 的做法并不足以我们通过本题，因此我们需要考虑优化该做法。

双层循环 + 判重可拆分为以下 3 个部分。

（1）第一层循环：依次遍历数组中的每个元素。时间复杂度为 $O(n)$。

（2）第二层循环：令一个元素的值不断 $+1$，直至它的值未曾出现过。时间复杂度为 $O(n)$。

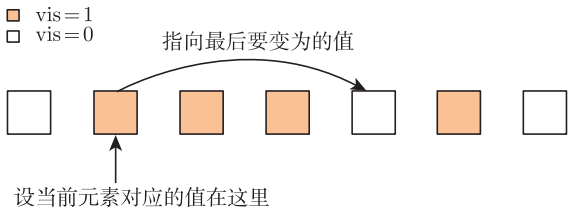
（3）判重：判断一个数是否出现过。时间复杂度为 $O(1)$。

对于第 (1) 部分，由于我们要依次处理每个元素，所以遍历数组 A 的每个元素是必要的，暂不考虑优化。

对于第 (2) 部分，我们采用了笨方法，即让元素的值通过循环不断 $+1$，直到它的值未出现过。显然，这么做会导致元素的值是一点一点改变的，这并不高效。倘若我们能将它的值快速改变为它最终要变为的值，就能达到优化的目的。

对于第 (3) 部分，该部分的时间复杂度为 $O(1)$，已达最优，无法更进一步优化。

针对上述（第 (2) 部分）分析，我们来思考一下，对于一个元素，如何快速求出它最终要变为的值？如下图所示。



假设当前要改变的元素的值为 $x(\text{vis}[x]=1)$，最后要变为的元素的值为 $y(\text{vis}[y]=0)$，那么对于 y，由于它是在 x 不断加 1 后出现的第一个未出现过的数，因此在 $x\sim y-1$，所有的数必然都是已经出现过的了，即 $\text{vis}[x], \text{vis}[x+1], \ldots, \text{vis}[y-1]$ 的值均为 1。

根据这条信息，我们就能大致锁定 y 的范围，即如果某个大于 x 的整数 z 满足 $\sum_{x}^{z}\text{vis}[i]$（$x\sim z$ 已出现过的数的个数）小于 $z-x+1$（$x\sim z$ 所有数的个数），则 y 的范围必然为 $x\sim z$；反之，如果 $\sum_{x}^{j}\text{vis}[i]=z-x+1$，则 y 必然在 z 之后。

既然给定一个大于 x 的整数 z，就能锁定 y 的大致范围，那么不妨让我们使用二分法不断缩减 y 可能存在的区间，直至求出 y 的值。

但有一个问题：二分法的判定条件是 $\sum_{x}^{z}\text{vis}[i]$ 与 $z-x+1$ 的关系，可我们要如何求解 $\sum_{x}^{z}\text{vis}[i]$ 呢?

显然，求解 $\sum_{x}^{z}\text{vis}[i]$ 的方法也必须要高效。如果使用遍历的方法（时间复杂度为 $O(n)$）来求解 $\sum_{x}^{z}\text{vis}[i]$，那么使用二分法将毫无意义。

因此，我们需要一种能快速求解 $[x,z]$ 这一区间内 $\text{vis}[i]$ 之和的工具。

对于能快速求解区间和且支持修改（vis 的值会改变）的方法有许多，其中在算法竞赛中被广泛使用的有线段树、树状数组等。

与线段树相比，树状数组的码量少、常数小，于是本题使用树状数组来求解 vis 的区间和。

参考代码 7-1-2 【时间复杂度为 $O(n(\log n)^2)$】

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int n , a[N] , vis[N] , tree[N];
int lowbit(int x){
    return x & (-x);
}
void add(int pos , int val){
    while(pos < N) tree[pos] += val , pos += lowbit(pos);
}
int query(int pos){
    int res = 0;
    while(pos) res += tree[pos] , pos -= lowbit(pos);
    return res;
}
signed main(){
    cin >> n;
    for(int i = 1 ; i <= n ; i ++) cin >> a[i];
    for(int i = 1 ; i <= n ; i ++) {
        if(!vis[a[i]]) vis[a[i]] = 1 , add(a[i] , 1); // 将已出现过的值插入树状数组中
        else {
            int l = a[i] , r = N , y = -1;
            while(l <= r){
                int mid = l + r >> 1;
                // 判断区间 [a[i], mid] 的值是否小于 mid - a[i] + 1
                if(query(mid) - query(a[i] - 1) < mid - a[i] + 1) r = mid - 1 , y = mid;
                else l = mid + 1;
            }
            a[i] = y , vis[y] = 1 , add(a[i] , 1); // 将已出现过的值插入树状数组中
        }
```

```
31      }
32      for(int i = 1 ; i <= n ; i ++) cout << a[i] << " \n"[i == n];
33      return 0;
34  }
```

除了二分和树状数组外，还有一种在竞赛中被广泛使用的数据结构可以帮助我们快速找到一个元素最终要变为的值，它就是并查集。并查集是一种树形的数据结构，可用于处理一些没有交集的**合并**及**查询**问题。

在本题中，我们可以定义一个数组 far[ ]。其中，far[i] 表示当我们遍历到的元素的值为 i 时，应将这个元素的值改变为 far[i]。

在开始遍历数组之前，因为还没有任何元素的值重复（不需要改变），所以初始化所有 far[i] 的值为 i。

接着按照顺序，开始遍历数组 A。

假设在遍历的过程中，我们当前遍历到的元素的值为 x，那么根据 far[x] 的定义，我们要将这个元素的值改变为 far[x]。

这步没有任何问题。

同时，在处理完这个元素后，我们需要更新 far[x] 的值为 far[$x+1$]，为下一次遍历值为 x 的元素做准备。

这是什么意思?

简单来说，在我们下一次遍历到一个值为 x 的元素时，由于 x 已经重复了，所以正常情况下，我们会将 x 的值改变为 $x+1$。

$$x \to x+1.$$

但 $x+1$ 也可能已经重复了，因此我们要将 $x+1$ 改变为 far[$x+1$]（far[$x+1$] 的定义是，当我们遍历到的元素的值为 $x+1$ 时，应将元素的值改变为 far[$x+1$]）。

$$x \to x+1 \to \text{far}[x+1].$$

所以，根据 far[x] 的定义，当我们表示（再次）遍历到 x 时，要将 x 改变为 far[x]，可得

$$\text{far}[x] \to x \to x+1 \to \text{far}[x+1].$$

利用并查集处理这个过程，即可完成优化。

参考代码 7-1-3　【时间复杂度为 $O(n \log n)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int n , a[N] , far[N];
int find(int x){
 return far[x] = x == far[x] ? x : find(far[x]);
```

```
}
signed main(){
    cin >> n;
    for(int i = 1 ; i < N ; i ++) far[i] = i; // 初始化
    for(int i = 1 ; i <= n ; i ++) {
        cin >> a[i];
        a[i] = find(a[i]);
        far[a[i]] = find(a[i] + 1); // x = a[i] , far[x] = far[x + 1]
    }
    for(int i = 1 ; i <= n ; i ++) cout << a[i] << " \n"[i == n];
    return 0;
}
```

7.2 翻转括号序列 (★★★★)

题目信息

2021 年国赛 - 程序设计题

- ✧ C/C++ A 组第 8 题；C/C++ B 组第 9 题
- ✧ Java B 组第 9 题
- ✧ Python 组第 10 题

【问题描述】

给定一个长度为 n 的括号序列，要求支持两种操作：

（1）将 $[L_i, R_i]$ 区间内（序列中的第 L_i 个字符到第 R_i 个字符）的括号全部翻转（左括号变成右括号，右括号变成左括号）。

（2）求出以 L_i 为左端点时，最长的合法括号序列对应的 R_i（即找出最大的 R_i 使 $[L_i, R_i]$ 是一个合法括号序列）。

【输入格式】

输入的第一行包含两个整数 n, m，分别表示括号序列长度和操作次数。

第二行包含给定的括号序列，括号序列中只包含左括号和右括号。

接下来 m 行，每行描述一个操作。如果该行为“1 L_i R_i”，表示第一种操作，区间为 $[L_i, R_i]$；如果该行为“2 L_i”表示第二种操作，左端点为 L_i。

【输出格式】

对于每个第二种操作，输出一行，表示对应的 R_i。如果不存在这样的 R_i，请输出 0。

【样例输入】

7 5

(((())()
2 3
2 2
1 3 5
2 3
2 1

【样例输出】

4
7
0
0

【评测用例规模与规定】

对于 20% 的评测用例，$n, m \leqslant 5000$；
对于 40% 的评测用例，$n, m \leqslant 30000$；
对于 60% 的评测用例，$n, m \leqslant 100000$；
对于所有评测用例，$1 \leqslant n \leqslant 10^6, 1 \leqslant m \leqslant 2 \times 10^5$。

懒人速读

给定一个长度为 n 的括号序列，序列只包含“(”和“)”两种字符。
现有 m 个操作，操作有以下两种类型。

- 1 L R：将区间 $[L, R]$ 的括号进行翻转：“(” → “)”，“)” → “(”。
- 2 L：找出（输出）最大的 R，使得区间 $[L, R]$ 对应的括号序列是合法的。

请你完成这 m 个操作。

题目分析

核心考点

- ✧ 思维分析　✧前缀和
- ✧ 二分　✧线段树

对于一个合法的括号序列而言，其必然满足以下条件。

（1）序列中的每一个右括号之前都有一个左括号可以与之匹配。

（2）序列中左括号的个数等于右括号的个数。

若光是看这两个条件，是并不利于我们分析的。好在有关判断括号序列是否合法的问题中，我们有一个最为常用的“套路”，即将左括号视为 1、右括号视为 −1。这样，判断一个括号序列是否合法就可以转换为判断以下条件。

（1）判断序列的任意前缀和是否都大于等于 0（若出现前缀小于 0 的情况，则不是合法括号序列）。

（2）判断整个序列的前缀和是否等于 0（若不等于 0，则不是合法括号序列）。

回到本题，我们的任务即根据题目给定的操作 1，对括号序列进行相应的修改；根据题目给定的操作 2，完成对应的查询。

为了方便查询，我们就按“套路”所说，即将序列中的左括号视为 1、右括号视为 -1（用 a_i 表示第 i 个括号对应的值）。

先从暴力的角度入手。

对于修改操作，我们可以轻易地遍历需要翻转的区间，将其中值为 1（左括号）的数改为 -1，将其中值为 -1（右括号）的数改为 1 以完成序列的翻转。

对于查询操作，我们可以从题目给定的左端点 L 开始不断向后遍历。遍历的同时记录一个变量 sum，表示从 L 到当前（所遍历到的）位置之间的值的和（以 L 为起点的前缀和）。若出现 $\text{sum} = 0$，且在此之前不存在 $\text{sum} < 0$ 的情况，则更新答案为当前（所遍历到的）位置；若出现 $\text{sum} < 0$ 的情况，则停止遍历，输出答案，完成查询。

参考代码 7-2-1　【时间复杂度为 $O(n \times m)$，可通过 20% 的测试数据】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int n , m , a[N];
string s;
signed main(){
    cin >> n >> m >> s;
    // 左括号视为 1，右括号视为 -1
    for(int i = 0 ; i < s.size() ; i ++) {
        if(s[i] == '(') a[i + 1] = 1;
        else a[i + 1] = -1;
    }
    while(m --){
        int op , L , R;
        cin >> op;
        if(op == 1){
            cin >> L >> R;
            // 翻转
            for(int i = L ; i <= R ; i ++) a[i] = -a[i];
        }
        else {
            cin >> L;
            int sum = 0 , ans = 0;
            for(int i = L ; i <= n ; i ++){
                sum += a[i];
```

```
26              if(sum < 0) break; // 合法括号序列的任意前缀和都必须大于等于 0
27              if(!sum) ans = max(ans , i);
28          }
29          cout << ans << '\n';
30      }
31   }
32   return 0;
33 }
```

当 n, m 的数据范围大了之后，我们就要考虑效率更高的做法。

有关括号序列的任意前缀和，我们可以用数组 sum[] 来存储，即用 sum[i] 表示 $a_1 + a_2 + \ldots + a_i$。这样，对于区间 $[L, R]$ 而言，要使其对应的括号序列合法，就必须满足以下条件。

- $\forall_{i \in [L,R]}(\text{sum}[i] - \text{sum}[L-1]) \geqslant 0$：$\text{sum}[i] - \text{sum}[L-1] = a_L + \ldots + a_i \rightarrow$ 任意前缀和大于等于 0。
- $\text{sum}[R] - \text{sum}[L-1] = 0$：$\text{sum}[R] - \text{sum}[L-1] = a_L + \ldots + a_R \rightarrow$ 整个序列的前缀和等于 0。

操作次数的总和为 m，这是无法减免的，因此要想降低程序的计算量，我们就一定得从单次操作的方向考虑如何快速完成查询操作和如何快速完成修改操作。

1. 查询操作

如果只考虑查询操作，我们可以将查询的步骤分解为以下两个部分。

（1）找出以 L 为左端点的最长的合法区间，该区间满足 $\min(\text{sum}[i]) \geqslant \text{sum}[L-1]$。

（2）在该区间中找到最靠右的满足 $\text{sum}[i] - \text{sum}[L-1] = 0$ 的端点 i（i 即为答案）。

对于第一部分，因为随着 i 的增大（区间扩大），$\min(\text{sum}[i])$ 只会越来越小，也就是 $\min(\text{sum}[i])$ 是满足单调性的。于是，我们就可以使用二分法，即以 L 为左端点、二分区间长度 len，判定区间 $[L, L + \text{len}]$ 的最小值是否大于等于 $\text{sum}[L-1]$，以找出最长的合法区间 $[L, L + \text{len}]$。

提示

判定的过程中我们需要能够快速求解区间的最小值，可以采用诸如 RMQ、线段树、树状数组等算法、数据结构来处理。本题我们选择线段树，有关线段树查询区间最小值的操作不赘述。

完成了第 (1) 部分后，紧接着的第 (2) 部分就是要让我们在一段合法区间 $[L, L + \text{len}]$（该区间内任意一个前缀和均大于等于 $\text{sum}[L-1]$）中找出满足 $\text{sum}[i] = \text{sum}[L-1]$ 最靠右的 i。

该怎么找呢?

一种常用的方法是建立一棵线段树维护两个变量：mi、val。mi 表示树上节点所维护区间的最小 sum[] 值（最小前缀和）；叶子节点的 val 值表示其对应括号的值，非叶子节点的 val

值表示其对应区间的括号值之和。

有了这样一棵树后，我们就能通过在树上二分（为了找到最长的合法区间，需要先走右子树再走左子树），轻松找出最靠右的满足 $\text{mi} = \text{sum}[L-1]$ 的端点 i（i 即为答案）。

事实上，在 $L + \text{len} < n$ 的情况下，我们不建立线段树就确定答案，答案一定是 $L + \text{len}$。

因为我们在对 len 进行二分的过程中，使用的判断条件是 $\text{sum}[L + \text{len}] \geqslant \text{sum}[L-1]$ 。那么合法区间是 $[L, L + \text{len}]$ 而不是 $[L, L + \text{len} + 1]$ 的原因只可能是 $\text{sum}[L] \sim \text{sum}[L + \text{len}]$ 的值均大于等于 $\text{sum}[L-1]$、$\text{sum}[L + \text{len} + 1]$ 的值小于 $\text{sum}[L-1]$。

由于序列每一项的值要么为 1（左括号），要么为 -1（右括号），因此相邻两项前缀和的差值的绝对值必然是 1，即 $\text{sum}[L+\text{len}+1] = \text{sum}[L+\text{len}]+1$ 或 $\text{sum}[L+\text{len}+1] = \text{sum}[L+\text{len}]-1$。再根据 $\text{sum}[L+\text{len}] \geqslant \text{sum}[L-1]$、$\text{sum}[L+\text{len}+1] < \text{sum}[L-1]$ 可得 $\text{sum}[L+\text{len}] = \text{sum}[L-1]$。

2. 修改操作

分析完查询操作后，我们只要能保证修改操作不出错，即可完成解题。修改操作十分简单，我们可以沿用查询操作建立的线段树进行分析。

假设我们翻转了区间 $[L, R]$，那么存在以下情况。

（1）根据越大的数取反后将变得越小，越小的数取反后将变得越大可得**翻转后**区间 $[L, R]$ 的最小前缀和 $\min(\text{sum}[i])$（对应线段树中的 mi）将变为**翻转前**区间 $[L, R]$ 的最大前缀和的相反数，即 $-\max(\text{sum}[i])$。同样地，**翻转后**区间 $[L, R]$ 对应的 $\max(\text{sum}[i])$ 也将变为**翻转前**区间 $[L, R]$ 的 $-\min(\text{sum}[i])$。

因此，为了维护翻转后线段树中的 mi 的变化，我们还需要往线段树中添加一个变量 ma，表示树上节点所维护区间的最大 sum[] 值（最大前缀和）。

（2）区间 $[R+1, n]$ 中每一项的前缀和 sum（对应线段树中叶子节点的 mi）均会增加/减少 $2 \times ([L, R]$ 内左括号个数 $-[L, R]$ 内右括号个数 $)$。

到这我们的分析已经结束。

我们可以结合前文分析，对线段树中所有的变量进行相对应的修改和更新，其中 lazy, add，分别表示区间翻转、区间增加/减少的懒标记。

提示

因为区间翻转、区间增加/减少的先后顺序是有影响的，所以每次当区间需要翻转时，add 的值也需要取反，即令 $\text{add} = -\text{add}$。

此外，我们所说的前缀和都是针对于整个括号序列的，因此为了不影响我们思考的逻辑，我们可以将翻转区间 $[L, R]$ 的操作置换成以下两步。

（1）翻转区间 $[1, R]$。

（2）翻转区间 $[1, L-1]$。

具体代码见参考代码 7-2-2，该参考代码实际可以通过所有测试数据。

参考代码 7-2-2　【时间复杂度为 $O(mlog^2n)$】

```
#include<bits/stdc++.h>
#define fi first
#define se second
using namespace std;
const int N = 1e6 + 10 , inf = 0x3f3f3f3f;
struct Tree{
    int l , r;
    int mi , ma , sum;
    int lazy , add;
}tree[N << 2];
int n , m , a[N] , sum[N];
char s[N];
void F1(int rt){
    tree[rt].add *= -1 , tree[rt].lazy *= -1 , tree[rt].sum *= -1;
    tree[rt].ma *= -1 , tree[rt].mi *= -1;
    swap(tree[rt].ma , tree[rt].mi);
}
void F2(int rt , int val){
    tree[rt].add += val , tree[rt].mi += val , tree[rt].ma += val;
}
void push_up(int rt){
    tree[rt].ma = max(tree[rt << 1].ma , tree[rt << 1 | 1].ma);
    tree[rt].mi = min(tree[rt << 1].mi , tree[rt << 1 | 1].mi);
    tree[rt].sum = tree[rt << 1].sum + tree[rt << 1 | 1].sum;
}
void push_down(int rt){
    int &x = tree[rt].lazy;
    if(x == -1){
        F1(rt << 1) , F1(rt << 1 | 1);
        x = 1;
    }
    int &y = tree[rt].add;
    if(y){
        F2(rt << 1 , y) , F2(rt << 1 | 1 , y);
        y = 0;
    }
}
void build(int l , int r , int rt){
    tree[rt].l = l , tree[rt].r = r , tree[rt].lazy = 1;
    if(l == r){
        tree[rt].sum = a[l];
```

```
42          tree[rt].mi = tree[rt].ma = sum[l];
43          return ;
44      }
45      int mid = l + r >> 1;
46      build(l , mid , rt << 1);
47      build(mid + 1 , r , rt << 1 | 1);
48      push_up(rt);
49  }
50  void update_change(int L , int R , int rt){
51      int l = tree[rt].l , r = tree[rt].r;
52      if(L <= l && r <= R){
53          F1(rt);
54          return ;
55      }
56      push_down(rt);
57      int mid = l + r >> 1;
58      if(L <= mid) update_change(L , R , rt << 1);
59      if(R > mid) update_change(L , R , rt << 1 | 1);
60      push_up(rt);
61  }
62  void update_add(int L , int R , int rt , int val){
63      int l = tree[rt].l , r = tree[rt].r;
64      if(L <= l && r <= R){
65          F2(rt , val);
66          return ;
67      }
68      push_down(rt);
69      int mid = l + r >> 1;
70      if(L <= mid) update_add(L , R , rt << 1 , val);
71      if(R > mid) update_add(L , R , rt << 1 | 1 , val);
72      push_up(rt);
73  }
74  int query_range(int L , int R , int rt){
75      int l = tree[rt].l , r = tree[rt].r;
76      if(L <= l && r <= R) return tree[rt].sum;
77      push_down(rt);
78      int mid = l + r >> 1 , res = 0;
79      if(L <= mid) res += query_range(L , R , rt << 1);
80      if(R > mid) res += query_range(L , R , rt << 1 | 1);
81      return res;
82  }
83  int query_min(int L , int R , int rt){
```

```
84      int l = tree[rt].l , r = tree[rt].r;
85      if(L <= l && r <= R) return tree[rt].mi;
86      push_down(rt);
87      int mid = l + r >> 1 , res = inf;
88      if(L <= mid) res = query_min(L , R , rt << 1);
89      if(R > mid) res = min(res , query_min(L , R , rt << 1 | 1));
90      return res;
91  }
92  int query_check(int L , int R , int rt , int val){
93      int l = tree[rt].l , r = tree[rt].r;
94      if(l == r) return l;
95      push_down(rt);
96      int mid = l + r >> 1;
97      if(tree[rt << 1 | 1].mi <= val) return query_check(L , R , rt << 1 | 1 , val);
98      if(L <= mid) return query_check(L , R , rt << 1 , val);
99      return 0;
100 }
101 void Change(int L , int R){
102     int add1 = 2 * query_range(1 , R , 1) * -1;
103     update_change(1 , R , 1);
104     if(R + 1 <= n) update_add(R + 1 , n , 1 , add1);
105     if(L - 1 >= 1){
106         int add2 = 2 * query_range(1 , L - 1 , 1) * -1;
107         update_change(1 , L - 1 , 1);
108         update_add(L , n , 1 , add2);
109     }
110 }
111 bool check(int L , int mid , int pre){
112     if(query_min(L , L + mid , 1) >= pre) return true;
113     return false;
114 }
115 signed main(){
116     cin >> n >> m >> s + 1;
117     for(int i = 1 ; i <= n ; i ++){
118         if(s[i] == '(') a[i] = 1 , sum[i] = sum[i - 1] + 1;
119         else a[i] = -1 , sum[i] = sum[i - 1] - 1;
120     }
121     build(1 , n , 1);
122     while(m --){
123         int op , L , R;
124         cin >> op;
125         if(op == 1){
```

```
126             cin >> L >> R;
127             Change(L , R);
128         }
129         else{
130             cin >> L;
131             if(query_range(L , L , 1) == -1){ // 第 L 个字符是右括号，则必然不存在以它为左端
                    点的合法括号序列
132                 cout << 0 << '\n';
133                 continue ;
134             }
135             int pre = query_min(L - 1, L - 1, 1); // 查询前L-1个字符的前缀和（即sum[L - 1]）
136             int l = 1 , r = n - L , len = 0;
137             while(l <= r){
138                 int mid = l + r >> 1;
139                 // 二分找出最小的 len，满足 sum[L] ~ sum[L + len] 的值都大于等于 sum[L - 1]
                        （以 L 为左端点的括号序列的任意前缀都大于等于 0）
140                 if(check(L , mid , pre)) l = mid + 1 , len = mid;
141                 else r = mid - 1;
142             }
143             // 如果 L + len < n，则 sum[L + len] 必然等于 sum[L - 1]
144             if(L + len < n){
145                 cout << L + len << '\n';
146                 continue ;
147             }
148             int ans = query_check(L , n , 1 , pre);
149             if(ans != L) cout << ans << '\n';
150             else cout << 0 << '\n';
151         }
152     }
153     return 0;
154 }
```

7.3 双向排序 (★★★★)

题目信息

2021 年省赛 - 程序设计题

- ✧ C/C++ B 组第 9 题
- ✧ Java A 组第 9 题；Java B 组第 9 题；Java C 组第 10 题

【问题描述】

给定序列 $(a_1, a_2, \ldots, a_n) = (1, 2, \ldots, n)$，即 $a_i = i$。

小蓝将对这个序列进行 m 次操作，每次可能是将 $a_1, a_2, \ldots, a_{q_i}$ 降序排列，或者将 $a_{q_i}, a_{q_{i+1}}$, …, a_n 升序排列。

请求出操作完成后的序列。

【输入格式】

输入的第一行包含两个整数 n, m，分别表示序列的长度和操作次数。

接下来 m 行描述对序列的操作，其中第 i 行包含两个整数 p_i, q_i 表示操作类型和参数。当 $p_i = 0$ 时，表示将 $a_1, a_2, \ldots, a_{q_i}$ 降序排列；当 $p_i = 1$ 时，表示将 $a_{q_i}, a_{q_{i+1}}, \ldots, a_n$ 升序排列。

【输出格式】

输出一行，包含 n 个整数，相邻的整数之间使用一个空格分隔，表示操作完成后的序列。

【样例输入】

3 3

0 3

1 2

0 2

【样例输出】

3 1 2

【样例说明】

原序列为 $(1, 2, 3)$。

第 1 步后为 $(3, 2, 1)$。

第 2 步后为 $(3, 1, 2)$。

第 3 步后为 $(3, 1, 2)$。与第 2 步操作后相同，因为前两个数已经是降序了。

【评测用例规模与规定】

对于 30% 的评测用例，$n, m \leqslant 1000$；

对于 60% 的评测用例，$n, m \leqslant 5000$；

对于所有评测用例，$1 \leqslant n, m \leqslant 100000, 0 \leqslant p_i \leqslant 11 \leqslant q_i \leqslant n$。

题目分析

核心考点

- ✧ 模拟
- ✧ 计数排序
- ✧ 线段树

本题涉及多个得分点，下面根据不同得分点采用不同的解题策略。

方式 1：能拿 30% 分数的做法

大部分人在刚开始接触编程时，都会学习各式各样的排序算法，如冒泡排序、选择排序、归并排序、插入排序、快速排序等。

每种排序都有各自的优缺点。有些优缺点体现在时间复杂度上，有些优缺点体现在空间复杂度上，还有些体现在代码量上。

在这些排序中，快速排序当属其中的“佼佼者”。不论是在算法竞赛中，还是在实际开发中，快速排序的使用频率都是最高的。正因如此，几乎每种语言的前辈都将快速排序（优化后）封装在了对应的函数库中供我们使用。

对于 30% 的测试数据，我们就可以结合 C++ 自带的快速排序函数 **sort()** 来解决。

- **sort(a + x , a + y)**：将数组 a 的区间 $[x, y)$ 进行升序排序。
- **sort(a + x, a + y , greater())**：将数组 a 的区间 $[x, y)$ 进行降序排序。

调用一次 **sort()** 函数对 n 个数排序的最坏时间复杂度为 $O(n \log_2 n)$。那么即使 m 个操作都通过调用 **sort()** 函数来排序，最坏的时间复杂度也仅有 $O(nm \log_2 n)$，足够通过 30% 的测试数据。

该参考代码实际可以通过 60% 的测试数据。

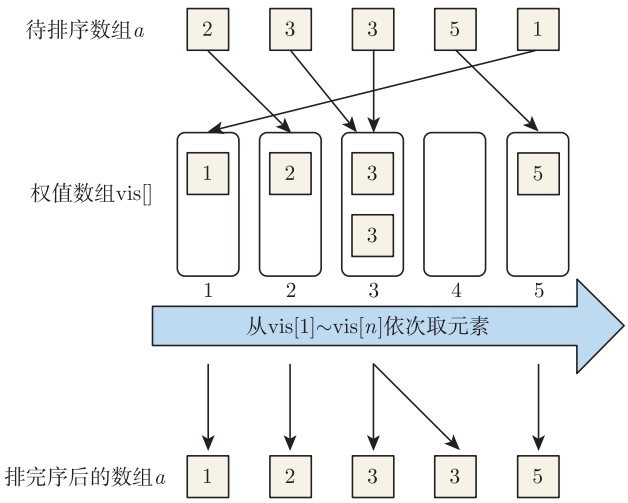
参考代码 7-3-1

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
int n , m , a[N];
signed main(){
    cin >> n >> m;
    for(int i = 1 ; i <= n ; i ++) a[i] = i;
    while(m --){
        int p , q;
        cin >> p >> q;
        if(!p) sort(a + 1 , a + 1 + q , greater<int>());
        else sort(a + q , a + 1 + n);
    }
    for(int i = 1 ; i <= n ; i ++) cout << a[i] << " \n"[i == n];
    return 0;
}
```

方式 2：能拿 60% 分数的做法

除了快速排序，我们也可以使用计数排序来解决本题。

计数排序是一种牺牲空间换取时间的做法，它的原理如下页图所示。



（1）定义一个权值（辅助）数组 vis[]。

（2）用 vis[i] 存储数组 a 中值为 i 的元素（个数）。

（3）从 vis[1] ~ vis[n] 或从 vis[n] ~ vis[1] 依次将元素取出，并按照取出的顺序将元素整齐摆放。

可以看出，计数排序并不是基于元素的比较，而是利用权值（辅助）数组的下标来确定元素的正确位置。

计数排序的时间复杂度为 $O(n+w)$，其中 w 代表待排序数组的值域大小。由于本题给定的序列 a 的范围为 $1 \sim n, n \leqslant 10^5$，因此使用计数排序要优于使用快速排序。

那么，对于降序排列操作，设它需要排序的区间为 $[1, \text{pos}]$。我们可以先将 $[1, \text{pos}]$ 内的每个数字打上标记，然后从 $n \sim 1$ 开始遍历每个数字。如果数字 i 是第 1 个被打上标记的数字，那么我们就令它插入第一个位置（即 $a[1] = i$），如果是第 2 个被打上标记的数字，我们就让它插入第二个位置（即 $a[2] = i$）..... 依次插入每个数，直到插完 $[1, \text{pos}]$ 的所有位置（如果一个数没有被标记，则说明它不在 $[1, \text{pos}]$ 这个区间内）。

对于升序排列操作同理。

参考代码 7-3-2　【时间复杂度为 $O(nm)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
int n , m , a[N] , vis[N];
signed main(){
    cin >> n >> m;
    for(int i = 1 ; i <= n ; i ++) a[i] = i;
    while(m --){
        int p , q;
        cin >> p >> q;
```

```
11        if(!p) {
12            for(int i = 1 ; i <= q ; i ++) vis[a[i]] = 1;
13            int pos = 0;
14            for(int i = n ; i >= 1 ; i --) if(vis[i]) a[++ pos] = i , vis[i] = 0;
15        }
16        else {
17            for(int i = q ; i <= n ; i ++) vis[a[i]] = 1;
18            int pos = q - 1;
19            for(int i = 1 ; i <= n ; i ++) if(vis[i]) a[++ pos] = i , vis[i] = 0;
20        }
21    }
22    for(int i = 1 ; i <= n ; i ++) cout << a[i] << " \n"[i == n];
23    return 0;
24 }
```

方式 3：100% 满分做法

由于并没有什么排序算法的时间复杂度能稳定在 $O(n)$ 以下，因此当 $n \leqslant 10^5$ 时，继续讨论排序算法将不再有意义。我们应记得及时调整切入方向，去深入研究题目（隐藏）的性质。

在本题中，只存在两种类型的操作：升序操作和降序操作。

既然这 m 次操作都源自于以上两种类型，那这其中是否会存在一些关联呢?

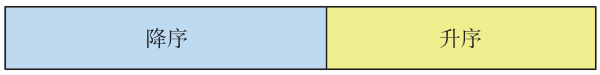
我们来分析一下。

起初，序列是升序的。

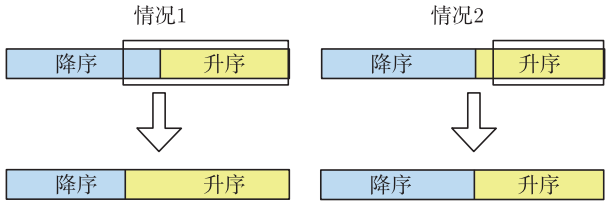
如果一开始就进行升序操作，显然是毫无意义的（不会影响序列的排序），即无效操作。

对于无效操作，我们理当将其忽视。待我们将所有的无效操作忽视后，首先进行的有效操作必然会是个降序操作。

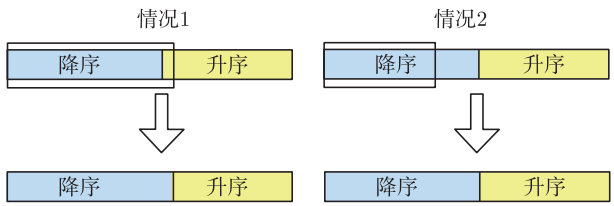
因此，在进行了第一次的有效操作后，序列将会产生一段降序序列与升序序列，如下图所示。



此时，若我们想对序列进行一次升序操作，则该操作可分为两种情况，如下图所示。



若想对序列进行一次降序操作，则该操作可分为两种情况，如下页图所示。



从上图不难发现，无论是进行了哪种类型的操作、产生了哪种情况，在每次操作结束后，都总会有一个位置（设为 pos）满足 $[1, \text{pos}]$ 的元素单调递减，$[\text{pos}+1, n]$ 的元素单调递增。

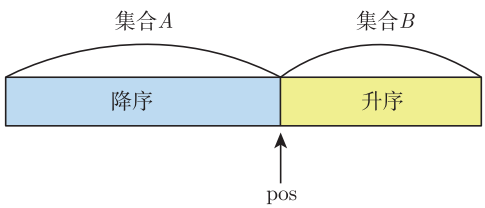
我们可以使用 60% 分数做法中的方法将区间 $[1, \text{pos}]$ 的数标记为 0，将区间 $[\text{pos}+1, n]$ 的数标记为 1，这样在所有操作结束后，从 $n \sim 1$ 依次输出所有标记为 0 的数，得到的就是答案的降序部分；从 $1 \sim n$ 依次输出所有标记为 1 的数，得到的就是答案的升序部分。

但问题来了，在 60% 分数的做法中，我们是可以通过遍历区间来给数字打标记的。而在 $n \leqslant 10^5$ 的数据范围下，遍历区间是必然不可行的。那么，要怎么为区间内的数字打标记呢？

我们接着往下思考。

由于刚开始 $(a_1, a_2, \ldots, a_n) = (1, 2, \ldots, n)$，整个序列都是升序的，所以我们可以在初始化的时候将所有数的标记都置为 1（或者可以将 a_1 单独视为一个递减序列，将 $a_2 \sim a_n$ 视为递增序列，这样数字 1 的标记就为 0，数字 $2 \sim n$ 的标记就为 1）。

为了方便描述，我们可以将递减的序列译为集合 A，将递增的序列译为集合 B，那么区间 $[1, \text{pos}]$ 的元素就属于集合 A，区间 $[\text{pos}+1, n]$ 的元素就属于集合 B，如下图所示。



对于某次**升序操作**，若其对应的升序区间为 $[x, n]$，则会存在以下两种情况。

（1）$\text{pos} \leqslant x$：由于 $[\text{pos}, n]$ 是升序的，即 $[x, n]$ 也是升序的，所以本次操作是个无效操作，无须处理。

（2）$\text{pos} > x$：此时 $[\text{pos}, n]$ 是升序的，但 $[x, \text{pos}-1]$ 还是降序的，所以我们需要将区间 $[x, \text{pos}-1]$ 内的数的标记改为 1。

在进行情况 2 中所说的处理之前，我们需要先知道区间 $[x, \text{pos}-1]$ 内的数分别是什么（它们有什么样的性质、特点）。

根据集合 A 是降序的可得，$[x, \text{pos}-1]$ 内的数其实就是标记为 0 的、最小的 $\text{pos}-x$ 个数。可我们要如何将这标记为 0 的、最小的 $\text{pos}-x$ 的标记**快速**修改为 1 呢？

不妨让我们从数据结构的角度思考。

我们需要一种能够存储信息（每个数的标记值），同时能够**快速**、**批量**地修改多个数的信息的数据结构。这时就非常适合用线段树。

我们可以建立一棵权值线段树，树的每个叶子节点的值对应每个数字的标记值，树的每个节点维护其对应区间内的标记为 0 的数的个数。

这样处理之后，若想将标记为 0 的、最小的 $\mathrm{pos}-x$ 个数的标记值改为 1，就只要在整棵树上二分（先找左子树再找右子树）修改标记为 0 的数的个数即可。

对于某次**降序操作**，其对应情况和处理方式与**升序操作**相似，这里就不再过多赘述。

在处理完所有操作后，从 $n\sim 1$ 依次输出所有标记为 0 的数，再从 $1\sim n$ 依次输出所有标记为 1 的数，即可完成解答。

参考代码 7-3-3　【时间复杂度为 $O(m\log_2 n)$】

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
int n , m , op , pos;
struct Tree{
    int l , r , sum , lazy;
}tree[N << 2];
void push_up(int rt){
    tree[rt].sum = tree[rt << 1].sum + tree[rt << 1 | 1].sum;
}
void push_down(int rt){
    int &x = tree[rt].lazy;
    if(~x){
        int len1 = tree[rt << 1].r - tree[rt << 1].l + 1;
        int len2 = tree[rt << 1 | 1].r - tree[rt << 1 | 1].l + 1;
        tree[rt << 1].sum = len1 * x , tree[rt << 1 | 1].sum = len2 * x;
        tree[rt << 1].lazy = tree[rt << 1 | 1].lazy = x;
        x = -1;
    }
}
void build(int l , int r , int rt){
    tree[rt].l = l , tree[rt].r = r , tree[rt].lazy = -1;
    if(l == r) {
        tree[rt].sum = 1;
        return ;
    }
    int mid = l + r >> 1;
    build(l , mid , rt << 1);
    build(mid + 1 , r , rt << 1 | 1);
    push_up(rt);
}
void update1(int L , int R , int rt , int cnt){
    if(!cnt) return ;
```

```
34      if(tree[rt].sum == cnt){
35          tree[rt].sum = tree[rt].lazy = 0;
36          return ;
37      }
38      push_down(rt);
39      if(cnt < tree[rt << 1].sum) update1(L , R , rt << 1 , cnt);
40      else {
41          update1(L , R , rt << 1 | 1 , cnt - tree[rt << 1].sum);
42          tree[rt << 1].sum = 0;
43          tree[rt << 1].lazy = 0;
44      }
45      push_up(rt);
46  }
47  void update2(int L , int R , int rt , int cnt){
48      if(!cnt) return ;
49      int len = tree[rt].r - tree[rt].l + 1;
50      if(len - tree[rt].sum == cnt){
51          tree[rt].sum = len;
52          tree[rt].lazy = 1;
53          return ;
54      }
55      push_down(rt);
56      int cnt1 = tree[rt << 1].r - tree[rt << 1].l + 1 - tree[rt << 1].sum;
57      if(cnt < cnt1) update2(L , R , rt << 1 , cnt);
58      else {
59          update2(L , R , rt << 1 | 1 , cnt - cnt1);
60          tree[rt << 1].sum = tree[rt << 1].r - tree[rt << 1].l + 1;
61          tree[rt << 1].lazy = 1;
62      }
63      push_up(rt);
64  }
65  int query(int L , int R , int rt){
66      int l = tree[rt].l , r = tree[rt].r;
67      if(L <= l && r <= R) return tree[rt].sum;
68      push_down(rt);
69      int mid = l + r >> 1 , res = 0;
70      if(L <= mid) res += query(L , R , rt << 1);
71      if(R > mid) res += query(L , R , rt << 1 | 1);
72      return res;
73  }
74  signed main(){
75      cin >> n >> m;
```

```
76      build(1 , n , 1);
77      while(m --){
78          cin >> op >> pos;
79          if(!op){
80              int cnt0 = n - tree[1].sum; // 整个序列中 0 的个数
81              int cnt = max(0 , pos - cnt0); // 1 ~ pos 中 0 的个数
82              update1(1 , n , 1 , cnt); // 将最大的 cnt 个 0 改为 1
83          }
84          else{
85              int cnt1 = tree[1].sum;
86              int cnt = max(0 , n - pos + 1 - cnt1); // 同上
87              update2(1 , n , 1 , cnt);
88          }
89      }
90      for(int i = n ; i >= 1 ; i --) if(!query(i , i , 1)) cout << i << " "; // 输出集合 A
91      for(int i = 1 ; i <= n ; i ++) if(query(i , i , 1)) cout << i << " "; // 输出集合 B
92      return 0;
93  }
```

7.4 冰山 (★★★★)

题目信息

2021 年国赛 - 程序设计题

- ✧ C/C++ A 组第 7 题
- ✧ Java A 组第 7 题；Java C 组第 9 题
- ✧ Python 组第 7 题

【问题描述】

一片海域上有一些冰山，第 i 座冰山的体积为 V_i。

随着气温的变化，冰山的体积可能增大或缩小。第 i 天，每座冰山的变化量都是 X_i。当 $X_i > 0$ 时，所有冰山体积增加 X_i；当 $X_i < 0$ 时，所有冰山体积减小 X_i；当 $X_i = 0$ 时，所有冰山体积不变。

如果第 i 天某座冰山的体积变化后小于等于 0，则冰山会永远消失。

冰山有大小限制 k。如果第 i 天某座冰山 j 的体积变化后 V_j 大于 k，则它会分裂成一个体积为 k 的冰山和 $V_j - k$ 座体积为 1 的冰山。

第 i 天结束前（冰山增大、缩小、消失、分裂完成后），会漂来一座体积为 Y_i 的冰山（$Y_i = 0$ 表示没有冰山漂来）。小蓝在连续的 m 天对这片海域进行了观察，并准确记

录了冰山的变化。

小蓝想知道，每天结束时所有冰山的体积之和（包括新漂来的）是多少。由于答案可能很大，请输出答案除以 998244353 的余数。

【输入格式】

输入的第一行包含三个整数 n, m, k，分别表示初始时冰山的数量、观察的天数以及冰山的大小限制。

第二行包含 n 个整数 $V_1, V_2, \ldots, V_n$，表示初始时每座冰山的体积。

接下来 m 行描述观察的 m 天的冰山变化。其中第 i 行包含两个整数 X_i, Y_i，意义如前所述。

【输出格式】

输出 m 行，每行包含一个整数，分别对应每天结束时所有冰山的体积之和除以 998244353 的余数。

【样例输入】

1 3 6

1

6 1

2 2

−1 1

【样例输出】

8

16

11

【样例说明】在本样例说明中，用 $[a_1, a_2, \ldots, a_n]$ 来表示每座冰山的体积。

初始时的冰山为 [1]。

第 1 天结束时，有 3 座冰山：[1, 1, 6]。

第 2 天结束时，有 6 座冰山：[1, 1, 2, 3, 3, 6]。

第 3 天结束时，有 5 座冰山：[1, 1, 2, 2, 5]。

【评测用例规模与规定】

对于 40% 的评测用例，$n, m, k \leqslant 2000$；

对于 60% 的评测用例，$n, m, k \leqslant 20000$；

对于所有评测用例，$1 \leqslant n, m \leqslant 100000, 1 \leqslant k \leqslant 10^9, 1 \leqslant V_i \leqslant k, 0 \leqslant Y_i \leqslant k, -k \leqslant X_i \leqslant k$。

懒人速读

给定一个长度为 n 的数组 $a_1, a_2, \ldots, a_n$ 和一个 k。

现有 m 种操作，每种操作包含两个常数 x, y，我们需要将数组中的每个元素的值加上 x（x 可能为负数），然后再向数组中插入一个值为 y 的元素。

在操作的过程中，需要满足以下两个约束条件。

- 若一个元素的值 $\text{val} \geqslant k$，则它的值将被限制为 k，同时需要向数组中添加 $\text{val} - k$ 个值为 1 的元素。
- 若一个元素的值 $\text{val} \leqslant 0$，则它将从数组中被删除。

问每次操作结束后数组中所有元素的总和为多少？

题目分析

核心考点

- ✧ 思维分析
- ✧ 模拟
- ✧ 平衡树

若暂时忽略数据范围，本题也只是一个基础的模拟题，它的实现并不复杂。

我们可以创建一个 vector 容器来代表数组。

每进行一种操作就遍历容器中的所有元素，令这些元素的值加上 x；待遍历结束后，通过 vector 的 push_back() 函数向容器的末尾添加一个元素 y。这样，我们就完成了一次常规的操作。

当然，我们还得满足题目给定的约束条件。因此，在我们操作结束后，还需要再一次遍历数组，将其中元素值小于等于 0 的元素删除（可通过调用 vector 自带的 erase() 函数处理）、元素值大于 k 的元素改为 k（多出来的部分可以先统一用一个变量存储，等遍历结束后再转换成一个个值为 1 的元素 push_back() 进容器的末尾）。

参考代码 7-4-1　【时间复杂度约为 $O(n^n)$】

```
#include<bits/stdc++.h>
using namespace std;
const int mod = 998244353;
signed main(){
  int n , m , k;
  cin >> n >> m >> k;
  vector<int>vec;
  for(int i = 1 ; i <= n ; i ++){
    int x;
```

```
    cin >> x;
    vec.push_back(x);
  }
  while(m --){
    int x , y;
    cin >> x >> y;
    for(int i = 0 ; i < vec.size() ; i ++) vec[i] += x;
    vec.push_back(y);
    for(vector<int>::iterator it = vec.begin() ; it != vec.end() ; ) {
      if(*it <= 0) it = vec.erase(it); // 如果值小于等于 0，则删除
      else it ++ ;
    }
    int cnt = 0;
    for(int i = 0 ; i < vec.size() ; i ++) if(vec[i] > k) cnt += vec[i] - k , vec[i] = k;
        // 用 cnt 记录将要添加的 1 的个数
    while(cnt --) vec.push_back(1);
    long long V = 0;
    for(int i = 0 ; i < vec.size() ; i ++) V = (V + vec[i]) % mod;
    cout << V << '\n';
  }
  return 0;
}
```

显然，上面的方法过于“暴力”。不难想象，在数据较为极端的情况下，即使没有时间上的限制，空间上的限制也会让计算机无法承担。

对此，我们必须进行优化。

通常情况下，相较于时间复杂度，空间复杂度往往是更为容易优化的。那么，不妨让我们先分析一下如何优化空间复杂度。

在本题中，我们每次的操作都是针对于整个数组（容器）的，并不会针对于某个特定的位置或空间。也就是说，对于一个元素，不论将其放置在数组（容器）中的哪个位置，它都会受到操作的影响，即位置是不会影响元素值的改变的。

我们知道，本题的答案为数组（容器）中所有元素值之和，它只受到元素值的影响。既然位置不会影响元素的值，那么不难推断出，位置是不会影响答案的。

因此，我们是不是可以认为，只要是值相同的元素，就都可以看作是相同的元素呢?

确实可以。这是因为区别数组中两个元素为不同元素的因素仅有以下两个条件。

（1）元素在数组中的位置。

（2）元素的值。

由于元素的位置对答案并不会造成影响（可以忽略这个属性），所以只要两个元素的值相同，我们就能认为这两个元素没有区别（为同一元素）。

相同元素受到操作的影响也一定是相同的。正因如此，我们可以使用一个二元组 $<$ value,

cnt > 来表示元素 value 在数组（容器）中共出现了 cnt 次，以此来控制并维持元素的总个数在 $O(n)$ 级别。

那么接下来，我们的解题核心就在于要找到一个能够处理以下 4 点的数据结构。

（1）维护二元组 $<\text{value}, \text{cnt}>$。

（2）可以插入元素。

（3）可以实现加法操作 —— 值超过 k 的元素分裂，值小于等于 0 的元素删除。

（4）维护所有元素之和。

从简单的方向寻找，用集合 set $<\text{value}, \text{cnt}>$ 来处理是最为合适的。具体的处理方法如下。

（1）对于插入元素，直接使用 set 自带的 insert() 函数完成。

（2）对于加法操作，我们可以将其分为以下两种情况。

- $x > 0$：
  - value 范围为 $[1, k-x-1]$ 的二元组，直接暴力将它们的值加上 x（加上 x 后值不会超过 k）即可。
  - value 范围为 $[k-x, k]$ 的二元组，将它们对应的元素值加上 x 后，它们的值都会大于等于 k。根据题目的约束条件，这些元素的值都要变为 k，且会分裂出若干个值为 1 的元素。同样地，直接暴力将它们的值变为 k，并计算它们变成 k 时会分裂出多少个 1。假设有 tot 个 1，则在最后向集合中插入一个新的二元组 $<1, \text{tot}>$。
- $x < 0$：做法与 $x < 0$ 的类似。

（3）对于所有元素之和（答案），它会等于集合中所有二元组的 value × cnt 之和，即 $\sum(\text{value} \times \text{cnt})$。我们可以遍历 set、暴力统计。

参考代码 7-4-2　【时间复杂度为 $O(nm\log)$】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int N = 5e5 + 10 , mod = 998244353;
set<pair<int , int>>se;
map<int , int>mp;
int n , m , k;
signed main(){
   cin >> n >> m >> k;
   for(int i = 1 ; i <= n ; i ++){
      int x;
      cin >> x;
      if(mp.find(x) != mp.end()) se.insert(make_pair(x , 1));
      else {
         se.erase(make_pair(x , mp[x]));
```

```
16          se.insert(make_pair(x , mp[x] + 1));
17      }
18      mp[x] ++ ; // mp 用于判断某个数值是否存在
19  }
20  while(m --){
21      int x , y;
22      cin >> x >> y;
23      set<pair<int , int>>tmp1;
24      map<int , int>tmp2;
25      // 先将操作的结果保存在 tmp1, tmp2 中，待操作结束后再令 se = tmp1, mp = tmp2
26      int cnt = 0 , tot = 0; // cnt 记录值大于等于 k 的元素个数， tot 记录即将生成的值为 1
            的元素个数
27      for(auto i : se){
28          if(i.first + x <= 0) continue ; // 若值小于等于 0 则跳过
29          if(i.first + x < k){
30              tmp1.insert(make_pair(i.first + x , i.second));
31              tmp2[i.first + x] = i.second;
32          }
33          else {
34              tot += (i.first + x - k) * i.second;
35              cnt += i.second;
36          }
37      }
38      if(cnt > 0) tmp1.insert(make_pair(k , cnt));
39      if(tot > 0) tmp1.insert(make_pair(1 , tot));
40      tmp2[1] = tot , tmp2[k] = cnt;
41      if(tmp2.find(y) != tmp2.end()){
42          tmp1.erase(make_pair(y , tmp2[y]));
43          tmp1.insert(make_pair(y , tmp2[y] + 1));
44          tmp2[y] ++ ;
45      } else {
46          tmp1.insert(make_pair(y , 1));
47          tmp2[y] = 1;
48      }
49      se = tmp1 , mp = tmp2; // 操作结束后
50      int ans = 0;
51      for(auto i : se) { // 遍历集合，计算答案
52          ans += i.first * i.second % mod;
53          ans %= mod;
54          cout << i.first << " " << i.second << '\n';
55      }
56      cout << ans << '\n';
```

```
57      }
58      return 0;
59  }
```

set 凭借着其特殊的存储方式以及封装好的函数，帮助我们优化了程序的空间复杂度，并实现了一些基本操作。但其也有着致命的缺点，即不支持区间（范围）的修改、查询，一次只能处理一个二元组。

因为这个缺点，我们不得不在进行每次操作时都遍历整个 set 集合，对每一个二元组一一进行操作。

这无疑会使程序产生大量的计算，导致时间复杂度过高。

因此，我们还需要寻找一个更为优秀的数据结构，它需要在继承 set 优点的同时，弥补 set 的缺点，即满足以下 4 个条件。

（1）可维护二元组 $<\text{value}, \text{cnt}>$。

（2）可以插入元素。

（3）**可在权值上进行区间性的修改**。

（4）可求解所有元素之和。

这不禁让我们想到**平衡树**。

由于在平衡树的相关算法中，fhq – treap 是相对好理解且码量小的，因此，本题我们引入 fhq – treap 来解决。

fhq – treap 中的节点本身需要维护的信息有以下 3 项。

（1）l：左儿子的节点编号。

（2）r：右儿子的节点编号。

（3）key：节点的随机值。

在本题中，我们需要额外维护的信息有以下 5 项。

（1）value：节点的数值。

（2）cnt：节点对应数值出现的次数。

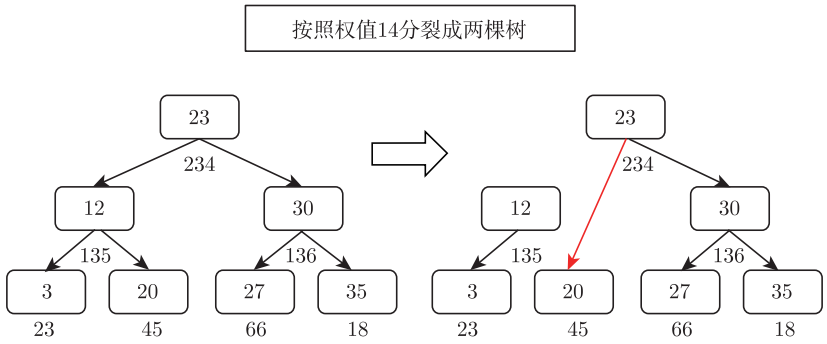
（3）size：所有子节点的 cnt 之和。

（4）sum：所有子节点的 value × cnt 之和。

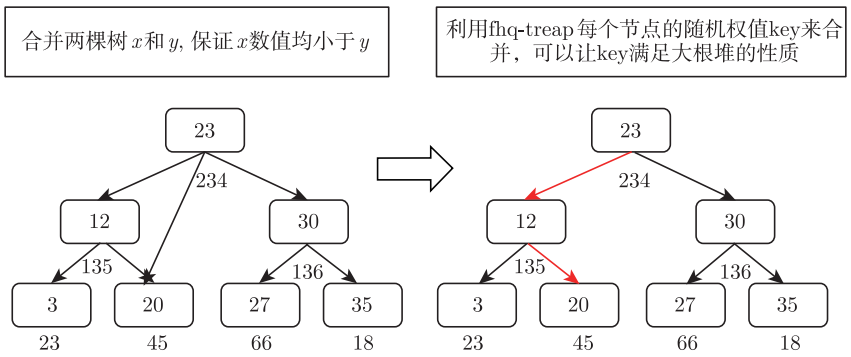
（5）lazy：区间修改的懒标记。

fhq – treap 是基于**分裂 (split)** 与**合并 (merge)** 来完成各项操作的。

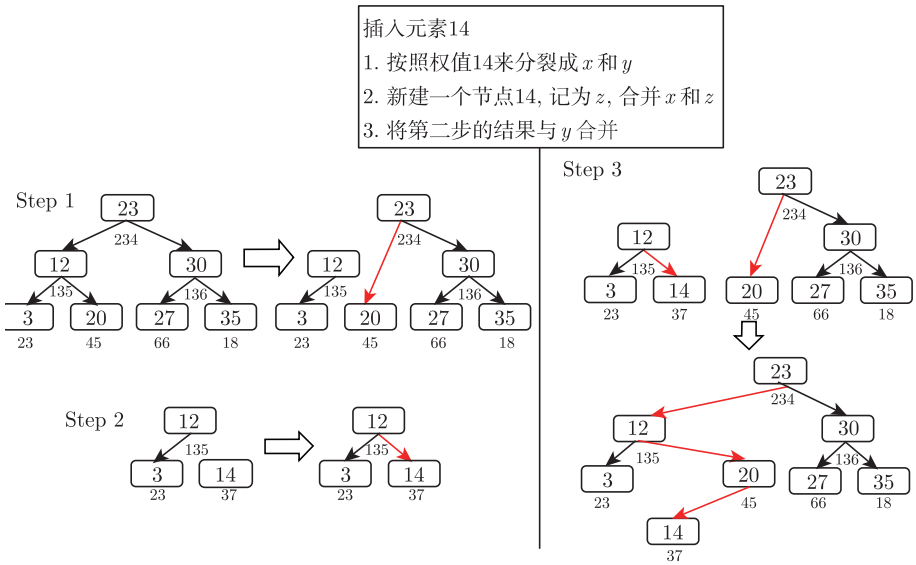
（1）**分裂**：根据待插入元素的值将整棵树分裂成两部分。节点下方的数字为 key（随机值），通过 key 随机确定树的形态，从而使树尽量平衡。



（2）**合并**：合并两棵树。



插入操作：插入元素 14 的步骤如下图所示。



删除操作：先根据待删除元素的值（记为 value）将树分裂成两部分，一部分小于等于 value（记为 x），另一部分大于 value（记为 y），再将 x 分裂为两部分，一部分小于 value（记

为 x'），另一部分等于 value（记为 x''），然后将 x' 与 y 合并即可。

简单回顾完 fhq – treap 的基本内容后，我们来分析一下如何用 fhq – treap 在元素的权值上进行区间性修改。

1. 整个序列的元素值加 x（即 $x > 0$）

针对该操作，我们可将序列分为以下两部分单独讨论。

（1）value $\leqslant k - x - 1$。这部分元素在它们的值加上了 x 后，我们要将它们的 value 更新为 $value + x$。

（2）value $> k - x - 1$。这部分元素在它们的值加上了 x 后，值都会大于等于 k。根据题目的约束条件，这些元素的值都要变为 k，且会分裂出若干个值为 1 的元素。

提示

“若干个”具体是多少呢?

若用 sum 表示这部分元素的 value 总和，用 size 表示它们的 cnt 总和（在平衡树的节点中，我们用 sum 记录了这部分元素的 value 总和，用 size 记录了这部分元素的 cnt 总和），那么操作带来的贡献总和就为 size $\times x$（操作能令这些元素的和增加）。

要使这部分元素的 value 全部变为 k，需要的贡献为 size $\times k -$ sum。

多出来的贡献（产生的值为 1 的元素个数）就为 size $\times x -$ (size $\times k -$ sum)。

那要怎么实现呢?

首先以 $(k - x - 1)$ 为权值，将平衡树分裂为 A, B 两部分（A 对应第一部分，B 对应第二部分）。

然后，对于 A 部分，我们只要在其根节点处加上值为 x 的 $lazy$ 即可。对于 B 部分，我们直接将其删除，然后再创建一个 value 为 k、cnt 为 size 的节点，以及一个 value 为 1、cnt 为 size $\times x -$ (size $\times k -$ sum) 的节点，令它们与 A 合并即可。

提示

操作前序列中 value 大于 $k - x - 1$ 的这部分二元组在操作后将不复存在，取而代之的是二元组 (k, size)，以及二元组 $(1, \text{size} \times x - (\text{size} \times k - \text{sum}))$。

2. 整个序列的元素值减 x (即 $x < 0$)

针对该操作，同样也将序列分为以下两部分单独讨论。

（1）value $\leqslant |x|$。这部分元素在它们的值减少了 $|x|$ 后，值都将小于等于 0，需要删除。

（2）value $> |x|$。这部分元素在它们的值减少了 $|x|$ 后，需要将它们的 value 更新为 value $- |x|$。

关于实现方法与将元素值加 x 的类似。

首先以 $|x|$ 为权值，将平衡树分裂为 A, B 两部分（A 对应第 (1) 部分，B 对应第 (2) 部分）。

然后，对于 A 部分，我们直接将其删除。对于 B 部分，我们只要在其根节点处加上一个值为 x 的 $lazy$ 即可。

3. 所有元素之和

对于平衡树上的节点，我们用 sum 记录了所有子节点的 value × cnt 之和。因此，每次操作结束后，根节点的 sum 即为所有元素之和（答案）。

参考代码 7-4-3　【时间复杂度为 $O((n+m)\log)$】

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int N = 5e5 + 10 , mod = 998244353;
struct FHQ_Tree{
    int l , r , key , lazy = 0;
    int size , sum , val , cnt;
}tree[N];
int root , A , B , Z , tot;
int n , m , k , x , y;
int create(int val , int cnt){
    tree[++ tot].size = cnt;
    tree[tot].cnt = cnt;
    tree[tot].val = val;
    tree[tot].key = rand();
    tree[tot].sum = val * cnt % mod;
    return tot;
}
void push_up(int rt){
    tree[rt].size = (tree[tree[rt].l].size + tree[tree[rt].r].size + tree[rt].cnt) % mod;
    tree[rt].sum = (tree[tree[rt].l].sum + tree[tree[rt].r].sum + tree[rt].val * tree[rt
        ].cnt % mod + mod) % mod;
}
void push_down(int rt){
    if(tree[rt].lazy){
        int &x = tree[rt].lazy;
        tree[tree[rt].l].lazy += x , tree[tree[rt].r].lazy += x;
        tree[tree[rt].l].val += x , tree[tree[rt].r].val += x;
        tree[tree[rt].l].sum = (tree[tree[rt].l].sum + tree[tree[rt].l].size * x % mod +
            mod) % mod;
        tree[tree[rt].r].sum = (tree[tree[rt].r].sum + tree[tree[rt].r].size * x % mod +
            mod) % mod ;
        x = 0;
    }
}
```

```
33 void split(int rt , int val , int &x , int &y){
34     if(!rt) {
35         x = y = 0;
36         return ;
37     }
38     push_down(rt);
39     if(tree[rt].val <= val){
40         x = rt;
41         split(tree[rt].r , val , tree[rt].r , y);
42     }
43     else{
44         y = rt;
45         split(tree[rt].l , val , x , tree[rt].l);
46     }
47     push_up(rt);
48 }
49 int merge(int x , int y){
50     if(!x || !y) return x + y;
51     if(tree[x].key > tree[y].key){
52         push_down(x);
53         tree[x].r = merge(tree[x].r , y);
54         push_up(x);
55         return x;
56     }
57     else{
58         push_down(y);
59         tree[y].l = merge(x , tree[y].l);
60         push_up(y);
61         return y;
62     }
63 }
64 void insert(int val , int size){
65     split(root , val , A , B);
66     root = merge(merge(A , create(val , size)) , B);
67 }
68 void update(int x){
69     if(!x) return ;
70     if(x > 0){
71         int val = k - x - 1;
72         split(root , val , A , B);
73         tree[A].lazy += x , tree[A].val += x;
74         tree[A].sum = (tree[A].sum + tree[A].size * x % mod) % mod;
```

```
75          int cha = (k * tree[B].size % mod - tree[B].sum + mod) % mod;
76          int add_cnt = (x * tree[B].size % mod - cha + mod) % mod;
77          root = merge(A , create(k , tree[B].size));
78          insert(1 , add_cnt);
79      }
80      else{
81          split(root , abs(x) , A , B);
82          tree[B].lazy += x , tree[B].val += x;
83          tree[B].sum = ((tree[B].sum + tree[B].size * x % mod) % mod + mod) % mod;
84          root = B;
85      }
86  }
87  signed main(){
88      cin >> n >> m >> k;
89      for(int i = 1 ; i <= n ; i ++) {
90          cin >> x;
91          insert(x , 1);
92      }
93      while(m --){
94          cin >> x >> y;
95          update(x);
96          if(y) insert(y , 1);
97          cout << tree[root].sum << '\n';
98      }
99      return 0;
100 }
```

7.5　巩固与练习

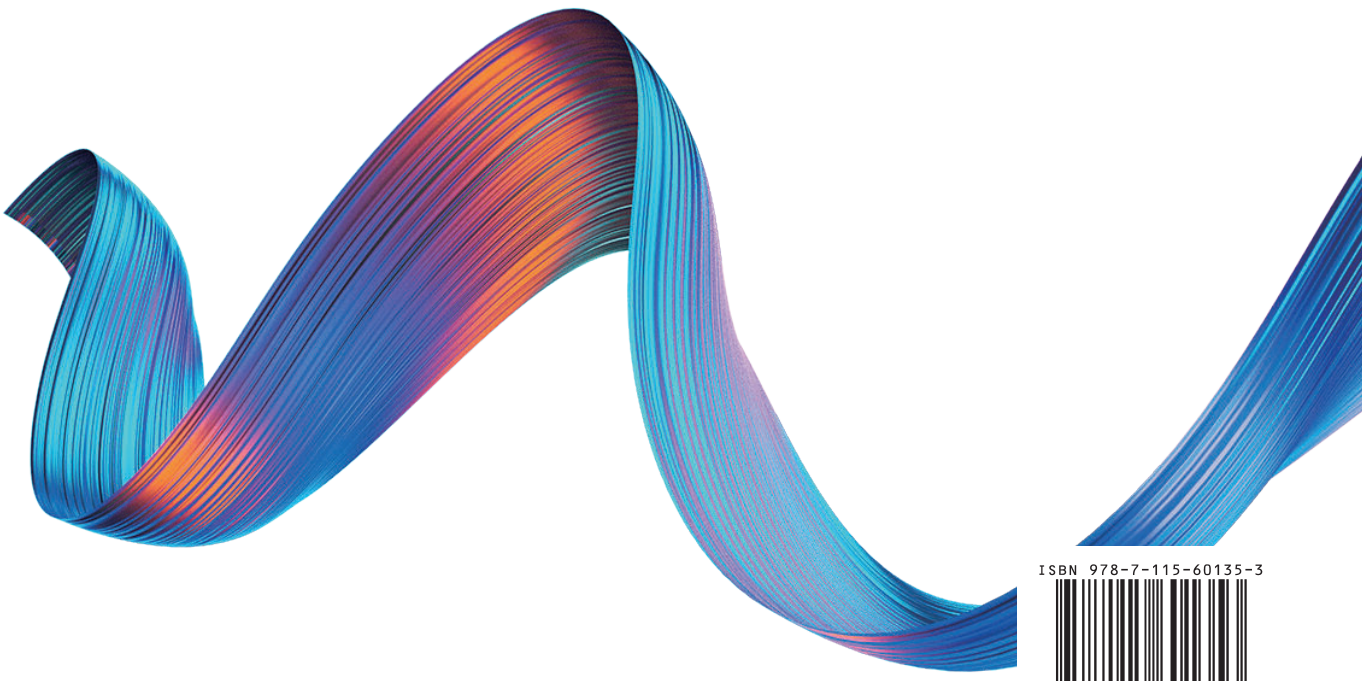
题目	难度	题目年份	组别
第八大奇迹	★★★	2019 年国赛	• C/C++ B 组第 9 题
递增三元组	★	2018 年省赛	• C/C++ B 组第 6 题 • Java B 组第 6 题
选数异或	★★★	2022 年省赛	• C/C++ A 组第 4 题；C/C++ C 组第 6 题 • Java C 组第 6 题
推导部分和	★★★	2022 年省赛	• C/C++ A 组第 10 题 • Java A 组第 10 题
最长不下降子序列	★★★	2022 年省赛	• C/C++ A 组第 7 题 • Java C 组第 10 题 • Python A 组第 8 题；Python B 组第 9 题

C/C++ 程序设计竞赛

真题实战特训教程 图解版

当前，全面建设社会主义现代化强国的航船已经启程，加快建设制造强国、网络强国、数字中国是其强劲的动力。人才是第一资源，青年是关键支撑。蓝桥杯大赛一直致力于我国的软件和信息技术人才培养和选拔工作，业已形成激励年轻才俊脱颖而出，以及政府支持、企业欢迎、社会鼓励的良好环境与氛围。蓝桥杯大赛系列丛书由专家老师撰写，深入浅出，精炼实用，不仅能帮助青年朋友们更好地学习和实践计算机相关技能，而且为愿意参与蓝桥杯活动的同学们设定了路标参考，一定能成为大家迈向数字中国的得力工具，更好地为我国社会主义现代化建设添砖加瓦。

——**朱宏任**（蓝桥杯大赛组委会主任，现任中国企业联合会 / 中国企业家协会党委书记、常务副会长兼秘书长，工业和信息化部原党组成员、总工程师）



ISBN 978-7-115-60135-3

封面设计：董志桢

分类建议：计算机 / 程序设计

人民邮电出版社网址：www.ptpress.com.cn