

N 诺考研系列

# 计算机考研机试攻略

满分篇



N 诺课程教研团队

2022. 04. 20 更新

## 写在前面的话

相信各位同学都知道，N 诺是一个大佬云集的平台。N 诺每周都会定期举办比赛，让同学们在日复一日的枯燥生活中找到一种乐趣。

本书是 N 诺专为计算机考研的同学精心准备的一本**神书**，为什么说它是一本神书，它到底神在什么地方？

- 1、这本书是 N 诺邀请众多 **CSP 大佬**、**ACM 大佬**、**BAT 专业大佬**以及往年**机试高分大佬**共同修订而成。
- 2、这本书与市面上的书都不一样，这是一本专门针对**计算机考研机试**而精心编写而成的书籍。
- 3、这本书上的**例题讲解**以及**习题**都可以在 N 诺上找到进行练习，并且每道题都有大佬们发布的**题解**可以学习。
- 4、N 诺有自己的官方群，群里大佬遍地走，遇到问题在群里可以随时提问。群里**神仙**很多，不怕没有问题，就怕问题太简单。

相信同学们一定听过各种各样的 OJ 平台，但是那些平台不是为了计算机考研而准备的，而 N 诺是**唯一**一个纯粹为计算机考研而准备的学习平台。

相信每一个同学在学习本书之前，都觉得机试是一个很头疼的问题，机试有可能**速成**吗？我可以一周就**变得很强**吗？我们可以在短短两三周之内的学习就能拿到**机试高分**吗？

别人可不可以我不知道，但是在 N 诺这里，你就可以。

虽然考研机试题目千千万，但是你可以“一招鲜，吃遍天”。

管它乱七八糟的排序，我就一个 sort 你能拿我怎么办？

管它查来查去的问题，我会 map 我怕谁？

管它飘来飘去的规律，我有 OEIS 神器坐着看你秀！

管它眼观缭乱的算法，我有 N 诺万能算法模板怕过谁？

本书虽然不能让你变成一个算法高手，但是本书可以让你快速变成一个机试高手，这也是本书最重要的特点，以解决同学们的实际需求为目的。你可以不会算法，你可以不必弄懂算法的原理，你只要学会本书教给你的各种技巧，就足以应对 99.9%的情况。

N 诺的课程教研团队全是由大佬组成，每个人都做过几千道编程题目，做编程题的经验十分丰富，我们分析近百所院校的历年机试真题，从中发现了各个学校的机试真题都有相同的规律，万变不离其中，于是本书也就诞生了。

计算机考研机试攻略交流群请扫描下方二维码或直接搜索群号：960036920



群名称：N诺 - 计算机机试交流群  
群号：960036920

本书配套精讲视频：<https://www.bilibili.com/video/av91373687>

## 关于 N 诺

N 诺是全国最大的计算机考研在线学习平台。N 诺致力于为同学们提供一个良好的网络学习环境，一个与大佬们随时随地交流的舒适空间。如果你不知道 N 诺，那么你已经输在起跑线上了，因为 N 诺是 - 计算机学习考研必备神器。

N 诺整理了**计算机考研报考指南**，帮助你了解考研的点点滴滴赢在起跑线上。

<http://www.noobdream.com/media/upload/2020/01/29/guide.pdf>

在 N 诺，你可以查询各个院校的**考研信息**，包括分数线、录取人数、考试大纲、导师信息、经验交流等信息，应有尽有。

<http://www.noobdream.com/schoollist/>

在 N 诺，你可以尽情的刷各个科目的**题库**，还可以将题目加入**错题本**方便以后复习，也可以写下自己的**学习笔记**，记录考研过程中跌宕起伏。

<http://www.noobdream.com/Practice/index/>

在 N 诺，你可以在**讨论区**里发表你的问题或感想，与全国几百万考研 er 分享你的喜怒哀乐。

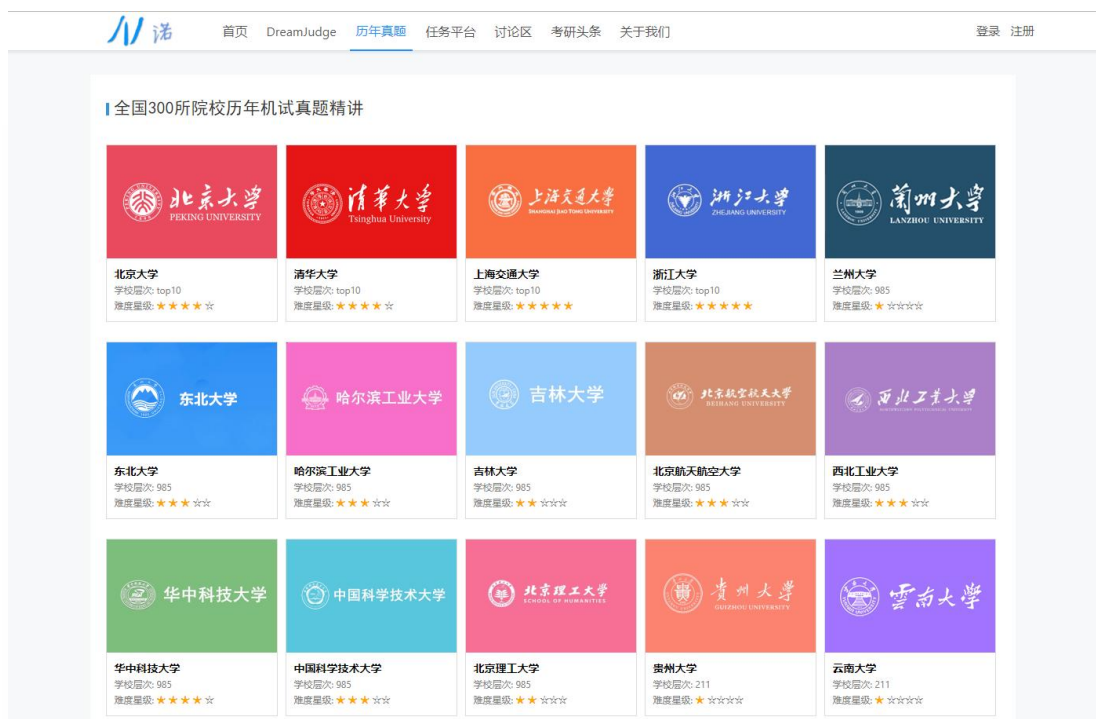
<http://www.noobdream.com/forum/0/>

在 N 诺，你可以将你不用**的书籍或资料**放到**二手交易市场**，既能帮助他人，还能为自己省下一笔当初买书买资料的费用。

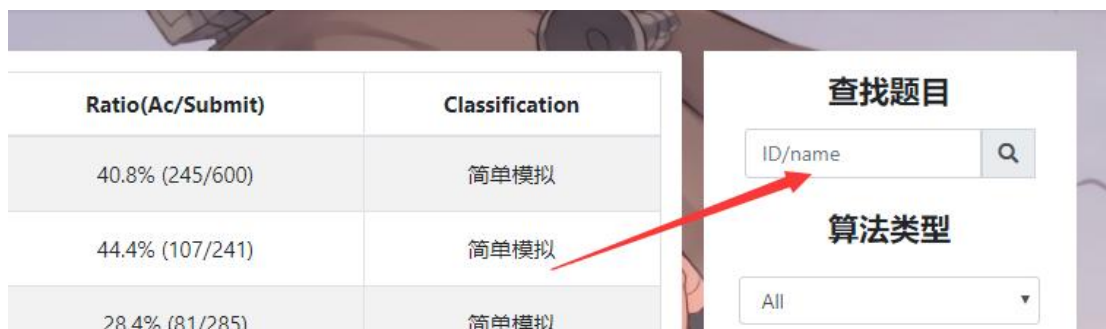
<http://www.noobdream.com/Task/tasklist/>

## 如何使用本书？

访问 N 诺平台（www.noobdream.com）的历年真题，即可查询到全国各个学校的历年机试真题。



如下图输入课后习题的编号即可搜索到题目



小技巧：N 诺是可以随意更换皮肤的哟，在右上角头像旁边。

## 目录

写在前面的话 .....	2
关于 N 诺 .....	4
如何使用本书? .....	5
目录 .....	6
第一章 技巧之巅 .....	7
1.1 输入输出加速外挂 .....	8
1.2 调试技巧 .....	10
1.3 位运算技巧 .....	12
1.4 考试最佳策略 .....	16
1.5 预处理与打表技巧 .....	18
1.6 对数器技巧 .....	21
第二章 满分之路上 .....	24
2.1 计算几何基础 .....	25
2.2 进阶背包问题 .....	32
2.3 毛毛虫算法 .....	36
2.4 博弈类问题 .....	40
2.5 二分答案技巧 .....	50
2.6 前缀和技巧 .....	53
2.7 双指针技巧 .....	56
第三章 满分之路中 .....	58
3.1 线段树单点更新 .....	59
3.2 线段树区间更新 .....	65
3.3 字符串匹配问题 .....	72
3.4 二分图的匹配问题 .....	79
3.5 kmp 算法的应用 .....	83
3.6 路径进阶问题 .....	84
3.7 图的连通性问题 .....	91
3.8 单调队列/栈 .....	95
第四章 满分之路下 .....	98
4.1 容斥与抽屉原理 .....	99
4.2 除法取模问题 .....	103
4.3 组合数取模类问题 .....	105
4.4 矩阵快速幂 .....	109
4.5 带状态压缩的搜索 .....	112
4.6 数位类型动态规划 .....	116
4.7 FFT 快速傅里叶变换 .....	120
4.8 机试押题 .....	125
完结撒花 .....	126
N 诺考研系列图书 .....	128
N 诺 Offer 训练营 .....	错误! 未定义书签。

# 第一章 技巧之巅

建议大家在阅读本书之前先阅读本书的姊妹篇：**计算机考研机试攻略 - 高分篇**。

这本书是高分篇的加强版，适合学习完高分篇的同学继续学习，拿到机试满分。

本章是 N 诺计算机考研机试攻略满分篇的第一章，本章我们带领大家领略技巧的极致魅力。本章包括：输入输出加速外挂、调试技巧、位运算技巧、考试最佳策略、预处理与打表技巧、对数器技巧等内容。可以帮助读者迅速掌握一些使用的考试技巧，完成更高难度的挑战。



本书配套视频精讲：<https://www.bilibili.com/video/av91373687>



## 1.1 输入输出加速外挂

有的时候题目的输入数据量比较大, 比如要输入 10W 和数字进行排序。这个时候, 如果你直接使用 C++ 的 `cin` 和 `cout` 函数进行输入输出, 有很大的概率会超出题目的时间限制。

在这种情况下, 需要的优化的就不再是你的算法过程, 而是你的读写数据的速度优化。如果使用 `cin` 和 `cout` 函数进行输入输出, 我们建议你在 `main()` 里首先写入下面两行代码。

### C++ `cin/cout` 加速

```
1. ios_base::sync_with_stdio(0);  
2. cin.tie(0); cout.tie(0);
```

如果题目的输入量巨大, 比如要输入 100W 个数字, 这个时候我们建议你使用 C 语言的 `scanf` 和 `printf` 语句进行输入输出。

如果在这种情况下还超时, 那么你就需要用到下面的输入输出加速外挂了。

```
1. //适用于正负整数  
2. template <class T>  
3. inline bool scan_d(T &ret) {  
4.     char c; int sgn;  
5.     if(c=getchar(),c==EOF) return 0; //EOF  
6.     while(c!='-'&&(c<'0' || c>'9')) c=getchar();  
7.     sgn=(c=='-')?-1:1;  
8.     ret=(c=='-')?0:(c-'0');  
9.     while(c=getchar(),c>='0'&&c<='9') ret=ret*10+(c-'0');  
10.    ret*=sgn;  
11.    return 1;  
12. }  
13. inline void out(int x) {  
14.     if(x>9) out(x/10);  
15.     putchar(x%10+'0');  
16. }
```



加速外挂原理:

getchar 的速度 快于 scanf 的速度

速度比较

getchar > scanf > cin

putchar > printf > cout

参考代码

```
1. // 求 1 + n 的和
2. #include <bits/stdc++.h>
3. using namespace std;
4.
5. //适用于正负整数
6. template <class T>
7. inline bool scan_d(T &ret) {
8.     char c; int sgn;
9.     if(c=getchar(),c==EOF) return 0; //EOF
10.    while(c!='-'&&(c<'0' || c>'9')) c=getchar();
11.    sgn=(c=='-')?-1:1;
12.    ret=(c=='-')?0:(c-'0');
13.    while(c=getchar(),c>='0'&&c<='9') ret=ret*10+(c-'0');
14.    ret*=sgn;
15.    return 1;
16. }
17. inline void out(int x) {
18.     if(x>9) out(x/10);
19.     putchar(x%10+'0');
20. }
21. // 请注意只有在大量输入或大量输出的时候才能看出时间的区别
22. int main() {
23.     int n;
24.     scan_d(n); //加速输入
25.     long long sum = 0;
26.     for (int i = 1; i <= n; i++)
27.         sum += i;
28.     out(sum); //加速输出
29.     return 0;
30. }
```

## 1.2 调试技巧

调试是我们在编写程序时不得不经历的过程, 我们这一节来讲解如何快速定位错误的调试技巧。

首先, 编译错误不在我们的讨论范围之内, 语法上的错误问题相信学过高分篇的同学都不会再问了。

其实, 断点调试也不在我们的讨论范围之内。

这里讲一下我们为什么不建议大家使用断点调试, 断点调试基本上是每个开发者都会的调试技巧。

但是我们的机试有一些特殊的情况。

- 1、机试的代码往往很短, 几行到几十行不等。
- 2、比赛中争分夺秒, 我们对调试时间要求更为迫切。
- 3、我们的错误往往是由于代码细节没考虑周全导致的。

所以, 断点调试更适于项目代码中且对时间的迫切度没有那么高的情况。

接下来, 给大家介绍一种超级棒的调试技巧, 当你熟练掌握它以后你会深深的迷恋上它。

## 输出调试

顾名思义，就是**通过输出的方式定位**我们的错误所在。

大部分没有足够调试经验的同学使用输出调试的时候，不知道应该如何使用输出调试。

如果从前往后逐条语句输出调试去排查错误，那么很容易要找很久。

反之从后往前逐条语句输出调试去排查错误，那么也很容易要找很久。

不知道同学们是不是发现了点什么，对，没错，上面两种方法是不是就是顺序查找的方法。

那么，与此对应的就应该是二分查找的方法。

使用**二分查找**的思想来调试定位错误，可以更快，更节约时间。

**特别注意：输出调试完成之后提交代码之前一定要删除或注释掉调试信息。**

## 1.3 位运算技巧

位运算，相信所有的同学都有所了解。

那么关于位运算有哪些特殊的技巧呢？

### 速度比较

取模时间 > 四则运算时间 > 位运算时间

所以对于一个语句

```
1. if (a % 2 == 1) {  
2.     a /= 2;  
3. }
```

我们可以优化成为

```
1. if (a & 1 == 1) {  
2.     a >>= 1;  
3. }
```

异或运算的特殊性

异或同一个数 2 次或者偶数次，那么本身的价值不变。

例如：

$$a \oplus b \oplus b = a$$

$$x \oplus y \oplus y \oplus y \oplus y = x$$

接下来我们来看一下它的应用。

### 缺页问题

题目描述：

小明有一本很喜欢的漫画书，但是由于他上课的时候偷看这本漫画书被老师发现，于是老师将

小明的漫画书撕了扔进垃圾桶。

小明放学后从垃圾桶里将漫画书捡了回来，却发现漫画书少了一页。小明的漫画书一共有  $N$  页，由于现在页码顺序错乱，小明也不知道是少了哪一页，聪明的你能告诉小明缺失的是哪一页吗？

#### 输入描述：

第一行输入一个整数  $N$  ( $N < 1000000$ )，表示小明书的总页数。

第二行输入  $N-1$  个数整数，代表小明找到的页码。

#### 输出描述：

请输出小明缺失的页码在单独的一行。

请注意：本题的空间很小，要求你尽量不使用额外的空间来解决。

#### 输入样例#：

6

6 2 1 5 3

#### 输出样例#：

4

#### 题目来源：

DreamJudge 1506

**题目解析：**由于本题要求我们以尽量小的空间来解决问题，所以我们不能够使用数组来存储每一个数。那么我们应该怎么办呢？这个时候可以想到异或运算的特殊技巧，同一个数异或两次那么就会消除，如果我们提前将 1 到  $N$  的所有数字进行异或处理，然后再去异或输入的  $N-1$  个数，那么答案就是缺失的那个数。

#### 参考代码

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

```
5.     int n, x;
6.     scanf("%d", &n);
7.     int sum = 0;
8.     for (int i = 1; i <= n; i++) {
9.         sum ^= i;
10.    }
11.    for (int i = 1; i < n; i++) {
12.        scanf("%d", &x);
13.        sum ^= x;
14.    }
15.    printf("%d\n", sum);
16.    return 0;
17. }
```

## 常见位运算问题

### 1. 位操作实现乘除法

数  $a$  向右移一位, 相当于将  $a$  除以 2; 数  $a$  向左移一位, 相当于将  $a$  乘以 2

```
1.  int a = 2;
2.  a >> 1; ---> 1
3.  a << 1; ---> 4
```

### 2. 取相反数

思路就是取反并加 1, 也即  $\sim n + 1$  或者  $(n \wedge -1) + 1$ 。

### 3. 判断奇偶性

```
1.  /* 判断是否是奇数 */
2.  bool is_odd(int n)
3.  {
4.      return (n & 1 == 1);
5.  }
```

#### 4. 不用临时变量交换两个数

```
1. a ^= b;  
2. b ^= a; // 相当于 b = b ^ ( a ^ b );  
3. a ^= b;
```

#### 5. 统计二进制中 1 的个数

```
1. count = 0  
2. while(a){  
3.     a = a & (a - 1);  
4.     count++;  
5. }
```

#### 题目练习

DreamJudge 1118 将军的书

noobdream.com



## 1.4 考试最佳策略

很多上机考试经验不够丰富的同学，往往很难发挥出自己应有的水平。

下面我们来讲一下如何应对一场考研的上机考试。

### 提前准备

- 1、算法模板一定要提前准备好，不管是单独打印还是记录在书上。
- 2、算法模板一定要验证它的正确性，在验证的过程中也知道了该如何使用。
- 3、手机调成静音模式，最好再开一个飞行模式，不要上交以备不时之需。

### 开考前

- 1、验证鼠标键盘是否可用，如果不可用及时向老师反应，更换电脑。
- 2、验证 IDE（如：codeblocks）是否可用，如果不可用及时向老师反应，更换电脑。
- 3、提前将头文件和主函数框架写好，最好创建两个文件，代码复制一份就行。因为在做题过程中遇到卡题情况，可以及时切换到另一道题上继续写。

### 考试中

- 1、机试中的题目难度不是从简单到难，难度是随机的，所以刚开始一定要将所有题目都看一遍。
- 2、找到你认为最简的那道题开始做，记住，一定要从最简单的题目开始做。
- 3、考试过程中要注意看排行榜，通过人数最多的题目一般都是最简单的题目。
- 4、注意：要看通过人数的多少来判断难易程度，而不是第一个人通过的时间来判断。
- 5、如果你的水平强，可以选择先做那种代码不长但是需要算法思维的题，快速解出来将排行榜带偏，给其他人一种这题最简单的感觉，然后让他们死磕这题，成功让竞争对手翻车。
- 6、之所以要赤裸裸的将上一条写出来，就是为了告诫那些头铁的同学，不要在一棵树上吊死，都走复试这一步了，还这么死脑筋基本没得救了，
- 7、考试的时候遇到规律题，千万不要忘记 OEIS 这个神器，特别是周围上厕所的同学开始变多。
- 8、当无法通过一个题的时候，先看看有没有其他能做的题，如果也没有其他题能做了。

这个时候你就可以使用很多特殊的办法，首先，你一定要相信，机试的数据一定不够强。一般情况下，机试的判题数据都会找学生帮忙生成，往往强度就不会很高，这个要看学生的尽责程度。如果是老师自己构造的数据，你睡着都会笑醒。

9、不要被题目的数据范围吓到，有可能后台都是小数据，没有更好的解法的时候一定要试试暴力。

10、我们往往被卡都是因为算法不够优秀导致超时，一方面我们可以强行优化输入输出加速来看看能不能水过去。另一方面可以采用小数据暴力，大数据随机的思想来解决问题。

### 小数据暴力、大数据随机

原理：由于大数据出多组容易导致判题很慢，所以往往不会有很多组。另外对于特殊的数据可能需要手动构造，大数据构造起来麻烦，还要自己构思生成数据的代码。所以一般都是用小数据来验证算法正确性，再加上两组大数据组验证算法复杂度。

### 举例说明

#### 背包问题

我们背包问题一般使用动态规划来解，但是你不会怎么办？

那么我们就可以小数据暴力搜索，大数据直接贪心。对于数据不够强的题就能水过去。

### 更简单的例子

给你很多个数，要你从中找出最大的数。

你需要从第一个找到最后一个才能判断出最大的数是哪个吗？

如果是这样几个数：5 2 7 1 6 3

我们是不是只需要找到一半就能找到最大那个数了

如果是这样几个数：8 2 7 1 6 3

我们是不是只需要找到第一个就能找到最大那个数了

一般情况下，判题数据就像上面那样，你可以通过不严谨的代码通过题目。

**主要是领悟思想，不同的题都有不同策略。**

## 1.5 预处理与打表技巧

### 预处理

预处理是指将答案提前处理好然后再进行查询的方法。

什么时候会用到预处理？


查询量特别大的时候。

### 例题

我们要查询斐波那契数列的第  $n$  项,  $n$  ( $n < 10000$ )。

并且我们要查询 10 万次。

### 错误的超时解法



```
1. #include<bits/stdc++.h>
2. using namespace std;
3.
4. int f[10005];
5. int main(){
6.     int t;
7.     scanf("%d", &t); // 查询次数
8.     while (t--) {
9.         int n;
10.        scanf("%d", &n); // 查询第 k 项
11.        f[1] = 1;
12.        f[2] = 1;
13.        for (int i = 3; i <= n; i++) {
14.            f[i] = f[i-1] + f[i-2];
15.        }
16.        printf("%d\n", f[n]);
17.    }
18.    return 0;
19. }
```

**超时分析：**每一次查询，我们都递推了一遍斐波那契数列，如果每一次查询的都是最后一项。那么最坏情况就是：100000\*10000，必然是会超时的。

### 正确的预处理解法

```
1. #include<bits/stdc++.h>
2. using namespace std;
3.
4. int f[10005];
5. int main(){
6.     f[1] = 1;
7.     f[2] = 1;
8.     for (int i = 3; i <= 10000; i++) {
9.         f[i] = f[i-1] + f[i-2];
10.    }
11.    int t;
12.    scanf("%d", &t); // 查询次数
13.    while (t--) {
14.        int n;
15.        scanf("%d", &n); // 查询第 k 项
16.        printf("%d\n", f[n]);
17.    }
18.    return 0;
19. }
```

可以发现两段代码惊人的相似，有什么区别呢？

区别在于下面的代码只会递推一遍斐波那契数列，然后将所有的答案都存储于 f 数组中。

## 一般打表技巧

打表是指我们提前使用暴力的方法，将答案记录下来写到纸上或代码中，然后再根据题目的实际需求去直接输出提前记录好的答案。

### 例题

求一个数  $x$  的  $100000000$  次方模  $2333$  的值是多少。

$x$  的范围是 ( $1 \leq x \leq 10$ )

**解析：**这个时候我们如果不会二分快速幂的话，可以直接用一个暴力的代码将  $1$  到  $10$  的答案分别求出来，可能要等上几分钟，然后直接判断输入的值对应输出结果即可。

## 打表找规律

在一些可能存在规律的问题中，我们可以通过暴力打表求出前几十项的值，发现其中的规律，从而帮助我们解出该题。

### 例题

输入一个整数  $a$  ( $a < 10^9$ ),  $b$  可以取任意值，求  $a \% b$  的可能产生多少种不同的结果。

**解析：**乍一看，我们一脸懵逼。这个时候你只要去打表分析，首先  $b$  大于  $a$  就重复了，那么我们只需要对于任意一个  $a$ ，去枚举  $b$  的值，看  $a \% b$  会产生多少种不同的结果即可。通过打表分析，你会发现：当  $a = 1$  时，结果  $ans = 1$ ，当  $a$  为偶数时， $ans = a / 2 + 1$ ，当  $a$  为奇数时， $ans = a / 2 + 2$ 。

## 题目练习

DreamJudge 1488 数字填充

## 1.6 对数器技巧

大多数同学应该都有过这样的经历...

这道题我做了半天怎么还是错的啊, 看了别人正确的代码跟我的差不多啊, 我到底错在哪里了啊? 抓狂...

然后开始疯狂 diss 管理员, 管理员在吗? 你这个小破站, 为什么不能告诉我是哪一组数据出错了? 你有没有考虑过我们这种新手的感受? 你看看别人某站, 错误还会给我反馈错误的数  
据, 你们是不是技术不行啊?

管理员的无奈: 大部分题目的极限数据都是上千上万行, 反馈给你也没用的, 你要花大量的时间去查找去比较, 还不如再认真检查一下代码。

但是, 当你学会这一节的内容之后, 这一切的烦恼都不存在了。

使用对数器进行对拍, 可以快速的找到的你的代码错误所在。

简单说, 就是将你的代码和正确的代码放在一起, 然后随机生成一些符合题目的要求的数据, 然后比较你的代码的输出结果和正确代码的输出结果是否一致, 从而来验证你的代码的正确性。

当然, 随机的数据越多, 正确性也就越高。

下面我们举个排序的例子

我自己写了一个排序的算法, 但是我不知道我写的这个排序算法到底有没有问题, 但是我拿到别人的正确的排序的算法代码, 我如何比较呢?

我的排序算法代码

```
1. for (int i = 1; i <= n; i++)  
2.     for (int j = 1; j < n; j++)  
3.         if (b[j] > b[j+1])  
4.             swap(b[j], b[j+1]);
```

正确的排序算法代码

```
1. sort(c+1, c+n+1);
```

下面就是完整的对数器使用的模板

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. struct node {
5.     int a[105]; //随机的数
6.     int b[105]; //我的方法排序后的数
7.     int c[105]; //正确的方法排序后的数
8.     int n;
9.     // 按照题意随机一些输入数据
10. void Rand() {
11.     n = rand()%100 + 1;
12.     for (int i = 1; i <= n; i++)
13.         a[i] = rand()%10000;
14. }
15. // 使用我的解法得到的答案
16. void My_method() {
17.     for (int i = 1; i <= n; i++)
18.         b[i] = a[i];
19.     for (int i = 1; i <= n; i++)
20.         for (int j = 1; j < n; j++)
21.             if (b[j] > b[j+1])
22.                 swap(b[j], b[j+1]);
23. }
24. // 使用其他正确的解法得到的答案
25. void Right_method() {
26.     for (int i = 1; i <= n; i++)
27.         c[i] = a[i];
28.     sort(c+1, c+n+1);
29. }
30. // 比较我的答案和正确答案是否一致
31. // 如果不一致输出测试数据然后分析
32. int Check() {
33.     for (int i = 1; i <= n; i++) {
34.         if (b[i] != c[i]) {
35.             printf("my code is error!\n");
36.             printf("the test data:\n");
37.             for (int j = 1; j <= n; j++) {
38.                 printf("%d ", a[j]);
39.             }
40.             printf("\n");
```



```
41.         return 1;
42.     }
43. }
44.     return 0;
45. }
46. };
47.
48.
49. int main() {
50.     srand(time(NULL)); //将时间作为随机种子
51.     int t = 100; //样本大小
52.     while (t--) {
53.         node code;
54.         code.Rand();
55.         code.My_method();
56.         code.Right_method();
57.         if (code.Check()) return 0;
58.     }
59.     printf("my code is right!\n");
60. }
```

代码分析：首先随机 100 组输入数据，然后使用自己的算法去跑出结果，然后使用正确的算法去跑出结果，最后比较两个结果是否一致，如果不一致就输出是哪一组数据导致的结果不一致，最后分析自己的算法问题所在。

## 第二章 满分之路上

高分篇的内容已经将机试可能涉及到的题型及考点都一一列举出来, 并给大家提供了练习题目, 帮助大家巩固基础加深理解。在大多数院校的机试中, 学会高分篇的内容已经足够考到 90 分以上的成绩, 题目不难的情况下甚至能拿到 100 分。在少数难度较大的院校也足够拿到 80 分左右的分数, 如果你基础比较薄弱, 我们建议你掌握高分篇的内容就可以了。本书满分之路的内容不是常考点, 只是有一定可能会考到, 我们建议基础好的同学继续学习增加拿满分的把握。

距离满分只有一步之遥, 你! 愿意放弃吗?

本章我们重点讲解一些较为深入的题型, 包括计算几何基础、进阶背包问题、毛毛虫算法、博弈类问题、二分答案技巧和前缀和技巧等内容。希望能帮助读者更好的掌握计算机考研机试中所涉及到的各类较难的问题。

**提示: 满分篇的内容选择性的学就行, 不要求全部掌握, 会其中一部分就很厉害了。**

本书配套视频精讲: <https://www.bilibili.com/video/av91373687>

## 2.1 计算几何基础

在考研机试中, 有的时候会考到一些基础的计算几何知识, 其中包括点积、叉积、凸包等内容。

首先在二维坐标下介绍一些定义:

点:  $A(x_1, y_1)$ ,  $B(x_2, y_2)$

向量: 向量  $AB = (x_2 - x_1, y_2 - y_1) = (x, y)$

向量的模  $|AB| = \sqrt{x^2 + y^2}$

### 点积

向量的点积: 结果为  $x_1 * x_2 + y_1 * y_2$ 。

点积的结果是一个数值。

点积的集合意义: 我们以向量  $a$  向向量  $b$  做垂线, 则  $|a| * \cos(a, b)$  为  $a$  在向量  $b$  上的投影, 即点积是一个向量在另一个向量上的投影乘以另一个向量, 且满足交换律。

应用: 可以根据集合意义求两向量的夹角,  $\cos(a, b) = (\text{向量 } a * \text{向量 } b) / (|a| * |b|) = x_1 * x_2 + y_1 * y_2 / (|a| * |b|)$

noobdream.com

### 叉积

向量的叉积: 结果为  $x_1 * y_2 - x_2 * y_1$

叉积的结果也是一个向量, 是垂直于向量  $a, b$  所形成的平面, 如果看成三维坐标的话是在  $z$  轴上, 上面结果是它的模。

方向判定: 右手定则, (右手半握, 大拇指垂直向上, 四指右向量  $a$  握向  $b$ , 大拇指的方向就是叉积的方向)

叉积的集合意义:

- 1、其结果是  $a$  和  $b$  为相邻边形成平行四边形的面积。
- 2、结果有正有负, 有  $\sin(a, b)$  可知和其夹角有关, 夹角大于  $180^\circ$  为负值。
- 3、叉积不满足交换律

应用:

1、通过结果的正负判断两矢量之间的顺逆时针关系

若  $a \times b > 0$  表示  $a$  在  $b$  的顺时针方向上

若  $a \times b < 0$  表示  $a$  在  $b$  的逆时针方向上

若  $a \times b = 0$  表示  $a$  在  $b$  共线，但不确定方向是否相同

2、判断折线拐向，可转化为判断第三点在前两的形成直线的顺逆时针方向，然后判断拐向。

3、判断一个点在一条直线的那一侧，同样上面的方法。

4、判断点是否在线段上，可利用叉乘首先判断是否共线，然后在判断是否在其上。

5、判断两条直线是否想交（跨立实验）

根据判断点在直线那一侧我们可以判断一个线段的上两点分别在另一个线段的两侧，当然这是不够的，因为我们画图发现这样只能够让直线想交，而不是线段，所以我们还要对另一条线段也进行相同的判断就 ok。

## 凸包

### 凸多边形

凸多边形是指所有内角大小都在  $[0, \pi]$  范围内的 简单多边形 。

### 凸包

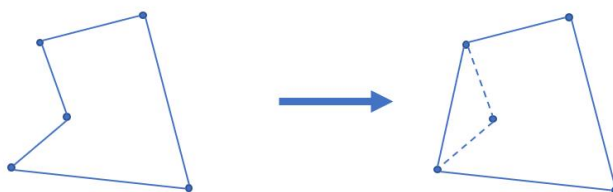
在平面上能包含所有给定点的最小凸多边形叫做凸包。

其定义为：

对于给定集合  $X$ ，所有包含  $X$  的凸集的交集  $S$  被称为  $X$  的凸包 。

实际上可以理解为用一个橡皮筋包含住所有给定点的形态。

凸包用最小的周长围住了给定的所有点。如果一个凹多边形围住了所有的点，它的周长一定不是最小，如下图。根据三角不等式，凸多边形在周长上一定是最优的。



## 凸包的求法

我们这里不详细讲解与证明凸包算法的原理, 其本质就是极角坐标排序, 我们主要是告诉大家如何使用凸包算法的模板。

### 凸包面积

#### 题目描述:

麦兜是个淘气的孩子。一天, 他在玩钢笔的时候把墨水洒在了白色的墙上。再过一会, 麦兜妈就要回来了, 麦兜为了不让妈妈知道这件事情, 就想用一个白色的凸多边形把墙上的墨点盖住。你能告诉麦兜最小需要面积多大的凸多边形才能把这些墨点盖住吗? 现在, 给出了这些墨点的坐标, 请帮助麦兜计算出覆盖这些墨点的最小凸多边形的面积。

#### 输入描述:

多组测试数据。第一行是一个整数  $T$ , 表明一共有  $T$  组测试数据。 每组测试数据的第一行是一个正整数  $N$  ( $0 < N \leq 105$ ), 表明了墨点的数量。接下来的  $N$  行每行包含了两个整数  $X_i$  和  $Y_i$  ( $0 \leq X_i, Y_i \leq 2000$ ), 表示每个墨点的坐标。每行的坐标间可能包含多个空格。

#### 输出描述:

每行输出一组测试数据的结果, 只需输出最小凸多边形的面积。面积是个实数, 小数点后面保留一位即可, 不需要多余的空格。

#### 输入样例#:

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

#### 输出样例#:

```
1.0
```

0.0

题目来源:

DreamJudge 1113

解题分析: 使用求凸包面积的算法模板即可。

参考代码

```
1. #include <bits/stdc++.h>
2. #include <math.h>
3. using namespace std;
4. const int maxn=1005;
5. inline double sqr(double x){return x*x;}
6. struct point{
7.     double x,y;
8.     point(){}
9.     point(double _x,double _y):x(_x),y(_y){}
10.    //判断两点相同
11.    bool operator ==(const point &b) const{
12.        return x-b.x==0 && y-b.y==0;
13.    }
14.    point operator -(const point &b) const{
15.        return point(x-b.x,y-b.y);
16.    }
17.    //叉积
18.    int operator ^(const point &b) const{
19.        return x*b.y-y*b.x;
20.    }
21.    //点积
22.    int operator *(const point &b) const{
23.        return x*b.x+y*b.y;
24.    }
25.    //重载小于号 求最左下角的点
26.    bool operator <(const point &b) const{
27.        return x-b.x==0?y-b.y<0:x<b.x;
28.    }
29. };
30. struct line{
31.     point s,e;
32.     line(){}
}
```

```

33.     line(point _s,point _e):s(_s),e(_e){}
34. };
35. double dis(point a,point b){ //求两点之间的距离
36.     return sqrt(1.0*sqr(a.x-b.x)+1.0*sqr(a.y-b.y));
37. }
38. struct polygon{
39.     int n;
40.     point p[maxn];
41.     void add(point q){p[n++]=q;}
42.     struct cmp{
43.         point p;
44.         cmp(const point &p0):p(p0){}
45.         bool operator()(const point &aa,const point &bb){
46.             point a=aa,b=bb;
47.             int k=(a-p)^(b-p);
48.             if (k==0){
49.                 return dis(a,p)-dis(b,p)<0;
50.             }
51.             return k>0;
52.         }
53.     };
54.     //极角排序 先找到左下角的点
55.     //重载好 point 的 '<'
56.     void norm(){
57.         point mi=p[0];
58.         for (int i=1;i<n;i++) mi=min(mi,p[i]);
59.         sort(p,p+n,cmp(mi));
60.     }
61.     //得到凸包, 点编号为 0--n-1
62.     void Graham(polygon &convex){
63.         norm();
64.         int &top=convex.n;
65.         top=0;
66.         if (n==1){
67.             top=1; convex.p[0]=p[0]; return ;
68.         }
69.         if (n==2){
70.             top=2; convex.p[0]=p[0]; convex.p[1]=p[1];
71.             if (convex.p[0]==convex.p[1]) top--;
72.             return ;
73.         }
74.         convex.p[0]=p[0]; convex.p[1]=p[1]; top=2;
75.         for (int i=2;i<n;i++){

```



```

76.         while (top>1 && ((convex.p[top-1]-convex.p[top-2])^(p[i]-convex.p[top-2]))<=0)
            top--;
77.         convex.p[top++]=p[i];
78.     }
79.     if (convex.n==2 && (convex.p[0]==convex.p[1])) convex.n--;
80. }
81. double getarea(){//求凸包的面积
82.     double sum=0;
83.     for (int i=0;i<n;i++) sum+=(p[i]^(p[(i+1)%n]));
84.     return sum/2.0;
85. }
86. };
87. polygon C;
88. int main(){
89.     int t;
90.     cin >> t;
91.     while (t--) {
92.         int n,x,y; cin >> n; C.n=0;
93.         for (int i=0;i<n;i++){
94.             cin >> x >> y;
95.             C.add(point(x,y));
96.         }
97.         polygon ans; C.Graham(ans);
98.         double res = ans.getarea();
99.         printf("%.11f\n", res);
100.    }
101.    return 0;
102. }

```

总结：对于机试中的计算几何来说，记住上面的算法模板并知道是如何使用的即可。

### 简单的变形应用

例：一片森林里有很多树，每棵树对应二维平面上的一个坐标，现在小明想要在森林里放牛，所以小明用线缠绕在树上划出牛栏的边界，已知每头牛都需要 50 单位面积的活动空间，请问小明最多可以放多少头牛？

解析：很明显，将树抽象成二维平面上的点，然后画一个凸多边形就是最大的面积，其实就是

求凸包的面积，将凸包面积/50 即为可以放牛的数量。

### 题目练习

DreamJudge 1160 球的半径和体积

DreamJudge 1183 Freckles

DreamJudge 1211 矩形相交

DreamJudge 1596 球形空间产生器

DreamJudge 1615 多边形的面积

DreamJudge 1620 放牧

DreamJudge 1621 玩具



## 2.2 进阶背包问题

在高分篇中，我们学习了简单背包问题，和 01 背包问题。

当时我们用的二维数组来进行递推的，仔细思考就可以发现其实我们没有必要开这么大的数组。因为我们所有的递推都是从上一行推到下一行的，也就是说，我们只需要开两行的数组即可完成这个过程。我们还可以进一步压缩空间，前半段和后半段也可以进行滚动。

### 滚动数组（上下滚动）

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     int dp[2][1005] = {0}; //只需要 0 和 1 进行滚动
6.     int w[1005];
7.     int s, n;
8.     while (scanf("%d%d", &s, &n) != EOF) {
9.         for (int i = 1; i <= n; i++)
10.             scanf("%d", &w[i]);
11.         memset(dp, 0, sizeof(dp));
12.         dp[0][0] = 1;
13.         for (int i = 1; i <= n; i++) {
14.             int now = i & 1;
15.             for (int j = s; j >= 0; j--) { //将 i-1 变成 i^1
16.                 if (dp[now^1][j] == 1) dp[now][j] = 1;
17.                 if (j - w[i] >= 0 && dp[now^1][j-w[i]] == 1) dp[now][j] = 1;
18.             }
19.         }
20.         if (dp[n&1][s] == 1) printf("YES\n");
21.         else printf("NO\n");
22.     }
23.     return 0;
24. }
```

### 滚动数组（前后滚动）

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
```

```

5.     int dp[1005] = {0}; //前后滚动
6.     int w[1005];
7.     int s, n;
8.     while (scanf("%d%d", &s, &n) != EOF) {
9.         for (int i = 1; i <= n; i++)
10.            scanf("%d", &w[i]);
11.        memset(dp, 0, sizeof(dp));
12.        dp[0] = 1;
13.        for (int i = 1; i <= n; i++) {
14.            for (int j = s; j >= 0; j--) { //从后往前枚举
15.                if (j - w[i] >= 0 && dp[j-w[i]] == 1) dp[j] = 1;
16.            }
17.        }
18.        if (dp[s] == 1) printf("YES\n");
19.        else printf("NO\n");
20.    }
21.    return 0;
22. }

```

## 完全背包

有  $N$  种物品和一个容量为  $V$  的背包，每种物品都有无限件可用。第  $i$  种物品的费用是  $c[i]$ ，价值是  $w[i]$ 。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

$$F[i][j] = \begin{cases} F[i-1][j] & , \quad \text{不放第} i \text{种物品} \\ F[i][j - C[i]] + W[i] & , \quad j \geq C[i] \text{至少放一件第} i \text{种物品} \end{cases}$$

## 完全背包问题

### 题目描述:

有  $n$  种（每一种有无数个）重量和价值分别为  $W_i, V_i$  的物品，现从这些物品中挑选出总量不超过  $W$  的物品，求所有方案中价值总和的最大值。

### 输入描述#:

输入包含多组测试用例，每一例的开头为两位整数  $n, W$  ( $1 \leq n \leq 10000, 1 \leq W \leq 1000$ )，接下来有  $n$  行，每一行有两位整数  $W_i, V_i$  ( $1 \leq W_i \leq 10000, 1 \leq V_i \leq 100$ )

### 输出描述#:

输出为一行，即所有方案中价值总和的最大值。

输入样例#:

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

输出样例#:

```
10
7
```

题目来源:

DreamJudge 1569

题目解析：这个题是一个完全背包的算法模板题，直接套用完全背包的模板即可。

参考代码

```
1. #include<bits/stdc++.h>
2. using namespace std;
3.
4. int n,W,v[10005],w[10005],dp[1005];
5. int main(){
6.     while(cin>>n>>W){
7.         for(int i=0;i<n;i++){
8.             cin>>w[i]>>v[i];
9.             memset(dp,0,sizeof(dp));
10.            for(int i=0;i<n;i++) //种类
11.                for(int j=w[i];j<=W;j+=w[i]) //重量从小到大枚举
12.                    dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
13.            cout<<dp[W]<<endl;
14.        }
```

```
15.     return 0;  
16. }
```

**特别说明：**多重背包的问题几乎没有学校机试会考，如果学有余力的同学可以尝试自己去学习一下。暴力的做法几乎人人都会，一般如果考到都需要优化。一种方法是类似二分快速幂的思想用二进制方式拆分求解，另一种方法是用单调队列来维护。



## 2.3 毛毛虫算法

毛毛虫算法（官方称为尺取法）通常是指对数组保存一对下标（起点、终点），然后根据实际情况交替推进两个端点直到得出答案的方法，这种操作很像是毛毛虫（尺取虫）爬行的方式故得名。

### Subsequence

#### 题目描述：

给出了  $N$  个正整数 ( $10 < N < 100\ 000$ ) 的序列，每个正整数均小于或等于 10000，并给出了一个正整数  $S$  ( $S < 100\ 000\ 000$ )。编写程序以查找序列中连续元素的子序列的最小长度，其总和大于或等于  $S$ 。

#### 输入描述：

第一行是测试用例的数量。对于每个测试用例，程序必须从第一行读取数字  $N$  和  $S$ ，并以一个间隔分隔开。序列号在测试用例的第二行中给出，以间隔分隔。输入将以文件结尾结束。

#### 输出描述：

对于每种情况，程序都必须将结果打印在输出文件的单独一行上。如果没有答案，则打印 0。

#### 输入样例#：

```
2
10 15
5 1 3 5 10 7 4 9 2 8
5 11
1 2 3 4 5
```

#### 输出样例#：

```
2
3
```

#### 题目来源：

DreamJudge 1570

题目解析：这是一个典型的毛毛虫算法的应用实例，下面详细讲解一下算法过程。

我们设以  $a_s$  开始总和最初大于  $S$  时的连续子序列为  $a_s + \dots + a_{t-1}$ ，这时



$$a_{s+1} + \dots + a_{t-2} < a_s + \dots + a_{t-2} < S$$

所以从  $a_{s+1}$  开始总和最初超过  $S$  的连续子序列如果是  $a_{s+1} + \dots + a_{t'-1}$  的话, 则必然有  $t \leq t'$ 。利用这一性质便可以设计出如下算法:

- (1) 以  $s = t = \text{sum} = 0$  初始化。
- (2) 只要依然有  $\text{sum} < S$ , 就不断将  $\text{sum}$  增加  $a_t$ , 并将  $t$  增加 1。
- (3) 如果(2)中无法满足  $\text{sum} \geq S$  则终止。否则的话, 更新  $\text{res} = \min(\text{res}, t-s)$ 。
- (4) 将  $\text{sum}$  减去  $a_s$ ,  $s$  增加 1 然后回到(2)。

### 参考代码

```

1. #include<bits/stdc++.h>
2. using namespace std;
3.
4. int a[100005];
5. int main() {
6.     int T, n, s;
7.     scanf("%d", &T);
8.     while(T--) {
9.         scanf("%d%d", &n, &s);
10.        for(int i = 1; i <= n; i++)
11.            scanf("%d", &a[i]);
12.        int l = 1, r = 1, sum = 0, ans = 0x3f3f3f3f;
13.        while(1) {
14.            while(sum < s && r <= n) sum += a[r++];
15.            if(sum < s) break;
16.            ans = min(ans, r-l);
17.            sum -= a[l++];
18.        }
19.        if(ans == 0x3f3f3f3f) printf("0\n");
20.        else printf("%d\n", ans);
21.    }
22.    return 0;
23. }
```

### 题目练习

DreamJudge 1591 逛画展

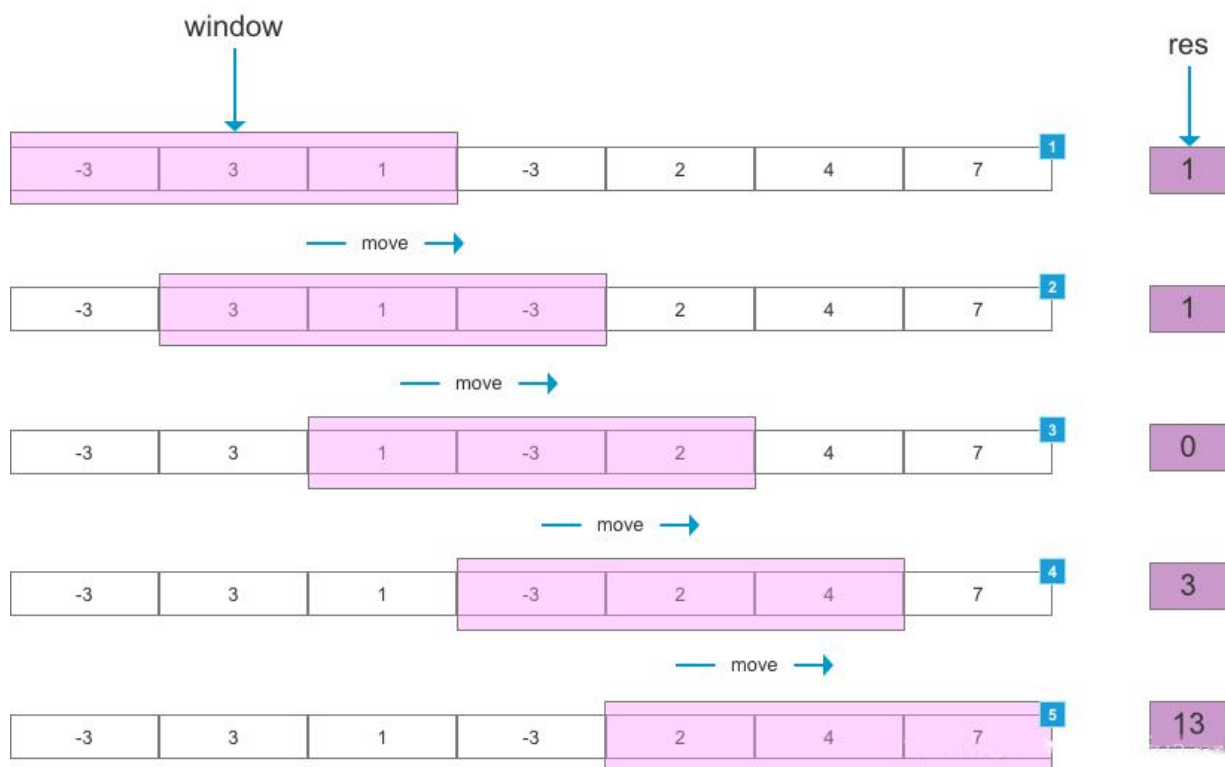
## 滑动窗口算法

毛毛虫算法有时候又叫做滑动窗口算法，我们可以换个方式来理解这类解题思想。

学过计算机网络的同学，都知道滑动窗口协议（Sliding Window Protocol），该协议是 TCP 协议 的一种应用，用于网络数据传输时的流量控制，以避免拥塞的发生。该协议允许发送方在停止并等待确认前发送多个数据分组。由于发送方不必每发一个分组就停下来等待确认。因此该协议可以加速数据的传输，提高网络吞吐量。

滑动窗口算法其实和这个是一样的，只是用的地方场景不一样，可以根据需要调整窗口的大小，有时也可以是固定窗口大小。

滑动窗口算法是在给定特定窗口大小的数组或字符串上执行要求的操作。该技术可以将一部分问题中的嵌套循环转变为一个单循环，因此它可以减少时间复杂度。简而言之，滑动窗口算法在一个特定大小的字符串或数组上进行操作，而不在整个字符串和数组上操作，这样就降低了问题的复杂度，从而也达到降低了循环的嵌套深度。其实这里就可以看出来滑动窗口主要应用在**数组**和**字符串**上。



可以用来解决一些查找满足一定条件的连续区间的性质（长度等）的问题。由于区间连续，因此当区间发生变化时，可以通过旧有的计算结果对搜索空间进行剪枝，这样便减少了重复计算，降低了时间复杂度。往往类似于“请找到满足  $xx$  的最  $x$  的区间（子串、子数组）的  $xx$ ”这类问题都可以使用该方法进行解决。

需要注意的是，滑动窗口算法更多的是一种思想，而非某种数据结构的使用。

## 实战例子

### 例题：尽可能使字符串相等

给你两个长度相同的字符串， $s$  和  $t$ 。

将  $s$  中的第  $i$  个字符变到  $t$  中的第  $i$  个字符需要  $|s[i] - t[i]|$  的开销（开销可能为 0），也就是两个字符的 ASCII 码值的差的绝对值。

用于变更字符串的最大预算是  $maxCost$ 。在转化字符串时，总开销应当小于等于该预算，这也意味着字符串的转化可能是不完全的。

如果你可以将  $s$  的子字符串转化为它在  $t$  中对应的子字符串，则返回可以转化的最大长度。如果  $s$  中没有子字符串可以转化成  $t$  中对应的子字符串，则返回 0。

示例 1：

输入： $s = "abcd"$ ， $t = "bcdf"$ ， $cost = 3$

输出：3

解释： $s$  中的  $"abc"$  可以变为  $"bcd"$ 。开销为 3，所以最大长度为 3。

示例 2：

输入： $s = "abcd"$ ， $t = "cdef"$ ， $cost = 3$

输出：1

解释： $s$  中的任一字符要想变成  $t$  中对应的字符，其开销都是 2。因此，最大长度为 1。

示例 3：

输入： $s = "abcd"$ ， $t = "acde"$ ， $cost = 0$

输出：1

解释：你无法作出任何改动，所以最大长度为 1。

## 2.4 博弈类问题

我们把机试中会考到的博弈问题分为了下面四类

- 1、简单博弈问题
- 2、巴什博弈
- 3、尼姆博弈
- 4、威佐夫博弈

博弈类的题目大多都是考察同学们推理和观察能力，大多数题目背景都是两个人做游戏，并且两个人都足够聪明，然后判断最终谁会取得胜利。

### 简单博弈

一眼能看出胜败关系的博弈或者存在必胜或必败的博弈。

#### 简单博弈

##### 题目描述:

有  $2*n$  个硬币放在桌子上围成一个圆。现在甲和乙依次轮流拿硬币，每人每次最多拿  $k$  枚硬币，并且这  $k$  枚硬币必须相邻才能拿，每次最少要拿走 1 枚硬币，没有硬币拿的一方输掉比赛。

假设两个人都足够聪明，甲先拿，问甲能取得最后的胜利吗？

##### 输入描述:

多组数据输入。

输入两个正整数  $n$  和  $k$ 。 ( $k \leq n$ ,  $n \leq 10^9$ )

##### 输出描述:

如果甲一定能赢则输出“WIN”，如果甲一定会输则输出“LOSE”，如果不能确定胜负则输出“?”。

##### 输入样例#:

1 1

##### 输出样例#:

LOSE

题目来源:

DreamJudge 1632

**题目解析:** 很多同学刚看的时候觉得很难推出胜负关系, 那是没有找到题目的重点, 显然这个题的重点是圆, 圆具有从任何方向都对称的性质。使用对称原理, 不论甲拿几个, 乙都可以在对称的位置拿一样的数量, 这样不管甲怎么拿, 都是乙必胜。

参考代码

```
1. #include<stdio.h>
2.
3. int main() {
4.     int n, k;
5.     while (scanf("%d%d", &n, &k) != EOF) {
6.         printf("LOSE\n");
7.     }
8.     return 0;
9. }
```

noobdream.com

简单变种问题

两个人轮流往一张圆桌上放硬币（硬币须全部在桌面上），当一方没有位置可放的时候，另一方获胜。问是否有一种策略可以判断是先手获胜还是后手获胜？如果有，策略是什么？

**提示:** 也是利用圆的对称性

## 巴什博弈

两个顶尖聪明的人在玩游戏，有  $n$  个石子，每人可以随便拿  $1-m$  个石子，不能拿的人为败者，问谁会胜利？

巴什博弈是博弈论问题中基础的问题，它是最简单的一种情形对应一种状态的博弈。

我们从最简单的情景开始分析

当石子有  $1-m$  个时，毫无疑问，先手必胜

当石子有  $m+1$  个时，先手无论拿几个，后手都可以拿干净，先手必败

当石子有  $m+2-2m$  时，先手可以拿走几个，剩下  $m+1$  个，先手必胜

我们不难发现，面临  $m+1$  个石子的人一定失败。

这样的话两个人的最优策略一定是通过拿走石子，使得对方拿石子时还有  $m+1$  个

我们考虑往一般情况推广

设当前的石子数为  $n=k*(m+1)+r$

先手会首先拿走  $r$  个，接下来假设后手拿走  $x$  个，先手会拿走  $m+1-x$  个，这样博弈下去后手最终一定失败

设当前的石子数为  $n=k*(m+1)$

假设先手拿  $x$  个，后手一定会拿  $m+1-x$  个，这样下去先手一定失败

### Brave Game

#### 题目描述：

十年前读大学的时候，中国每年都要从国外引进一些电影大片，其中有一部电影就叫《勇敢者的游戏》（英文名称：Zathura），一直到现在，我依然对于电影中的部分电脑特技印象深刻。今天，大家选择上机考试，就是一种勇敢（brave）的选择；这个短学期，我们讲的是博弈（game）专题；所以，大家现在玩的也是“勇敢者的游戏”，这也是我命名这个题目的原因。当然，除了“勇敢”，我还希望看到“诚信”，无论考试成绩如何，希望看到的都是一个真实

的结果，我也相信大家一定能做到的~

各位勇敢者要玩的第一个游戏是什么呢？很简单，它是这样定义的：

- 1、 本游戏是一个二人游戏；
- 2、 有一堆石子一共有  $n$  个；
- 3、 两人轮流进行；
- 4、 每走一步可以取走  $1 \cdots m$  个石子；
- 5、 最先取光石子的一方为胜；

如果游戏的双方使用的都是最优策略，请输出哪个人能赢。

**输入描述：**

如果先走的人能赢，请输出“first”，否则请输出“second”，每个实例的输出占一行。

**输出描述：**

共  $T$  行，如果对于这组数据存在先手必胜策略则输出 Yes，否则输出 No，每个单词一行。

**输入样例#：**

2

23 2

4 3

**输出样例#：**

first

second

**题目来源：**

DreamJudge 1636

**题目解析：**我们可以发现，当  $n=0$  的时候后手必胜(设其为 P 态),  $n=1 \sim m$  这几种状态由于先手可以一次全部取完导致先手必胜(设其为 N 态)。

接着当  $n=m+1$  时，因为先手无论取走几个都会使后手一次全部取走导致后手必胜；

接着当  $n=m+2 \sim 2*m$  时，因为先手可以一次就将石子取剩  $m+1$  个，导致无论后手取几个，最后都会使先手一次全部取完，导致先手必胜。

综上所述, 不难发现一个人想要获胜就要把对方逼到当对方选择时棋子剩  $m+1$  的倍数个, 由于先手的先发性, 只要  $n \neq (m+1)$  的倍数, 先手就一定可以一次就把石子取到剩  $(m+1)$  的倍数个, 不断逼使后手一直走必败路线。所以当  $n \bmod (m+1) = 0$  时, 后手必胜, 否则先手必胜。

## 参考代码

```
1. #include<stdio.h>
2.
3. int main () {
4.     int t;
5.     scanf("%d", &t);
6.     while(t--) {
7.         int n, k;
8.         scanf("%d%d", &n, &k);
9.         if(n % (k + 1) == 0){
10.            printf("second\n");
11.        }
12.        else{
13.            printf("first\n");
14.        }
15.    }
16.    return 0;
17. }
```



## 尼姆博弈

有两个顶尖聪明的人在玩游戏，游戏规则是这样的：

有  $n$  堆石子，两个人可以从任意一堆石子中拿任意多个石子(不能不拿)，没法拿的人失败。问谁会胜利？

nim 游戏是巴什博弈的升级版，它不再是简单的一个状态，因此分析起来也棘手许多。

如果说巴什博弈仅仅博弈论的一个引子的话，nim 游戏就差不多算是真正的入门了。

### 定理

对于 nim 游戏，前辈们发现了一条重要的规律！

当  $n$  堆石子的数量异或和等于 0 时，先手必胜，否则先手必败

### 简单证明（可略过）

设  $\oplus$  表示异或运算

nim 游戏的必败态我们是知道的，就是当前  $n$  堆石子的数量都为零

设  $a[i]$  表示第  $i$  堆石子的数量，那么当前局面就是

$$0 \oplus 0 \oplus 0 \oplus \dots \oplus 0 = 0$$

对于先手来说，如果当前局面是

$$a_1 \oplus a_2 \oplus a_3 \oplus \dots \oplus a_n = k$$

那么一定存在某个  $a_i$ ，它的二进制表示在最高位  $k$  上一定是 1

我们将  $a_i \oplus k$ ，这样就变成了

$$a_1 \oplus a_2 \oplus a_3 \oplus \dots \oplus a_n \oplus k = 0$$

此时先手必胜

对于先手来说，如果当前局面是

$$a_1 \oplus a_2 \oplus a_3 \oplus \dots \oplus a_n = 0$$

那么我们不可能将某一个  $a_i$  异或一个数字后使得

$$a_1 \oplus a_2 \oplus a_3 \oplus \dots \oplus a_n = 0$$

此时先手必败

## nim 游戏

### 题目描述：

甲，乙两个人玩 Nim 取石子游戏。

Nim 游戏的规则是这样的：地上有  $n$  堆石子（每堆石子数量小于  $10^4$ ），每人每次可从任意一堆石子里取出任意多枚石子扔掉，可以取完，不能不取。每次只能从一堆里取。最后没石子可取的人就输了。假如甲是先手，且告诉你这  $n$  堆石子的数量，他想知道是否存在先手必胜的策略。

#### 输入描述：

本题有多组测试数据。

第一行一个整数  $T$  ( $T \leq 10$ )，表示有  $T$  组数据

接下来每两行是一组数据，第一行一个整数  $n$ ，表示有  $n$  堆石子， $n \leq 10000$ 。

第二行有  $n$  个数，表示每一堆石子的数量。

#### 输出描述：

共  $T$  行，如果对于这组数据存在先手必胜策略则输出 Yes，否则输出 No，每个单词一行。

#### 输入样例#：

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

#### 输出样例#：

```
No
Yes
```

#### 题目来源：

DreamJudge 1635

**题目解析：**根据 NIM 游戏的定理当  $n$  堆石子的数量异或和等于 0 时，先手必胜，否则先手必败即可。

#### 参考代码

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

```
2.  
3. int main(){  
4.     int t, n;  
5.     scanf("%d", &t);  
6.     while(t--){  
7.         scanf("%d", &n);  
8.         int ans = 0;  
9.         for(int i = 1; i <= n; ++i){  
10.            int shu;  
11.            scanf("%d", &shu);  
12.            ans ^= shu;  
13.        }  
14.        if(!ans) printf("No\n");  
15.        else printf("Yes\n");  
16.    }  
17.    return 0;  
18. }
```



## 威佐夫博弈

noobdream.com

有两堆石子，两个顶尖聪明的人在玩游戏，每次每个人可以从任意一堆石子中取任意多的石子或者从两堆石子中取同样多的石子，不能取得人输，分析谁会获得胜利？

威佐夫博弈不同于 Nim 游戏与巴什博奕，它的特殊之处在于不能将两堆石子分开分析。证明略。

定理

假设两堆石子为 $(x, y)$ （其中  $x < y$ ）

那么先手必败，当且仅当

$$(y - x) * \frac{(\sqrt{5}+1)}{2} = x$$

其中的 $\frac{(\sqrt{5}+1)}{2}$ 实际就是 1.6181.618，黄金分割数！怎么样，博弈论是不是很神奇？

### 取石子游戏

#### 题目描述：

有两堆石子，数量任意，可以不同。游戏开始由两个人轮流取石子。游戏规定，每次有两种不同的取法，一是可以在任意的一堆中取走任意多的石子；二是可以在两堆中同时取走相同数量的石子。最后把石子全部取完者为胜者。现在给出初始的两堆石子的数目，你先取，假设双方都采取最好的策略，问最后你是胜者还是败者。

#### 输入描述：

多组测试数据。

一行输入两个数 a, b, 表示石子的初始情况。

a, b <= 1 000 000 000

#### 输出描述：

输出共一行。

第一行为一个数字 1、0 或-1，如果最后你是胜利者则为 1；若失败则为 0；若结果无法确定则为-1。

#### 输入样例#：

8 4

#### 输出样例#：

1

#### 题目来源：

DreamJudge 1637

**题目解析：**利用威佐夫博弈定理即可解决。

## 参考代码

```
1. #include<stdio.h>
2. #include<math.h>
3. #include<stdlib.h>
4.
5. main() {
6.     int a, b;
7.     while (scanf("%d%d", &a, &b) != EOF) {
8.         int temp = abs(a - b);
9.         int ans = temp*(1.0+sqrt(5.0))/2.0;
10.        if (b < a) a = b;
11.        if(ans == a) printf("0\n");
12.        else printf("1\n");
13.    }
14.    return 0;
15. }
```



noobdream.com

## 题目练习

DreamJudge 1633 Euclid's Game

DreamJudge 1634 A interesting game

## 2.5 二分答案技巧

二分查找是最基础的算法, 其效率较高且应用广泛, 但它要求表中元素按关键字单调有序排列, 同样二分答案:

**应用前提:**

二分答案要求满足条件的答案单调

否则你就不能确定下一次查找答案所在的区间

**基本思想:**

在答案可能的范围内  $[L, R]$  二分查找答案, 检查当前答案是否满足题目的条件要求, 根据判断结果更新查找区间。

不难发现, 二分查找就是在二分答案, 答案所在区间为有序线性表的第一个元素到最后一个, 条件即为要找的那个值。

### 解决四类问题

1. 求最大的最小值
2. 求最小的最大值
3. 求满足条件的最大(小)值
4. 求最靠近一个值的值

**提示:** 注意一些题目表面上一眼看上去就像二分, 但实际上仔细验证其答案并没有单调性, 所以正解往往是暴力枚举。

#### Aggressive cows

**题目描述:**

农夫 John 建造了一个新的畜栏, 其中有  $N$  个 ( $2 \leq N \leq 100,000$ ) 隔间。隔间沿直线位于

位置  $x_1, \dots, x_N$  ( $0 \leq x_i \leq 1,000,000,000$ )。

John 的  $C$  ( $2 \leq C \leq N$ ) 头牛不喜欢这种布局, 而且几头牛放在一个隔间里, 他们就要发生争斗。为了不让牛互相伤害。John 决定自己给牛分配隔间, 使任意两头牛之间的最小距离尽可能的大, 那么, 这个最大的最小距离是什么呢?

#### 输入描述:

第 1 行: 两个以空格分隔的整数:  $N$  和  $C$

第 2.. $N + 1$  行: 第  $i + 1$  行包含整数停转位置  $x_i$

#### 输出描述:

第 1 行: 一个整数: 最大最小距离

#### 输入样例#:

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

#### 输出样例#:

```
3
```

#### 题目来源:

DreamJudge 1638

题目解析: 暴力选  $c$  个点肯定超时, 考虑二分答案, 这个最大值的范围为  $[0, \text{点的最大值}-\text{最小值}]$ , 如果能找出  $c$  个使得他们相邻的点之间的最小距离为  $\text{mid}$ , 那么小于  $\text{mid}$  的答案肯定也满足要求, 若不能找到, 那么大于  $\text{mid}$  的答案就更不能找到

#### 参考代码

```
1. #include <bits/stdc++.h>
```

```

2. using namespace std;
3.
4. const int N = 1e6+10;
5. int a[N], n, m;
6. bool judge(int k){ //枚举间距 k, 看能否使任意两相邻牛
7.     int cnt = a[0], num = 1; //num 为 1 表示已经第一头牛放在 a[0] 牛栏中
8.     for(int i = 1; i < n; i++){ //枚举剩下的牛栏
9.         if(a[i] - cnt >= k){ //a[i] 这个牛栏和上一个牛栏间距大于等于 k, 表示可以再放进牛
10.             cnt = a[i];
11.             num++; //又放进了一头牛
12.         }
13.         if(num >= m) return true; //所有牛都放完了
14.     }
15.     return false;
16. }
17. void solve() {
18.     int l = 1, r = a[n-1] - a[0]; //最小距离为 1, 最大距离为牛栏编号最大的减去编号最小的
19.     while(l < r){ //二分答案
20.         int mid = (l+r+1) >> 1; //小技巧: 加 1 防止死循环
21.         if(judge(mid)) l = mid;
22.         else r = mid - 1;
23.     }
24.     printf("%d\n", l);
25. }
26. int main(){
27.     int i;
28.     while(~scanf("%d%d", &n, &m)){
29.         for(i = 0; i < n; i++)
30.             scanf("%d", &a[i]);
31.         sort(a, a+n); //对牛栏排序
32.         solve();
33.     }
34.     return 0;
35. }

```

什么时候才考虑用二分答案的技巧?

正向求出答案不好入手, 求解答案远远没有验证答案简单。



## 2.6 前缀和技巧

前缀和是一个十分实用的技巧，难度很低，机试中很常见，要求同学们必须掌握。

	定义式	递推式
一维	$b[i] = \sum_{j=0}^i a[j]$	$b[i] = b[i-1] + a[i]$
二维	$b[x][y] = \sum_{i=0}^x \sum_{j=0}^y a[i][j]$	$b[x][y] = b[x-1][y] + b[x][y-1] - b[x-1][y-1] + a[x][y]$

前缀和的应用十分的多，很多时候和差分思想结合起来能无往不利。

下面列举几个机试中常见的考察方法

1、给你  $n$  个整数，然后进行  $m$  次询问，其中  $n$  和  $m$  很大，每次询问让你求某段区间的和。

解析：由于  $n$  和  $m$  很大，所以不能在每次询问的时候暴力遍历这段区间的和。那么我们考虑用前缀和进行预处理， $sum[i]$  表示前  $i$  个数的前缀和，对于每次询问的区间  $[l, r]$  的结果为  $sum[r] - sum[l-1]$ 。

2、见下面例题

### TAT 的个数

**题目描述：**

小诺有一个只包含 T 和 A 的字符串，小诺想知道这个序列中有多少个 TAT？

比如：TATT 中包含了 2 个不同位置的 TAT，TAATT 中包含了 4 个不同位置的 TAT。

**输入描述：**

多组数据输入。

输入一行字符串，字符串长度小于 100000。

输出描述：

输出答案。

输入样例#：

TATT

输出样例#：

2

题目来源：

DreamJudge 1646

题目解析：我们只需要枚举字符串中的每个 A，如果我们知道每一个 A 它前面有多少个 T 记为 left，它后面有多少个 T 记为 right，那么经过这个 A 的 TAT 数量就是  $\text{left} * \text{right}$ 。然后我们用前缀和的思想预处理出每个位置的 left 和 right 即可直接枚举 A 求解。

参考代码

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. typedef long long ll;
5. ll left[100005]; //从左边开始到第 i 个有多少个 T
6. ll right[100005]; //从右边开始到第 i 个有多少个 T
7. char s[100005];
8. int main() {
9.     while (scanf("%s", s+1) != EOF) {
10.         int len = strlen(s+1);
11.         for (int i = 1; i <= len; i++) {
12.             if (s[i] == 'T') left[i] = left[i - 1] + 1;
13.             else left[i] = left[i - 1];
14.         }
15.         for (int i = len; i >= 1; i--) {
16.             if (s[i] == 'T') right[i] = right[i + 1] + 1;
17.             else right[i] = right[i + 1];
18.         }
19.         ll sum = 0;
```

```
20.     for (int i = 1; i <= len; i++) { //枚举每一个 A
21.         if (s[i] == 'A') sum += (left[i] * right[i]);
22.     }
23.     printf("%lld\n", sum);
24. }
25.     return 0;
26. }
```

N 诺

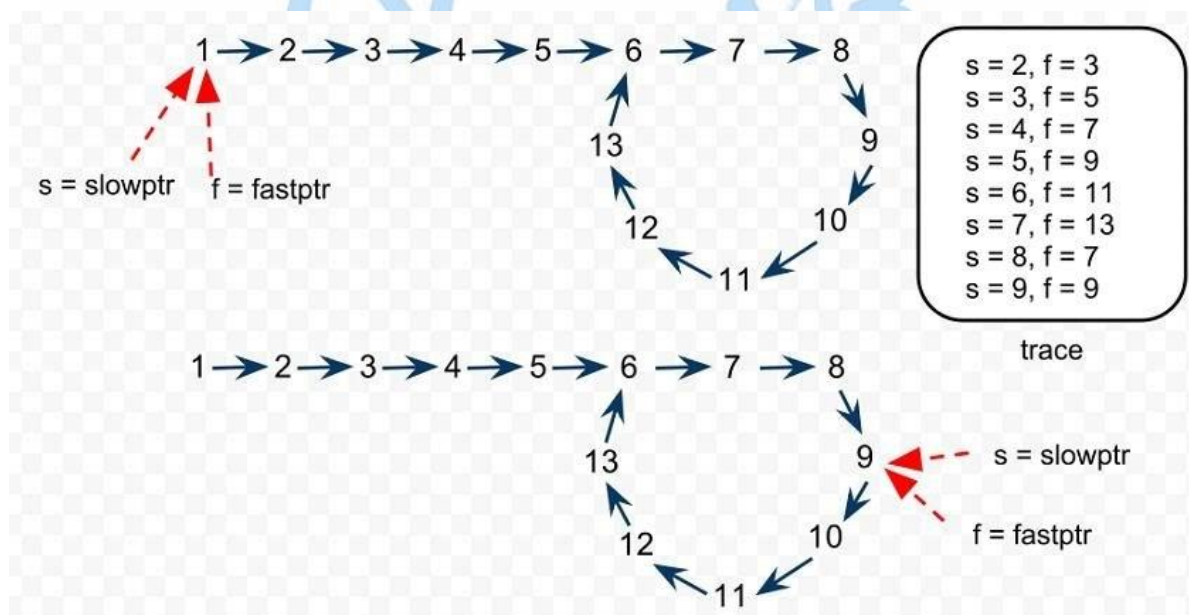
noobdream.com

## 2.7 双指针技巧

双指针是一种思想，一种技巧或一种方法，并不是什么特别具体的算法，在二分查找等算法中经常用到这个技巧。具体就是用两个变量动态存储两个或多个结点，来方便我们进行一些操作。通常用在线性的数据结构中，比如链表和数组，有时候也会用在图算法中。

在我们遇到像数组，链表这类数据结构的算法题目的时候，应该要想得到双指针的套路来解决问题。特别是链表类的题目，经常需要用到两个或多个指针配合来记忆链表上的节点，完成某些操作。链表这种数据结构也是树形结构和图的原型，所以有时候在关于图和树形结构的算法题目中也会用到双指针。

当你遇到此类数据结构，尝试使用双指针来解题的时候，可以从以下几个双指针类题目的套路入手进行思考。



### 快慢指针

类似于龟兔赛跑，两个链表上的指针从同一节点出发，其中一个指针前进速度是另一个指针的两倍。利用快慢指针可以用来解决某些算法问题，比如：

- 1、计算链表的中点：**快慢指针从头节点出发，每轮迭代中，快指针向前移动两个节点，慢指针向前移动一个节点，最终当快指针到达终点的时候，慢指针刚好在中间的节点。
  - 2、判断链表是否有环：**如果链表中存在环，则在链表上不断前进的指针会一直在环里绕圈子，且不能知道链表是否有环。使用快慢指针，当链表中存在环时，两个指针最终会在环中相遇。
  - 3、判断链表中环的起点：**当我们判断出链表中存在环，并且知道了两个指针相遇的节点，我们可以让其中任一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置。
  - 4、求链表中环的长度：**只要相遇后一个不动，另一个前进直到相遇算一下走了多少步就好了
- 求链表倒数第  $k$  个元素：先让其中一个指针向前走  $k$  步，接着两个指针以同样的速度一起向前进，直到前面的指针走到尽头了，则后面的指针即为倒数第  $k$  个元素。（严格来说应该叫先后指针而非快慢指针）

## 碰撞指针

一般都是排好序的数组或链表，否则无序的话这两个指针的位置也没有什么意义。特别注意两个指针的循环条件在循环体中的变化，小心右指针跑到左指针左边去了。常用来解决的问题有：

### 1、二分查找问题

- 2、 $n$  数之和问题：**比如两数之和问题，先对数组排序然后左右指针找到满足条件的两个数。如果是三数问题就转化为一个数和另外两个数的两数问题，以此类推。

## 第三章 满分之路中

高分篇的内容已经将机试可能涉及到的题型及考点都一一列举出来, 并给大家提供了练习题目, 帮助大家巩固基础加深理解。在大多数院校的机试中, 学会高分篇的内容已经足够考到 90 分以上的成绩, 题目不难的情况下甚至能拿到 100 分。在少数院校难度较大的院校也足够拿到 80 分左右的分数, 如果你基础比较薄弱, 我们建议你掌握高分篇的内容就可以了。本书满分之路的内容不是常考点, 只是有一定可能会考到, 我们建议基础好的同学继续学习增加拿满分的把握。

距离满分只有一步之遥, 你! 愿意放弃吗?

本章我们重点讲解一些较为深入的题型, 包括线段树单点更新、线段树区间更新、字符串匹配问题、kmp 的扩展和应用、二分图的匹配问题和路径进阶问题等内容。希望能帮助读者更好的掌握计算机考研机试中所涉及到的各类较难的问题。

**提示: 满分篇的内容难度大选择性的学就行, 不要求全部掌握, 把能学会的学会就可以了。**

本书配套视频精讲: <https://www.bilibili.com/video/av91373687>

## 3.1 线段树单点更新

线段树是一个非常有用的东西，基本上在考研或保研机试中，数据结构学到线段树这里就足以无所畏惧了。在部分机试难度较高的学校，有时候会出一道线段树的题目来作为区分学生水平的压轴题。这类题非常有意思，数据小的时候，一个刚会写代码的小白都会做，数据大的时候就能看出同学们各自的数据结构和代码功底了。

### 定义

线段树，是一种二叉搜索树。它将一段区间划分为若干单位区间，每一个节点都储存着一个区间。它功能强大，支持区间求和，区间最大值，区间修改，单点修改等操作。

线段树的思想是分治思想很相像。

线段树的每一个节点都储存着一段区间 $[L...R]$ 的信息，其中叶子节点 $L=R$ 。它的大致思想是：将一段大区间平均地划分成2个小区间，每一个小区间都再平均分成2个更小区间.....以此类推，直到每一个区间的 $L$ 等于 $R$ （这样这个区间仅包含一个节点的信息，无法被划分）。通过对这些区间进行修改、查询，来实现对大区间的修改、查询。

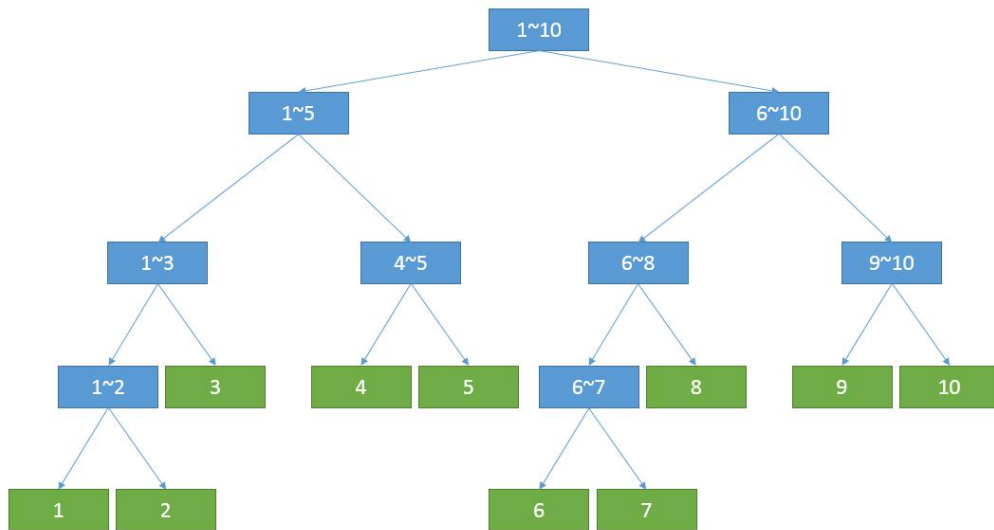
这样一来，每一次修改、查询的时间复杂度都只为 $O(\log n)$

但是，可以用线段树维护的问题必须满足区间加法，否则是不可能将大问题划分成子问题来解决的。

### 原理及实现

为了防止同学们看文字图片看的云里雾里，请直接看本书配套的视频，视频中详细讲解了线段树的原理及实现，深入浅出，相信能让同学们深刻理解线段树的单点更新。





### 线段树-单点更新 1

#### 题目描述:

小诺在家无聊, 和朋友小 A 玩一个游戏。

小 A 在纸上写了  $N$  个数字排成一排, 从左到右依次是  $a_1, a_2, a_3, \dots, a_n$ 。

小 A 会进行  $M$  次操作, 你能帮助小诺应对小 A 的挑战吗?

小 A 每次操作会是下面四种操作之一:

Add  $x\ y$ : 将  $a_x$  增加  $y$

Sub  $x\ y$ : 将  $a_x$  减少  $y$

Update  $x\ y$ : 将  $a_x$  变成  $y$

Query  $L\ R$ : 小 A 向小诺进行提问, 区间  $[L, R]$  之间的数的累加和是多少

#### 输入描述:

多组数据输入。

第一行输入两个数  $N$  和  $M$ , 代表有  $N$  个数, 小 A 进行  $M$  次操作。

第二行输入  $N$  个整数用空格隔开。

接下来  $M$  行, 代表小 A 的操作。

其中数据范围不会超出  $\text{int}$  且都是整数。

$1 \leq N, M \leq 10^5$



### 输出描述:

按题目要求输出结果。

### 输入样例#:

```
5 6
1 2 3 4 5
Query 1 4
Add 2 2
Query 1 4
Update 5 1
Query 1 4
Query 1 5
```

### 输出样例#:

```
10
12
12
13
```

### 题目来源:

DreamJudge 1648

**题目解析:** 由于数据量很大, 如果我们直接模拟的话, 每次查询操作最坏情况是  $O(n)$ , 在 10W 次查询的情况下必定是超时的。所以我们用线段树来解决这个问题, 具体看下面的参考代码。

### 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. #define lson x << 1
5. #define rson (x << 1) + 1
6.
7. const int maxn = 200000 + 5;
```

```
8. int tree[maxn << 2]; //定义线段树结点
9. int arr[maxn]; //输入的数据
10. int ans; //答案
11. int n, m;
12. //向上合并
13. void Push_Up(int x) {
14.     tree[x] = tree[lson] + tree[rson];
15. }
16. //创建一颗线段树
17. void Create(int x, int l, int r) {
18.     if (l == r) {
19.         tree[x] = arr[l];
20.         return;
21.     }
22.     int mid = (l + r) / 2;
23.     Create(lson, l, mid);
24.     Create(rson, mid + 1, r);
25.     Push_Up(x);
26. }
27. //将 pos 点的值更新为 val
28. void Update(int x, int l, int r, int pos, int val) {
29.     if (l >= r) {
30.         tree[x] = val;
31.         return;
32.     }
33.     int mid = (l + r) / 2;
34.     if (pos <= mid) Update(lson, l, mid, pos, val);
35.     if (pos > mid) Update(rson, mid + 1, r, pos, val);
36.     Push_Up(x);
37. }
38. //将 pos 点的值增加 val
39. void Add(int x, int l, int r, int pos, int val) {
40.     if (l >= r) {
41.         tree[x] += val;
42.         return;
43.     }
44.     int mid = (l + r) / 2;
45.     if (pos <= mid) Add(lson, l, mid, pos, val);
46.     if (pos > mid) Add(rson, mid + 1, r, pos, val);
47.     Push_Up(x);
48. }
49. //查询区间[L, R]之间所有值的累加和
50. void Query(int x, int l, int r, int L, int R) {
```

```
51.     if (L <= l && R >= r) {
52.         ans += tree[x];
53.         return;
54.     }
55.     int mid = (l + r) / 2;
56.     if (L <= mid) Query(lson, l, mid, L, R);
57.     if (R > mid) Query(rson, mid + 1, r, L, R);
58. }
59.
60.
61. int main() {
62.     while (scanf("%d%d", &n, &m) != EOF) {
63.         for (int i = 1; i <= n; i++) scanf("%d", &arr[i]);
64.         Create(1, 1, n);
65.         while (m--) {
66.             ans = 0;
67.             char ch[10];
68.             int a, b;
69.             scanf("%s", ch);
70.             scanf("%d%d", &a, &b);
71.             if (ch[0] == 'U') { //只判断第一个字符速度更快
72.                 Update(1, 1, n, a, b);
73.             }
74.             else if (ch[0] == 'A') {
75.                 Add(1, 1, n, a, b);
76.             }
77.             else if (ch[0] == 'S') {
78.                 Add(1, 1, n, a, -b); //减去一个数就是加上相反数
79.             }
80.             else {
81.                 Query(1, 1, n, a, b);
82.                 printf("%d\n", ans);
83.             }
84.         }
85.     }
86.     return 0;
87. }
```

小结：题目除了可能让同学们求和，也有可能让同学们求最大值、最小值等可以通过区间合并的问题。还要注意一些细节问题，比如题目给的区间范围不一定是从小到大的，例如[3, 7]是一般正常的输入，也可能是[7, 3]这样的区间，含义是一样的，所以同学们做题的时候一定要小心，如果存在这种情况，就需要将两个数先比较大小。

### 题目练习

DreamJudge 1649 线段树-单点更新 2



## 3.2 线段树区间更新

上一节给同学们讲了线段树的单点更新，同学们可能会有疑问，如果更新的时候不是一个点，而是一段区间我们应该怎么做呢？

是遍历这段区间的每一个点，然后一个点一个点的更新吗？

答案当然是否定的，这样的更新速度实在太慢，还不如暴力算法来的快。

那么这一节就给同学们讲解遇到区间更新的线段树解决技巧。

区间更新的核心就是打上一个延迟标记也叫懒惰标记，一般我们用 lazy 数组来实现。

### 原理及实现

为了防止同学们看文字图片看的云里雾里，请直接看本书配套的视频，视频中详细讲解了线段树的原理及实现，深入浅出，相信能让同学们深刻理解线段树的区间更新。

#### 线段树-区间更新 1

##### 题目描述：

小诺在家无聊，和朋友小 A 玩一个游戏。

小 A 在纸上写了  $N$  个数字排成一排，从左到右依次是  $a_1, a_2, a_3, \dots, a_n$ 。

小 A 会进行  $M$  次操作，你能帮助小诺应对小 A 的挑战吗？

小 A 每次操作会是下面三种操作之一：

Add  $L R x$ ：将区间  $[L, R]$  内的每一个数都增加  $x$

Sub  $L R x$ ：将区间  $[L, R]$  内的每一个数都减少  $x$

Query  $L R$ ：小 A 向小诺进行提问，区间  $[L, R]$  之间的数的累加和是多少

注意：数据有可能超出  $int$ ，但不会超出  $long\ long$  的范围。

##### 输入描述：

多组数据输入。

第一行输入两个数  $N$  和  $M$ ，代表有  $N$  个数，小 A 进行  $M$  次操作。

第二行输入  $N$  个整数用空格隔开。

接下来  $M$  行, 代表小 A 的操作。

其中数据范围不会超出 `long long` 且都是整数。

$1 \leq N, M \leq 10^5$

#### 输出描述:

按题目要求输出结果。

#### 输入样例#:

```
5 6
1 2 3 4 5
Query 1 4
Add 2 3 1
Query 1 4
Sub 5 5 1
Query 1 4
Query 1 5
```

#### 输出样例#:

```
10
12
12
16
```

#### 题目来源:

DreamJudge 1650

**题目解析:** 这个题和上一节例题的区别在于上一节的题目更新的时候都是更新的单个结点, 而这个题更新的是一段区间, 如果我们暴力去更新区间每一个点必然是会超时的。那么我们应该如何直接更新整个区间呢? 这时候我们就需要用延迟(懒惰)标记这个技巧, 具体请同学们看下面的参考代码。

## 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. #define lson x << 1
5. #define rson (x << 1) + 1
6. typedef long long ll;
7. const int maxn = 200000 + 5;
8. ll tree[maxn << 2]; //定义线段树结点
9. ll lazy[maxn << 2]; //懒惰标记
10. int arr[maxn]; //输入的数据
11. ll ans; //答案
12. int n, m;
13. //向上合并
14. void Push_Up(int x) {
15.     tree[x] = tree[lson] + tree[rson];
16. }
17. //向下传递 lazy 标记
18. void Push_Down(int x, int l, int r) {
19.     int mid = (l + r) / 2;
20.     if(lazy[x]){ //若有标记, 则将标记向下移动一层
21.         lazy[lson] += lazy[x];
22.         lazy[rson] += lazy[x];
23.         tree[lson] += (mid - l + 1) * lazy[x];
24.         tree[rson] += (r - mid) * lazy[x];
25.         lazy[x] = 0; //取消本层标记
26.     }
27. }
28. //创建一颗线段树
29. void Create(int x, int l, int r) {
30.     lazy[x] = 0;
31.     if (l == r) {
32.         tree[x] = arr[l];
33.         return;
34.     }
35.     int mid = (l + r) / 2;
36.     Create(lson, l, mid);
37.     Create(rson, mid + 1, r);
38.     Push_Up(x);
39. }
40. //将 pos 点的值增加 val
```

```
41. void Add(int x, int l, int r, int L, int R, int val) {
42.     if (L <= l && R >= r) { //找到要加的区间
43.         tree[x] += (r - l + 1) * val;
44.         lazy[x] += val;
45.         return;
46.     }
47.     Push_Down(x, l, r); //向下更新
48.     int mid = (l + r) / 2;
49.     if (L <= mid) Add(lson, l, mid, L, R, val);
50.     if (R > mid) Add(rson, mid + 1, r, L, R, val);
51.     Push_Up(x);
52. }
53. //查询区间[L, R]之间所有值的累加和
54. void Query(int x, int l, int r, int L, int R) {
55.     if (L <= l && R >= r) { //找到查询的区间
56.         ans += tree[x];
57.         return;
58.     }
59.     Push_Down(x, l, r); //向下更新
60.     int mid = (l + r) / 2;
61.     if (L <= mid) Query(lson, l, mid, L, R);
62.     if (R > mid) Query(rson, mid + 1, r, L, R);
63. }
64.
65. int main() {
66.     while (scanf("%d%d", &n, &m) != EOF) {
67.         for (int i = 1; i <= n; i++) scanf("%d", &arr[i]);
68.         Create(1, 1, n);
69.         while (m--) {
70.             ans = 0;
71.             char ch[10];
72.             int a, b, x;
73.             scanf("%s", ch);
74.             if (ch[0] == 'A') {
75.                 scanf("%d%d%d", &a, &b, &x);
76.                 Add(1, 1, n, a, b, x);
77.             }
78.             else if (ch[0] == 'S') {
79.                 scanf("%d%d%d", &a, &b, &x);
80.                 Add(1, 1, n, a, b, -x); //减去一个数就是加上相反数
81.             }
82.             else {
83.                 scanf("%d%d", &a, &b);
```



```
84.         Query(1, 1, n, a, b);
85.         printf("%lld\n", ans);
86.     }
87. }
88. }
89.     return 0;
90. }
```

**总结：**线段树有非常多的应用，题目难度变化也很大，简单的题目可以让你一眼就知道做法，难的题目往往结合其他算法和技巧足以让绝大多数初学者感到绝望。在考研机试中，学会这两节的内容，能让你深刻理解线段的含义，在遇到其他变形应用时能快速的反应过来。

由于担心同学们初学线段树对延迟标记的理解深度不够会出问题，所以把课后的练习题目代码也贴出来给同学们参考一下。

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. #define lson x << 1
5. #define rson (x << 1) + 1
6. #define INF 0x3f3f3f3f
7. typedef long long ll;
8. const int maxn = 200000 + 5;
9. ll tree[maxn << 2]; //定义线段树结点
10. ll lazy[maxn << 2]; //懒惰标记
11. int arr[maxn]; //输入的数据
12. ll ans; //答案
13. int n, m;
14. //向上合并
15. void Push_Up(int x) {
16.     tree[x] = max(tree[lson], tree[rson]);
17. }
18. //向下传递 lazy 标记
19. void Push_Down(int x, int l, int r) {
20.     int mid = (l + r) / 2;
21.     if(lazy[x]){ //若有标记,则将标记向下移动一层
```

```
22.     lazy[lson] += lazy[x];
23.     lazy[rson] += lazy[x];
24.     tree[lson] = max(tree[lson], tree[lson] + lazy[x]);
25.     tree[rson] = max(tree[rson], tree[rson] + lazy[x]);
26.     lazy[x] = 0; //取消本层标记
27. }
28. }
29. //创建一颗线段树
30. void Create(int x, int l, int r) {
31.     lazy[x] = 0;
32.     if (l == r) {
33.         tree[x] = arr[l];
34.         return;
35.     }
36.     int mid = (l + r) / 2;
37.     Create(lson, l, mid);
38.     Create(rson, mid + 1, r);
39.     Push_Up(x);
40. }
41. //将 pos 点的值增加 val
42. void Add(int x, int l, int r, int L, int R, int val) {
43.     if (L <= l && R >= r) { //找到要加的区间
44.         tree[x] = tree[x] + val;
45.         lazy[x] += val;
46.         return;
47.     }
48.     Push_Down(x, l, r); //向下更新
49.     int mid = (l + r) / 2;
50.     if (L <= mid) Add(lson, l, mid, L, R, val);
51.     if (R > mid) Add(rson, mid + 1, r, L, R, val);
52.     Push_Up(x);
53. }
54. //查询区间[L, R]之间所有值的累加和
55. void Query(int x, int l, int r, int L, int R) {
56.     if (L <= l && R >= r) { //找到查询的区间
57.         ans = max(ans, tree[x]);
58.         return;
59.     }
60.     Push_Down(x, l, r); //向下更新
61.     int mid = (l + r) / 2;
62.     if (L <= mid) Query(lson, l, mid, L, R);
63.     if (R > mid) Query(rson, mid + 1, r, L, R);
64. }
```

```
65.
66. int main() {
67.     while (scanf("%d%d", &n, &m) != EOF) {
68.         for (int i = 1; i <= n; i++) scanf("%d", &arr[i]);
69.         Create(1, 1, n);
70.         while (m--) {
71.             ans = -INF; // 由于有减少所以初始化为负无穷大
72.             char ch[10];
73.             int a, b, x;
74.             scanf("%s", ch);
75.             if (ch[0] == 'A') {
76.                 scanf("%d%d%d", &a, &b, &x);
77.                 Add(1, 1, n, a, b, x);
78.             }
79.             else if (ch[0] == 'S') {
80.                 scanf("%d%d%d", &a, &b, &x);
81.                 Add(1, 1, n, a, b, -x); // 减去一个数就是加上相反数
82.             }
83.             else {
84.                 scanf("%d%d", &a, &b);
85.                 Query(1, 1, n, a, b);
86.                 printf("%lld\n", ans);
87.             }
88.         }
89.     }
90.     return 0;
91. }
```

## 题目练习

DreamJudge 1651 线段树-区间更新 2

### 3.3 字符串匹配问题

字符串匹配是机试中非常常见的一个问题，我们可以把字符串匹配问题分为两大类。

#### 1、暴力匹配

#### 2、算法优化的匹配

我们看一个例子

例：输入两个字符串 s1 和 s2，询问 s2 是否在 s1 中出现过或者说 s2 是否是 s1 的子串？

如：s1 = “abcde” s2 = “cd” 那么 s2 在 s1 中出现过

如：s1 = “abcde” s2 = “ce” 那么 s2 没有在 s1 中出现

解析：当我们遇到这类问题的时候我们可以很轻松的写出一个暴力匹配的代码，即拿 s2 和 s1 中每一个长度和 s2 相等的子串去挨个比较，如果比较结果相等，即出现过，否则就没有出现过。

暴力代码如下

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. #define MAXN 1000010
5. char txt[MAXN]; //文本串
6. char str[MAXN]; //模式串
7.
8. int main() {
9.     scanf("%s", txt + 1);
10.    scanf("%s", str + 1);
11.    int str_len = strlen(str+1);
12.    int txt_len = strlen(txt+1);
13.    for (int i = 1; i <= txt_len; i++) {
14.        int flag = 0;
15.        for (int j = 1; j <= str_len; j++) {
16.            if (txt[i + j - 1] != str[j]) {
17.                flag = 1; //发现不匹配
18.                break; //小优化
19.            }
20.        }
21.        if (flag == 0) { //如果找到了就结束
```

```
22.         printf("YES\n");
23.         return 0; //小优化
24.     }
25. }
26. printf("NO\n");
27. return 0;
28. }
```

思考：很显现我们可以算出这个暴力匹配的时间复杂度是两个字符串长度的乘积，即  $O(\text{len}(s1) * \text{len}(s2))$ 。所以我们知道在大多数题目字符串长度不超过 3000 的情况下我们就可以这么做，但是如果两个字符串长度都很长，到达 10W 甚至 100W 呢？

## KMP 算法

定义：上面的 S1 我们一般叫做文本串，上面的 S2 我们一般叫做模式串，所以我们通常将这类匹配方法称为模式串匹配。

暴力匹配大概就是枚举每一个文本串元素，然后从这一位开始不断向后比较，每次比较失败之后都要从头开始重新比对。

而 KMP 的精髓在于，对于每次失配之后，我都不会从头重新开始枚举，而是根据我已经得知的数据，从某个特定的位置开始匹配；而对于模式串（即 S2）的每一位，都有唯一的“特定变化位置”，这个在失配之后的特定变化位置可以帮助我们利用已有的数据不用从头匹配，从而节约时间。

比如我们考虑一些样例：

模式串：abcbab

文本串：abcacababcbab

首先，前四位按位匹配成功，遇到第五位不同，而这时，我们选择将 abcbab 向右移三位，或者直接理解为移动到模式串中与失配字符相同的那一位。可以简单地理解为，我们将两个已经遍历过的模式串字符重合，导致我们可以不用一位一位地移动，而是根据相同的字符来实现快速移动。

模式串:     `abcbab`

文本串: `abcacababcbab`

但有时不光只会有单个字符重复:

模式串: `abcabc`

文本串: `abcabdababcbabc`

当我们发现在第六位失配时, 我们可以将模式串的第一二位移到第四五位, 因为它们相同。

模式串:     `abcabc`

文本串: `abcabdababcbabc`

那么现在已经很明了了, **KMP** 的重头戏就在于用失配数组来确定当某一位失配时, 我们可以将前一位跳跃到之前匹配过的某一位。而此处有几个先决条件需要理解:

1、我们的失配数组应当建立在模式串意义下, 而不是文本串意义下。因为显然模式串要更加灵活, 在失配后换位时, 更灵活简便地处理。

2、如何确定位置呢?

首先我们要明白, 基于先决条件 1 而言, 我们在预处理时应当考虑当模式串的第  $i$  位失配时, 应当跳转到哪里。因为在文本串中, 之前匹配过的所有字符已经没有用了——都是匹配完成或者已经失配的, 所以我们的 `kmp` 数组 (即用于确定失配后变化位置的数组, 下同) 应当记录的是:

在模式串 `str1` 中, 对于每一位 `str1(i)`, 它的 `kmp` 数组应当是记录一个位置  $j, j \leq i$  并且满足 `str1(i)=str1(j)` 并且在  $j \neq 1$  时理应满足 `str1(1)` 至 `str1(j-1)` 分别与 `str(i-j+1)~str1(i-1)` 按位相等  
上述即为移位法则

3、从前缀后缀来解释 **KMP** :

首先解释前后缀(因为太简单就不解释了):

给定串: ABCABA

前缀: A,AB,ABC,ABCA,ABCAB,ABCABA

后缀: A,BA,ABA,CABA,BCABA,ABCABA

其实刚才的移位法则就是对于模式串的前缀而言,用 `kmp` 数组记录到它为止的模式串前缀的真前缀和真后缀最大相同的位置(注意,这个地方没有写错,是真的有嵌套)。然而这个地方我们要考虑“模式串前缀的前缀和后缀最大相同的位置”原因在于,我们需要用到 `kmp` 数组换位时,当且仅当未完全匹配。所以我们的操作只是针对模式串的前缀——毕竟是失配函数,失配之后只有可能是某个部分前缀需要“快速移动”。所以这就可以解释 `KMP` 中前后缀应用的一个特点:

`KMP` 中前后缀不包括模式串本身,即只考虑真前缀和真后缀,因为模式串本身需要整体考虑,当且仅当匹配完整个串之后;而匹配完整个串不就完成匹配了吗。

## 代码实现

1、`_next[i]` 用于记录当匹配到模式串的第 `i` 位之后失配,该跳转到模式串的哪个位置,那么对于模式串的第一位和第二位而言,只能回跳到 1,因为是 `KMP` 是要将真前缀跳跃到与它相同的真后缀上去(通常也可以反着理解),所以当 `i=0` 或者 `i=1` 时,相同的真前缀只会是 `str1(0)` 这一个字符,所以 `_next[0]=_next[1]=1`。

2、那么我们该如何处理 `_next` 数组呢?我们可以考虑用模式串自己匹配自己

```
1. void Get_Next(int str_len) {
2.     int j = 0;
3.     for (int i = 2; i <= str_len; i++) {
4.         //此处判断 j 是否为 0 的原因在于,如果回跳到第一个字符就不用再回跳了
5.         while(j && str[i] != str[j+1])
6.             j = _next[j]; //通过自己匹配自己来得出每一个点的 next 值
7.         if(str[j+1] == str[i]) j++;
8.         _next[i] = j; //i+1 失配后应该如何跳
9.     }
10. }
```

那么这个“自己匹配自己”该如何理解呢？我们可以这么想：首先，在单次循环只有一个 if 来判断的原因在于每次至多向后多求一位的 next；

并且 j 是拥有可继承性的，由于 j 是用于比对前缀后缀的，那么对于一组前后缀而言，第 i-1 和第 j-1 位之前均相同或者有不同，决定着 i 和 j 匹配的结果是从 0 开始还是基于上一个 j 继续++

### 3、对于如何和文本串比对，很简单：

```
1. void Kmp(int str_len, int txt_len) {
2.     Get_Next(str_len);
3.     int j = 0; //j 可以看做表示当前已经匹配完的模式串的最后一位的位置
4.     //如果楼上看不懂，你也可以理解为 j 表示模式串匹配到第几位了
5.     for(int i = 1; i <= txt_len; i++) {
6.         while(j > 0 && str[j+1] != txt[i])
7.             j = _next[j];
8.         //如果失配，那么就不断向回跳，直到可以继续匹配
9.         if (str[j+1] == txt[i]) j++;
10.        if (j == str_len) { //如果匹配成功，那么对应的模式串位置++
11.            printf("%d\n", i-str_len+1);
12.            j = _next[j]; //继续匹配
13.        }
14.    }
15. }
```

## KMP 字符串匹配

### 题目描述：

如题，给出两个字符串 s1 和 s2，其中 s2 为 s1 的子串，求出 s2 在 s1 中所有出现的位置。为了减少骗分的情况，接下来还要输出子串的前缀数组 next。

### 输入描述：

第一行为一个字符串，即为 s1

第二行为一个字符串，即为 s2

$N \leq 1000000$ ,  $M \leq 1000000$ 。

### 输出描述：



若干行, 每行包含一个整数, 表示  $s_2$  在  $s_1$  中出现的位置

接下来 1 行, 包括  $|s_2|$  个整数, 表示前缀数组  $next[i]$  的值。

输入样例#:

ABABABC

ABA

输出样例#:

1

3

0 0 1

题目来源:

DreamJudge 1650

题目解析: 模板题, 按照 KMP 算法模板求解即可。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. #define MAXN 1000010
5. int _next[MAXN]; //注意不要用 next
6. char txt[MAXN]; //文本串
7. char str[MAXN]; //模式串
8.
9. void Get_Next(int str_len) {
10.     int j = 0;
11.     for (int i = 2; i <= str_len; i++) {
12.         //此处判断 j 是否为 0 的原因在于, 如果回跳到第一个字符就不用再回跳了
13.         while(j && str[i] != str[j+1])
14.             j = _next[j]; //通过自己匹配自己来得出每一个点的 next 值
15.         if(str[j+1] == str[i]) j++;
16.         _next[i] = j; //i+1 失配后应该如何跳
17.     }
```

```
18. }
19.
20. void Kmp(int str_len, int txt_len) {
21.     Get_Next(str_len);
22.     int j = 0; //j 可以看做表示当前已经匹配完的模式串的最后一位的位置
23.     //如果楼上看不懂, 你也可以理解为 j 表示模式串匹配到第几位了
24.     for(int i = 1; i <= txt_len; i++) {
25.         while(j > 0 && str[j+1] != txt[i])
26.             j = _next[j];
27.         //如果失配, 那么就不断向回跳, 直到可以继续匹配
28.         if (str[j+1] == txt[i]) j++;
29.         if (j == str_len) { //如果匹配成功, 那么对应的模式串位置++
30.             printf("%d\n", i-str_len+1);
31.             j = _next[j]; //继续匹配
32.         }
33.     }
34. }
35.
36. int main() {
37.     scanf("%s", txt + 1);
38.     scanf("%s", str + 1);
39.     int str_len = strlen(str+1);
40.     int txt_len = strlen(txt+1);
41.     Kmp(str_len, txt_len);
42.     for (int i = 1; i <= str_len; i++)
43.         printf("%d ", _next[i]);
44.     printf("\n");
45.     return 0;
46. }
```

## 题目练习

DreamJudge 1652 简单模式匹配

## 3.4 二分图的匹配问题

### 匹配

设  $G$  为二分图, 若在  $G$  的子图  $M$  中, 任意两条边都没有公共节点, 那么称  $M$  为二分图  $G$  的一个匹配, 且  $M$  的边数为匹配数。

### 最大匹配

寻找二分图边数最大的匹配称为最大匹配问题。

对此, 有解决此问题的 匈牙利算法, 时间复杂度为  $O(N*M)$ 。

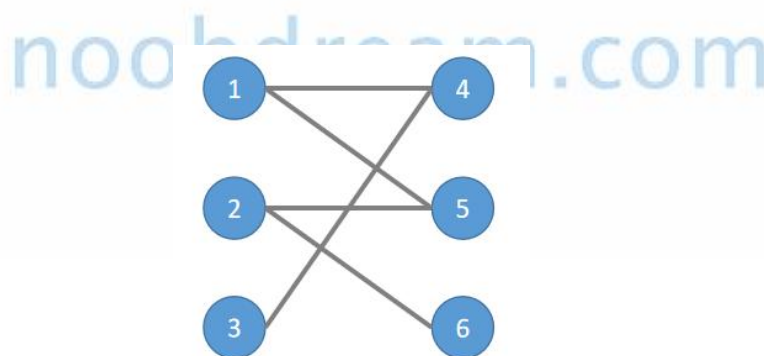
### 算法步骤大致如下:

首先从任意一个未配对的点  $u$  开始, 选择他的任意一条边 ( $u-v$ ), 如此时  $v$  还未配对, 则配对成功, 配对数加一, 若  $v$  已经配对, 则尝试寻找  $v$  的配对的另一个配对 (该步骤可能会被递归的被执行多次), 若该尝试成功, 则配对成功, 配对数加一。

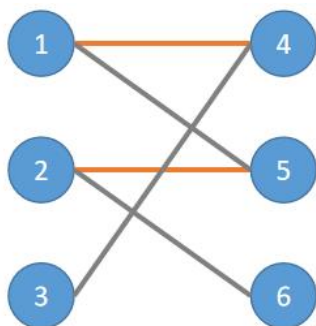
若果上一步配对不成功, 那么选择重新选择一条未被选择过的边, 重复上一步。

对剩下每一个没有被配对的点执行步骤 1, 直到所有的点都尝试完毕。

用下面的二分图为例:

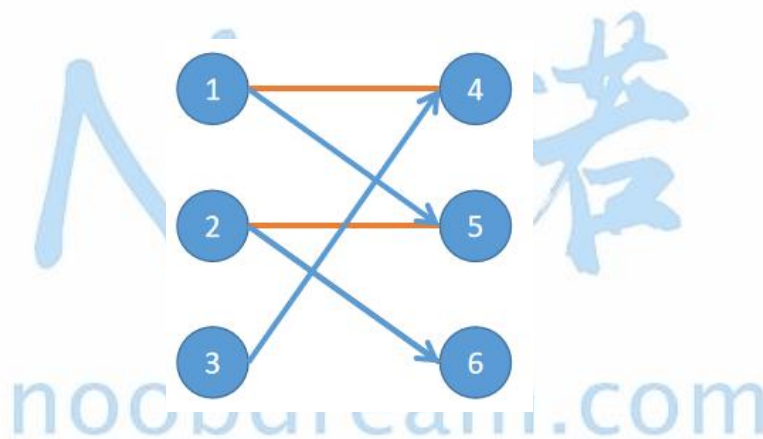


先对节点 1 和 2 尝试匹配, 假设他们分别找到了 4 和 5。

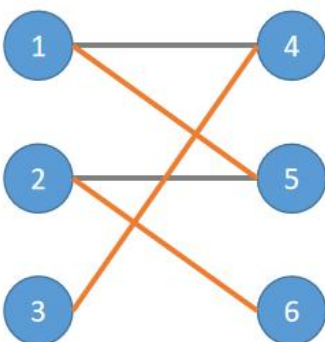


接下来对节点 3 尝试匹配，选择边 (3-4)，但发现 4 已经有匹配了，我们尝试寻找 4 的匹配的其他匹配，即 1 的其他匹配。这个匹配显然只能从未被选择的边里找（灰色的），我们可以遍历 1 的所有边，寻找未被选择的，很容易找到边 (1-5)。

我们发现 5 已经被匹配了，所以我们尝试寻找 5 的匹配的其他匹配，即 2 的其他匹配。类似的，可以找到 6。



于是我们得到了新的匹配方案，且该方案比之前的匹配数多一。



可以发现，当尝试对节点 3 进行匹配时，走过了一条路径 (3-4-1-5-2-6)，最后找到了新的匹配方案，我们把这样的道路叫做 **增广路**，其本质是一条起点和终点都是未匹配节点的路径。匈牙利算法执行的过程也可以看作是不断寻找增广路的过程，当在当前匹配方案下再也找不到增广路，那么当前匹配方案便是最大匹配了。

## 二分图匹配

### 题目描述:

给定一个二分图, 结点数分别为  $n, m$ , 边数为  $e$ , 求二分图最大匹配数

### 输入描述:

第一行,  $n, m, e$

第二至  $e+1$  行, 每行两个正整数  $u, v$ , 表示  $u, v$  有一条连边

$n, m \leq 1000, 1 \leq u \leq n, 1 \leq v \leq m, e \leq n \times m$

### 输出描述:

共一行, 二分图最大匹配

### 输入样例#:

1 1 1

1 1

### 输出样例#:

1

### 题目来源:

DreamJudge 1652

题目解析: 二分图模板题, 使用匈牙利算法解决。

### 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int N = 2e3 + 10;
5. int n, m, e;
6. vector<int> G[N]; //使用邻接表来储存边
7. int match[N], vis[N];
8. bool dfs(int u) {
9.     int len = G[u].size();
10.    for (int i = 0; i < len; i++) { //遍历每一条边
```

```
11.     int v = G[u][i];
12.     if (vis[v]) continue;
13.     vis[v] = 1;
14.     if (!match[v] || dfs(match[v])) { //如果 v 没有匹配, 或者 v 的匹配找到了新的匹配
15.         match[v] = u;
16.         match[u] = v; //更新匹配信息
17.         return 1;
18.     }
19. }
20. return 0;
21. }
22. int main() {
23.     scanf("%d %d %d", &n, &m, &e);
24.     for (int i = 1; i <= e; i++) {
25.         int a, b;
26.         scanf("%d %d", &a, &b);
27.         G[a].push_back(n + b);
28.         G[n + b].push_back(a);
29.     }
30.     int ans = 0;
31.     for (int i = 1; i <= n; i++) { //对每一个点尝试匹配
32.         for (int j = 1; j <= n + m; j++) vis[j] = 0;
33.         if (dfs(i)) ans++;
34.     }
35.     printf("%d", ans);
36.     return 0;
37. }
```

### 3.5 kmp 算法的应用

前面一节介绍了经典的 KMP 算法，这一节我们继续深入了解 KMP 有哪些常见应用方式。

#### 1、给出一个字符串 问它最多由多少相同的字串组成

如 abababab 由 4 个 ab 组成

解析：kmp 中的 next 数组求最小循环节的应用

#### 2、判断二叉树 B 是否为二叉树 A 的子树

解析：即将两颗二叉树序列化然后判断 B (tree2) 序列化的结果是否为 A (tree1) 序列化的子串即可

#### 3、判断一个字符串是否为一个子串重复得到

解析：和例 1 是一个意思

## 3.6 路径进阶问题

在高分篇中，我们讲解了最短路径的问题，这一节我们继续讲解路径相关的进阶问题。

### 1、最长路

一般是问有向无环图的最长路径是多少？

解析：由于是有向无环图，所以从入度为 0 的边进行动态规划或者 BFS 搜索都是可行的。

### 2、树的直径

一般是询问一棵树的最长路径或者说直径是多少？

解析：显然，树的直径是两个最远的端点经过的路径长度。我们可以从任意一点开始 BFS 或 DP 然后找到距离这个点最远的点，而这个最远的点一定是树的直径的其中一个端点。然后再从这个端点开始 BFS 或者 DP 找到距离这个端点最远的点，那么就把树的直径两个端点都找出来了，最长路径也就知道了。

### 3、次短路/K 短路

如果题目让你求次短路，也就是第二短的路径长度的话，应该怎么办呢？

解析：只要求一遍最短路，然后记录最短路经过了哪些边，然后枚举删除每一条边然后求最短路，显然，原图次短路一定是最短路删掉某一条边之后剩下的图的最短路。

注：一般考研机试不会让你求第三或第四或第 K 短的路，如果是保研机试且学校历年题目难度很大的学校可以备一份模板以防万一，一般使用 A\*算法进行优化。当然也可以用下面这个算法模板求次短路，即 K=2 的情况。

#### K 短路模板代码

```
1. #include<bits/stdc++.h>
```



```
2. using namespace std;
3.
4. #define INF 0x3f3f3f3f
5. const int maxn = 1005;
6. int n, m;
7. int dist[maxn]; // 存放起点到 i 点的最短距离
8. int vis[maxn]; // 标记是否访问过
9. int p[maxn]; // 存放路径
10.
11. struct Edge{
12.     int u, v, w;
13.     Edge(int u, int v, int w):u(u),v(v),w(w) {}
14. };
15.
16. struct node {
17.     int v;
18.     int g, f;
19.     node(int v, int g, int f):v(v),g(g),f(f) {}
20.     bool operator < (const node &t) const {
21.         if (t.f == f) return t.g < g;
22.         return t.f < f;
23.     }
24. };
25.
26. vector<Edge> edges;
27. vector<Edge> revedges;
28. vector<int> G[maxn];
29. vector<int> RG[maxn];
30.
31. queue<int> q;
32. void spfa(int s) {
33.     while (!q.empty()) q.pop();
34.     for (int i = 0; i <= n; i++) dist[i] = INF;
35.     dist[s] = 0;
36.     memset(vis, 0, sizeof(vis));
37.     q.push(s);
38.     while (!q.empty()) {
39.         int u = q.front(); q.pop();
40.         vis[u] = 0;
41.         for (int i = 0; i < G[u].size(); i++) {
42.             Edge& e = edges[G[u][i]];
43.             if (dist[e.v] > dist[u] + e.w) {
44.                 dist[e.v] = dist[u] + e.w;
```

```
45.         p[e.v] = G[u][i];
46.         if (!vis[e.v]) {
47.             vis[e.v] = 1;
48.             q.push(e.v);
49.         }
50.     }
51. }
52. }
53. }
54.
55. priority_queue<node> que;
56. int A_Star(int s, int t, int k) {
57.     while (!que.empty()) que.pop();
58.     que.push(node(s, 0, dist[s]));
59.     while (!que.empty()) {
60.         node now = que.top();
61.         que.pop();
62.         if (now.v == t) {
63.             if (k > 1) k--;
64.             else return now.g;
65.         }
66.         int u = now.v;
67.         for (int i = 0; i < RG[u].size(); i++) {
68.             Edge& e = edges[RG[u][i]];
69.             que.push(node(e.u, now.g + e.w, now.g + e.w + dist[e.u]));
70.         }
71.     }
72.     return -1;
73. }
74.
75. void addedge(int u, int v, int w) {
76.     edges.push_back(Edge(u, v, w));
77.     int sz = edges.size();
78.     G[u].push_back(sz - 1);
79. }
80.
81. void addrevedge(int u, int v, int w) {
82.     revedges.push_back(Edge(u, v, w));
83.     int sz = revedges.size();
84.     RG[u].push_back(sz - 1);
85. }
86.
87. void init() {
```

```

88.     for(int i = 0; i <= n; i++) {G[i].clear(); RG[i].clear();}
89.     edges.clear();
90.     revedges.clear();
91. }
92.
93. int main() {
94.     while (scanf("%d%d", &n, &m) != EOF) {
95.         init();
96.         for (int i = 0; i < m; i++) {
97.             int a, b, c;
98.             scanf("%d%d%d", &a, &b, &c);
99.             addrevedge(a, b, c);
100.            addedge(b, a, c);
101.        }
102.        int s, t, k;
103.        scanf("%d%d%d", &s, &t, &k); //起点、终点、第K短
104.        if (s == t) k++;
105.        spfa(t);
106.        int ans = A_Star(s, t, k);
107.        printf("%d\n", ans);
108.
109.    }
110.    return 0;
111. }

```

noobdream.com

#### 4、最小环

最小环是指在一个图中，有  $n$  个节点构成的边权和最小的环 ( $n \geq 3$ )。

求最小环可以用很多种方法，不管是暴力的枚举删边还是用 Dijkstra，又或者是 Floyd 等等都可以。

下面我们给出一个比较好的解决方案，使用 Floyd 求解。

```

1.  //floyd 求最小环
2.  int Floyd_MinCircle() {
3.      int Mincircle = Mod;
4.      int i, j, k;
5.      for (k = 1; k <= n; k++) {
6.          for (i = 1; i <= n; i++) {
7.              for (j = 1; j <= n; j++) {

```

```

8.         if (dis[i][j] != Mod && mp[j][k] != Mod && mp[k][i] != Mod && dis[i][j] + m
           p[j][k] + mp[k][i] < Mincircle)
9.             Mincircle = dis[i][j] + mp[j][k] + mp[k][i];
10.        }
11.    }
12.    //正常 Floyd
13.    for (i = 1; i <= n; i++) {
14.        for (j = 1; j <= n; j++) {
15.            if (dis[i][k] != Mod && dis[k][j] != Mod && dis[i][k] + dis[k][j] < dis[i][
                j]) {
16.                dis[i][j] = dis[i][k] + dis[k][j];
17.                pre[i][j] = pre[k][j];
18.            }
19.        }
20.    }
21. }
22. return Mincircle;
23. }

```



## 5、路径计数类问题

还有一类问题是路径计数，即统计满足题目要求的路径有多少条。

**问题：**一个  $N \times M$  的网格中，从左上角走到右下角共有多少条不同的路径可以走？每次只能往下或往右走。

**解析：**很明显，对于任意一点要么从它的上面过来，要么从它的左边过来，所以我们可以写出递推式子： $dp[i][j] = dp[i-1][j] + dp[i][j-1]$ ，其中  $dp[i][j]$  表示从起点走到点  $(i, j)$  的不同路径数量。那么我们可以用动态规划或者记忆化搜索的方式轻松写出这个程序。

### 路径计数 2

#### 题目描述：

一个  $N \times N$  的网格，你一开始在  $(1, 1)$ ，即左上角。每次只能移动到下方相邻的格子或者右方相

邻的格子，问到达  $(N, N)$ ，即右下角有多少种方法。

但是这个问题太简单了，所以现在有  $M$  个格子上有障碍，即不能走到这  $M$  个格子上。

#### 输入描述:

输入文件第 1 行包含两个非负整数  $N, M$ ，表示了网格的边长与障碍数。

接下来  $M$  行，每行两个不大于  $N$  的正整数  $x, y$ 。表示坐标  $(x, y)$  上有障碍不能通过，且有  $1 \leq x, y \leq n$ ，且  $x, y$  至少有一个大于 1，并注意障碍坐标有可能相同。

$N \leq 1000, M \leq 100000$

#### 输出描述:

一个非负整数，为答案  $\text{mod} 100003$  后的结果。

#### 输入样例#:

3 1

3 1

#### 输出样例#:

5

#### 题目来源:

DreamJudge 1600

**题目解析:** 对于路径计数类的问题我们要先去找寻它内在的联系规律，这个题和上面的例题相比多了一些障碍物，只是一个简单的小变形，我们只需要标记一下障碍物的位置，然后判断障碍物的坐标是不能走，即不能递推的就行了，就可以用动态规划或记忆化搜索来实现。

#### 参考代码

```
1. #include <stdio.h>
2.
3. int a[1005][1005], b[1005][1005];
4. int main() {
5.     int n, m, x, y;
6.     scanf("%d%d", &n, &m);
7.     for(int i = 1; i <= m; i++) {
```

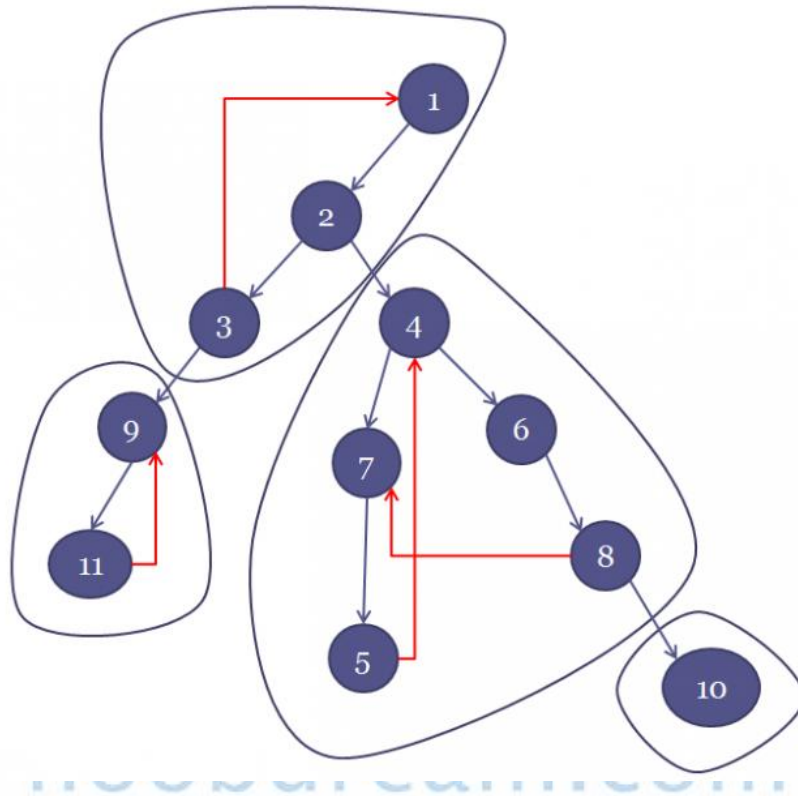
```
8.         scanf("%d%d", &x, &y);
9.         b[x][y] = 1; //b 数组标记不可以走的
10.    }
11.    a[1][1] = 1; //最开始 a[1][1]
12.    for(int i = 1; i <= n; i++) {
13.        for(int j = 1; j <= n; j++) {
14.            a[i][j] += a[i-1][j] + a[i][j-1]; //赋值
15.            if(b[i][j] == 1) a[i][j] = 0; //不可以走的赋值 0
16.            a[i][j] = a[i][j] % 100003; //取%
17.        }
18.    }
19.    printf("%d\n", a[n][n]);
20.    return 0;
21. }
```



### 3.7 图的连通性问题

图的连通性是一个常见的考点，如果是单纯的判断一个图是否联通或者有几个联通块都可以直接用 DFS 搜索来解决。

但是图的联通性考点有一个较为复杂的考点是缩点。



缩点

#### 题目描述:

给定一个  $n$  个点  $m$  条边有向图，每个点有一个权值，求一条路径，使路径经过的点权值之和最大。你只需要求出这个权值和。

允许多次经过一条边或者一个点，但是，重复经过的点，权值只计算一次。

#### 输入描述:

第一行两个正整数  $n, m$

第二行  $n$  个整数，依次代表点权

第三至  $m+2$  行，每行两个整数  $u, v$ ，表示一条  $u \rightarrow v$  的有向边。

$1 \leq n \leq 10^4$

$1 \leq m \leq 10^5$

点权  $\in [0, 1000]$

**输出描述:**

共一行，最大的点权之和。

**输入样例#:**

2 2

1 1

1 2

2 1

**输出样例#:**

2

**题目来源:**

DreamJudge 1613

**题目解析:** 首先我们想到如果是一个无向图，要求一条最长路径非常简单，在高分篇的最短路算法就讲解了这种方法，也可以直接搜索解决。但是这个题是有向图，很多同学就没有头绪了，那么我就需要将有向图缩点转化为无向图，然后就可以解出答案了，在这里我们用到了一个经典的算法叫做 Tarjan 算法。

**参考代码**

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 1e5 + 1e4, maxm = 1e5 + 1e4;
5. int Index, pd[maxn], DFN[maxn], LOW[maxn];
6. int tot, color[maxn], sum[maxn], f[maxn];
7. int edge, fir[maxn], Next[maxm], to[maxm];
8. int sta[maxn], top; //手写栈
9. int n, m, val[maxn], x[maxm], y[maxm], ans;
10. int read(){
11.     int x=0, f=1; char ch=getchar();
12.     while(ch<'0' || ch>'9'){if(ch=='-')f=-1; ch=getchar();}
13.     while(ch>='0' && ch<='9'){x=x*10+ch-48; ch=getchar();}
```



```
14.     return f*x;
15. }
16. void add(int x,int y){
17.     to[++edge]=y; Next[edge]=fir[x]; fir[x]=edge;
18. }
19. void tarjan(int x){
20.     sta[++top]=x;
21.     pd[x]=1;
22.     DFN[x]=LOW[x]= ++Index;
23.     for(int i=fir[x];i;i=Next[i]){
24.         int v=to[i];
25.         if(!DFN[v]){
26.             tarjan(v);
27.             LOW[x]=min(LOW[x],LOW[v]);
28.         }
29.         else if(pd[v]){
30.             LOW[x]=min(LOW[x],DFN[v]);
31.         }
32.     }
33.     if(DFN[x]==LOW[x]){
34.         tot++;
35.         while(sta[top+1]!=x){
36.             color[sta[top]]=tot;
37.             sum[tot]+=val[sta[top]];
38.             pd[sta[top--]]=0;
39.         }
40.     }
41. }
42. void search(int x){    //在 DAG 上记忆化搜索
43.     if(f[x]) return ;
44.     f[x]=sum[x];
45.     int maxsum = 0;
46.     for(int i=fir[x];i;i=Next[i]){
47.         if(!f[to[i]]) search(to[i]);
48.         maxsum=max(maxsum,f[to[i]]);
49.     }
50.     f[x]+=maxsum;
51. }
52. int main(){
53.     n=read(); m=read();
54.     for(int i=1;i<=n;i++) val[i]=read();
55.     for(int i=1;i<=m;i++){
```

```
56.         x[i]=read();                                // 这里用数组保存好原边, 便于后续重新建
   图;
57.         y[i]=read();
58.         add(x[i],y[i]);
59.     }
60.     for(int i=1;i<=n;i++) if(!DFN[i]) tarjan(i);
61.     memset(fir,0,sizeof(fir));                        //清空原图
62.     memset(Next,0,sizeof(Next));
63.     memset(to,0,sizeof(to));
64.     edge=0;
65.     for(int i=1;i<=m;i++){                            //重新建图(枚举每一条原边, 若不在同一强连通分量里, 则连一条边(方
   向同原边))
66.         if(color[x[i]]!=color[y[i]])
67.             add(color[x[i]],color[y[i]]);
68.     }
69.     for(int i=1;i<=tot;i++){
70.         if(!f[i]){
71.             search(i);
72.             ans=max(ans,f[i]);
73.         }
74.     }
75.     printf("%d",ans);
76.     return 0;
77. }
```

noobdream.com

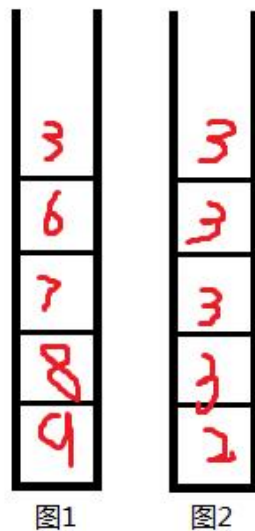
### 3.8 单调队列/栈

#### 单调栈

单调栈是指一个栈内部的元素具有严格单调性的一种数据结构, 分为单调递增栈和单调递减栈。

其具有以下两个性质:

- 1、满足栈底到栈顶的元素具有严格单调性。
- 2、满足栈的先进后出特性, 越靠近栈顶的元素越后出栈。



如图所示:

图1所示栈为单调递增栈。

图2所示栈既不是单调递减栈, 也不是单调递增栈。

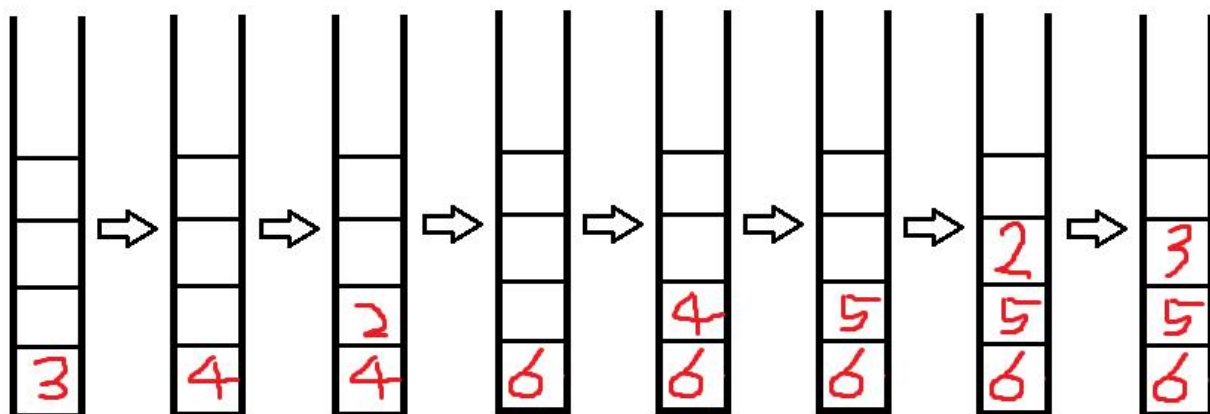
元素进栈过程:

对于一个单调递增栈来说, 若当前进栈的元素为  $a$ , 如果  $a <$  栈顶元素, 则直接将  $a$  进栈。如果  $a \geq$  栈顶元素, 则不断将栈顶元素出栈, 直到满足  $a <$  栈顶元素。

模拟一个数列构造一个单调递增栈

进栈元素分别为 3, 4, 2, 6, 4, 5, 2, 3

图片所示过程即为进栈过程。



## 单调队列

单调队列与单调栈及其相似，把单调栈先进后出的性质改为先进先出既可。

### 单调递增队列元素进队列的过程

对于一个元素  $a$ ，如果  $a >$  队尾元素，那么直接将  $a$  扔进队列，如果  $a \geq$  队尾元素，则将队尾元素出队列，直到满足  $a >$  队尾元素即可。

由于双端队列即可以在队头操作，也可以在队尾操作，那么这样的性质就弥补了单调栈只能在一边操作的不足。可以使得其左边也有一定的限制。

### 时间复杂度分析

对于每个元素，其有且仅有一次插入，最多出现一次删除，故其时间复杂度为  $O(n)$ 。

## 练习

给你  $n$  个数，让你在这  $n$  个数中选出连续的  $m$  个数 ( $m \leq n$ )，使这  $m$  个数的极差最小，若存在多个区间使得极差均最小，输出最靠前的区间。

很显然， $n \leq 10^4$  时暴力明显可做， $n \leq 10^6$  时通过线段树也可做，那如果  $n$  去到  $10^7$  呢？

解析:

我们考虑用单调队列维护区间 $[i, i+m-1]$ 的最小值和最大值, 我们以维护最大值举例。

首先, 我们把前  $m$  个数扔进一个单调递增队列中, 在扔进去的同时把这些数所对应的下边也扔进去。显然队头的数字即为区间 $[1, m]$ 最大的数。

考虑基于 $[1, m]$ 的数据去更新 $[2, m+1]$ 的最大值。若第一个数依然存在于队列中(很显然若存在仅可能位于队尾), 将这个数删除, 然后将第  $m+1$  个数插入改单调队列。显然队尾数字即为区间 $[2, m+1]$ 的最大值。

重复该过程  $n-m+1$  次即可, 显然时间复杂度为  $O(n)$ 。



## 第四章 满分之路下

高分篇的内容已经将机试可能涉及到的题型及考点都一一列举出来, 并给大家提供了练习题目, 帮助大家巩固基础加深理解。在大多数院校的机试中, 学会高分篇的内容已经足够考到 90 分以上的成绩, 题目不难的情况下甚至能拿到 100 分。在少数院校难度较大的院校也足够拿到 80 分左右的分数, 如果你基础比较薄弱, 我们建议你掌握高分篇的内容就可以了。本书满分之路的内容不是常考点, 只是有一定可能会考到, 我们建议基础好的同学继续学习增加拿满分的把握。

距离满分只有一步之遥, 你! 愿意放弃吗?

本章我们重点讲解一些较为深入的题型, 包括容斥与抽屉原理、除法取模问题、组合数取模问题、矩阵快速幂、带状态压缩的搜索和数位类型动态规划等内容。希望能帮助读者更好的掌握计算机考研机试中所涉及到的各类较难的问题。

**提示: 满分篇的内容难度大选择性的学就行, 不要求全部掌握, 把能学会的学会就可以了。**

本书配套视频精讲: <https://www.bilibili.com/video/av91373687>

## 4.1 容斥与抽屉原理

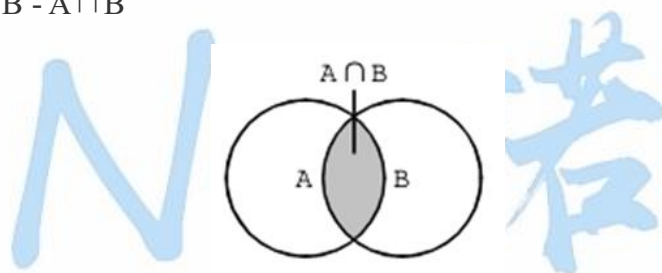
### 容斥原理

在计数时, 要保证无一重复, 无一遗漏。为了使重叠部分不被重复计算, 在不考虑重叠的情况下, 把包含于某内容中的所有对象的数目先计算出来, 然后再把计数时重复计算的数目排斥出去, 使得计算的结果既无遗漏又无重复, 这种计数的方法称为容斥原理。

#### 1. 容斥原理 1——两个集合的容斥原理

如果被计数的事物有 A、B 两类, 那么, 先把 A、B 两个集合的元素个数相加, 发现既是 A 类又是 B 类的部分重复计算了一次, 所以要减去。如图所示:

$$\text{公式: } A \cup B = A + B - A \cap B$$



总数 = 两个圆内的 - 重合部分的

【例 1】一次期末考试, 某班有 15 人数学得满分, 有 12 人语文得满分, 并且有 4 人语、数都是满分, 那么这个班至少有一门得满分的同学有多少人?

解: 数学得满分人数  $\rightarrow A$ , 语文得满分人数  $\rightarrow B$ , 数学、语文都是满分人数  $\rightarrow A \cap B$ , 至少有一门得满分人数  $\rightarrow A \cup B$ 。 $A \cup B = 15 + 12 - 4 = 23$ , 共有 23 人至少有一门得满分。

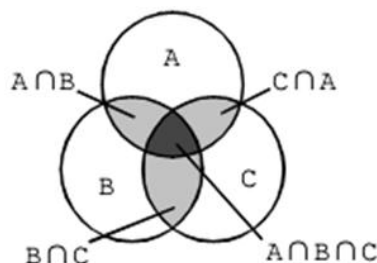
#### 2. 容斥原理 2——三个集合的容斥原理

如果被计数的事物有 A、B、C 三类, 那么, 将 A、B、C 三个集合的元素个数相加后发现两两重叠的部分重复计算了 1 次, 三个集合公共部分被重复计算了 2 次。

如图所示, 灰色部分  $A \cap B - A \cap B \cap C$ 、 $B \cap C - A \cap B \cap C$ 、 $C \cap A - A \cap B \cap C$  都被重复计算了 1 次, 黑色部分  $A \cap B \cap C$  被重复计算了 2 次, 因此总数  $A \cup B \cup C = A + B + C - (A \cap B - A \cap B \cap C) - (B \cap C - A \cap B \cap C) - (C \cap A - A \cap B \cap C) - 2(A \cap B \cap C)$

$\cap C) - (B \cap C - A \cap B \cap C) - (C \cap A - A \cap B \cap C) - 2A \cap B \cap C = A + B + C - A \cap B - B \cap C - C \cap A + A \cap B \cap C$ 。即得到：

公式： $A \cup B \cup C = A + B + C - A \cap B - B \cap C - C \cap A + A \cap B \cap C$



总数 = 三个圆内的 - 重合两次的 + 重合三次的

【例 2】某班有学生 45 人，每人都参加体育训练队，其中参加足球队的有 25 人，参加排球队的有 22 人，参加游泳队的有 24 人，足球、排球都参加的有 12 人，足球、游泳都参加的有 9 人，排球、游泳都参加的有 8 人，问：三项都参加的有多少人？

参加足球队  $\rightarrow A$ ，参加排球队  $\rightarrow B$ ，参加游泳队  $\rightarrow C$ ，足球、排球都参加的  $\rightarrow A \cap B$ ，足球、游泳都参加的  $\rightarrow C \cap A$ ，排球、游泳都参加的  $\rightarrow B \cap C$ ，三项都参加的  $\rightarrow A \cap B \cap C$ 。三项都参加的有  $A \cap B \cap C = A \cup B \cup C - A - B - C + A \cap B + B \cap C + C \cap A = 45 - 25 - 22 - 24 + 12 + 9 + 8 = 3$  人。

## 抽屉原理

能利用抽屉原理来解决的问题称为抽屉问题，抽屉原理又名鸽巢原理。

### 抽屉原理 1

将多于  $n$  件的物品任意放到  $n-1$  个抽屉中，那么至少有一个抽屉中的物品件数不少于 2。(至少有 2 件物品在同一个抽屉)

### 抽屉原理 2

将多于  $m \times n$  件的物品任意放到  $n$  个抽屉中，那么至少有一个抽屉中的物品的件数不少于  $m+1$ 。(至少有  $m+1$  件物品在同一个抽屉)



## 死神来了

### 题目描述:

有一天, 王小子在遨游世界时, 遇到了一场自然灾害。一个人孤独的在一个岛上, 没有吃的没有喝的。在他饥寒交迫将要死亡时, 死神来了。由于这个死神在成神之前是一个数学家, 所以他有一个习惯, 会和即死之人玩一个数学游戏, 来决定是否将其灵魂带走。游戏规则是死神给王小子两个整数  $n$  ( $100 \leq n \leq 1000000$ ),  $m$  ( $2 \leq m \leq n$ ), 在  $1 \sim n$  个数中, 随机取  $m$  个数, 问在这  $m$  个数中是否一定存在一个数是另一个数的倍数, 是则回答 "YES", 否则 "NO"。如果王小子回答正确, 将有再活下去的机会。但是他很后悔以前没有好好学习数学, 王小子知道你数学学得不错, 请你救他一命。

### 输入描述:

有多组测试数据, 不多于 10000;

每组有两个数  $n, m$ ;

以文件结束符 EOF 为结束标志。

### 输出描述:

输出 "YES" 或 "NO"。

### 输入样例#:

100 80

100 20

### 输出样例#:

YES

NO

### 题目来源:

DreamJudge 1723

**题目解析:** 这题用到了鸽笼原理 (有  $n+1$  件或  $n+1$  件以上的物品要放到  $n$  个抽屉中, 那么至少有一个抽屉里有两个 或两个以上物品。), 在本题, 我们  $m$  看作为  $m$  个鸽笼, 我们将  $1$  到  $N$  所有数的倍数分组, 最大分组数为  $p = N/2 + (N \& 1)$ , 看作  $p$  只鸽子, 问是否能把这  $p$  只鸽子 放入到  $m$  个笼子里, 保证每只笼子只有一只鸽子。比如 100 的分组为, 此处运用二进制

的思想

$A1 = \{1, 12, 14 \cdots 164\}$

$A2 = \{3, 32, 34 \cdots 332\}$

...

...

$A25 = \{49, 49 \times 2\}$

...

...

$A50 = \{99\}$

细心的同学可能已经发现了每个子集都是奇数开头的, 而且在一个集合中, 后一个元素总是前一个元素的 2 倍, 这样就能够保证该集合中任意两个元素的, 其中一个元素是另外一个元素的倍数, 而且因为每次都要乘 2, 所以每个集合除了开头是奇数, 后面的元素均为偶数, 这样也方便我们算有多少个集合, 即有多少个抽屉

所以公式为:  $n = (n >> 1) + (n \& 1)$

### 参考代码

```
1. #include<iostream>
2. using namespace std;
3.
4. int main() {
5.     int n, m;
6.     while(cin >> n >> m) {
7.         n = (n >> 1) + (n & 1);
8.         if (m > n)
9.             cout << "YES" << endl;
10.        else
11.            cout << "NO" << endl;
12.    }
13.    return 0;
14. }
```

## 4.2 除法取模问题

我们经常在做题时会看到这样一句话：由于答案较大，请输出答案  $\text{mod } m$  的结果。（其中  $m$  一般为一个大质数）

我们经常会使用以下几个等式：

$$(a + b) \equiv (a \% m + b \% m) (\% m)$$

$$(a + b) \equiv (a \% m + b \% m) (\% m)$$

$$(a - b) \equiv (a \% m - b \% m + m) (\% m) (a > b)$$

$$(a - b) \equiv (a \% m - b \% m + m) (\% m) (a > b)$$

$$(a \times b) \equiv (a \% m \times b \% m) (\% m)$$

$$(a \times b) \equiv (a \% m \times b \% m) (\% m)$$

但是很容易发现，这三个等式中并没有除法。

那我们怎样处理除法呢？这里就要使用到逆元。

noobdream.com

### 逆元

我们定义若  $(a * x) \% m = 1$ ，则称  $x$  为  $a$  模  $m$  的乘法逆元。

并且有  $(a / b) \% c = (a * x) \% c$ 。（其中  $a \div b$  为整除）

逆元一般用扩展欧几里得算法来求得，如果  $m$  为素数，那么还可以根据费马小定理得到逆元为  $a^{m-2} \text{mod } m$ 。（都要求  $a$  和  $m$  互质）

### 证明

费马小定理:对于素数  $M$  任意不是  $M$  的倍数的  $b$ ，都有： $b^{(M-1)} = 1 \pmod{M}$

于是可以拆成： $b * b^{(M-2)} = 1 \pmod{M}$

于是： $a/b = a/b * (b * b^{(M-2)}) = a * (b^{(M-2)}) \pmod{M}$ ，这里不就是用  $b^{(M-2)}$  代替了  $1/b$  吗！

## 扩展欧几里得解法

证明（略）

### 参考代码

```
1. #include<stdio>
2. typedef long long LL;
3.
4. LL inv(LL t, LL p) { //求 t 关于 p 的逆元, 注意:t 要小于 p, 最好传参前先把 t%p 一下
5.     return t == 1 ? 1 : (p - p / t) * inv(p % t, p) % p;
6. }
7.
8. int main() {
9.     LL a, p;
10.    while(~scanf("%lld%lld", &a, &p)) {
11.        printf("%lld\n", inv(a%p, p));
12.    }
13. }
```

它可以在  $O(n)$  的复杂度内算出  $n$  个数的逆元

```
1. #include<stdio>
2.
3. const int N = 200000 + 5;
4. const int MOD = (int)1e9 + 7;
5. int inv[N];
6. int init() { //O(n)复杂度求出所有数的逆元
7.     inv[1] = 1;
8.     for(int i = 2; i < N; i++)
9.         inv[i] = (MOD - MOD / i) * 1ll * inv[MOD % i] % MOD;
10. }
11.
12. int main() {
13.     init();
14. }
```

## 4.3 组合数取模类问题

在考研机试中，我们经常会遇到一类找规律的题目，如统计路径数量，由于数据过大，输出对某个数  $p$  的取模答案。而我们往往会推出一个组合数的公式，但却无从下手。

组合数取模就是求  $C_n^m \% p$  的值，当然根据  $n$ ， $m$  和  $p$  的取值范围不同，采取的方法也不一样。

1、当  $n, m$  都很小的时候可以利用杨辉三角直接求 ( $1 \leq n, m \leq 1000$  且  $1 \leq p \leq 1e9$ )。

$$C(n, m) = C(n-1, m) + C(n-1, m-1)$$

由于  $n$  和  $m$  的范围小，直接两重循环即可。

```
1. #include<bits/stdc++.h>
2. using namespace std;
3. typedef long long ll;
4.
5. const int mod=1e9+7;
6. const int N=10000+5;
7. int comb[N][N]; //comb[i][j]内存放的是 C(i,j)%mod
8. void init() {
9.     for(int i=0; i<N; i++) {
10.         comb[i][i]=1;
11.         comb[i][0]=1;
12.         for(int j=1; j<i; j++) {
13.             comb[i][j]=comb[i-1][j]+comb[i-1][j-1];
14.             if(comb[i][j]>=mod)
15.                 comb[i][j]-=mod;
16.         }
17.     }
18. }
19.
20. int main() {
21.     init();
22.     int n, m;
23.     scanf("%d%d", &n, &m);
24.     printf("%d\n", comb[n][m]);
25.     return 0;
26. }
```

## 2、n 和 m 较大，但是 p 为素数的时候 ( $1 \leq n, m \leq 1e18$ )

我们使用 Lucas 定理来求解组合数取模  $C_n^m \% p$  的问题

简略推导

$$C(n, m) \% p = C(n/p, m/p) * C(n \% p, m \% p) \% p$$

$$\text{也就是 } \text{Lucas}(n, m) \% p = \text{Lucas}(n/p, m/p) * C(n \% p, m \% p) \% p$$

求上式的时候，Lucas 递归出口为  $m = 0$  时返回 1

求  $C(n \% p, m \% p) \% p$  的时候，此处写成  $C(n, m) \% p$  ( $p$  是素数， $n$  和  $m$  均小于  $p$ )

$$C(n, m) \% p = n! / (m! * (n - m)!) \% p = n! * \text{mod\_inverse}[m! * (n - m)!, p] \% p$$

由于  $p$  是素数，有费马小定理可知， $m! * (n - m)!$  关于  $p$  的逆元就是  $m! * (n - m)!$  的  $p-2$  次方。

如果  $P$  较小 ( $2 \leq p \leq 1e5$ )，参考代码如下

```
1. #include<bits/stdc++.h>
2. using namespace std;
3. typedef long long ll;
4.
5. const int maxn = 1e5 + 10;
6. ll fac[maxn]; //阶乘打表
7. void init(ll p) //此处的 p 应该小于 1e5，这样 Lucas 定理才适用
8. {
9.     fac[0] = 1;
10.    for(int i = 1; i <= p; i++)
11.        fac[i] = fac[i - 1] * i % p;
12. }
13. ll pow(ll a, ll b, ll m)
14. {
15.     ll ans = 1;
16.     a %= m;
17.     while(b)
18.     {
19.         if(b & 1) ans = (ans % m) * (a % m) % m;
20.         b /= 2;
21.         a = (a % m) * (a % m) % m;
22.     }
23.     ans %= m;
24.     return ans;
25. }
26. ll inv(ll x, ll p) //x 关于 p 的逆元，p 为素数
```

```

27. {
28.     return pow(x, p - 2, p);
29. }
30. ll C(ll n, ll m, ll p)//组合数 C(n, m) % p
31. {
32.     if(m > n)return 0;
33.     return fac[n] * inv(fac[m] * fac[n - m], p) % p;
34. }
35. ll Lucas(ll n, ll m, ll p)
36. {
37.     if(m == 0)return 1;
38.     return C(n % p, m % p, p) * Lucas(n / p, m / p, p) % p;
39. }
40.
41. int main() {
42.     ll n, m, p;
43.     scanf("%lld%lld%lld", &n, &m, &p);
44.     init(p);//要先初始化
45.     ll ans = Lucas(n, m, p);
46.     printf("%lld\n", ans);
47.     return 0;
48. }

```

如果 P 较大, 参考代码如下

```

1. #include<bits/stdc++.h>
2. using namespace std;
3. typedef long long ll;
4.
5. ll pow(ll a, ll b, ll m)
6. {
7.     ll ans = 1;
8.     a %= m;
9.     while(b)
10.    {
11.        if(b & 1)ans = (ans % m) * (a % m) % m;
12.        b /= 2;
13.        a = (a % m) * (a % m) % m;
14.    }
15.    ans %= m;
16.    return ans;

```

```
17. }
18. ll inv(ll x, ll p)//x 关于 p 的逆元, p 为素数
19. {
20.     return pow(x, p - 2, p);
21. }
22. ll C(ll n, ll m, ll p)//组合数 C(n, m) % p
23. {
24.     if(m > n)return 0;
25.     ll up = 1, down = 1;//分子分母;
26.     for(int i = n - m + 1; i <= n; i++)up = up * i % p;
27.     for(int i = 1; i <= m; i++)down = down * i % p;
28.     return up * inv(down, p) % p;
29. }
30. ll Lucas(ll n, ll m, ll p)
31. {
32.     if(m == 0)return 1;
33.     return C(n % p, m % p, p) * Lucas(n / p, m / p, p) % p;
34. }
35.
36. int main() {
37.     ll n, m, p = 1e9 + 7;
38.     scanf("%lld%lld", &n, &m);
39.     ll ans = Lucas(n, m, p);
40.     printf("%lld\n", ans);
41.     return 0;
42. }
```

小结：第二种方法比第一种方法要慢一些，但是通用性更强，一般在考研机试中使用第二种方法即可。



## 4.4 矩阵快速幂

我们在高分篇中学习了二分快速幂的算法，它主要是解决  $x^y \% p$  这样一类问题，当  $y$  的值很大的时候，我们不能直接 for 循环来暴力求解，这样会超时，于是就有了二分快速幂的算法，它的时间复杂度是  $O(\log n)$ ，可以在极短的时间内求出答案。

那么，当我们的  $x$  不是一个普通的整数，而是一个矩阵  $A$  的话，又应该如何求解呢？

问题就变成了  $A^y \% p = ?$

通过观察，我们可以发现，矩阵的乘法也适用于这样的思想。

### 矩阵乘法基础

简单的说矩阵就是二维数组，数存在里面，矩阵乘法的规则： $A * B = C$

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$

其中  $c[i][j]$  为  $A$  的第  $i$  行与  $B$  的第  $j$  列对应乘积的和，即：
$$c_{ij} = \sum_{k=1}^n a_{ik} * b_{kj}$$

### 斐波那契数列加强版

#### 题目描述：

我们知道斐波那契数列的公式是：

$$f(n) = f(n-1) + f(n-2)$$

其中  $f(1) = 1$ ,  $f(2) = 1$ 。

#### 输入描述：

输入一个正整数  $n$  ( $n \leq 1e9$ )

#### 输出描述：

输出  $f(n) \% (1e9+7)$  的值

输入样例#:

5

输出样例#:

5

题目来源:

DreamJudge 1724

题目解析: 由于我们要求  $10^9$  那么多项的值, 直接递推或者递归都是会超时的。

于是我们对斐波那契数列公式  $f(n) = f(n-1) + f(n-2)$  构造一个矩阵递推式:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} f_{n-1} \\ f_{n-2} \end{bmatrix} = \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix}$$

于是我们可以发现, 当我们要求  $f(k)$  的时候

左边的系数矩阵会乘  $k$  次, 再乘上右边的初始矩阵, 就可以得到右边的最终值的矩阵。

由于  $k$  很大, 所以我们需要对  $k$  使用二分快速幂的思想, 从而很短的时间内求出结果。

参考代码

```
1. #include<stdio.h>
2. #include<cstring>
3. #define LL long long
4. #define M 10
5. #define MOD 1000000007
6. struct Matrix{
7.     LL matrix[M][M];
8. };
9. int n;//矩阵的阶数
10. void init(Matrix &res){
11.     for(int i = 0; i <= n; i++){
12.         for(int j = 0; j <= n; j++){
13.             res.matrix[i][j] = 0;
14.             res.matrix[i][i] = 1;
15.         }
16. }
```

```

17. Matrix multiplicative(Matrix a, Matrix b){ //矩阵乘法
18.     Matrix res;
19.     memset(res.matrix, 0, sizeof(res.matrix));
20.     for(int i = 0 ; i < n ; i++)
21.         for(int j = 0 ; j < n ; j++)
22.             for(int k = 0 ; k < n ; k++)
23.                 res.matrix[i][j] = (res.matrix[i][j]%MOD + a.matrix[i][k]%MOD * b.matrix[k]
                [j]%MOD)%MOD;
24.     return res;
25. }
26. Matrix pow(Matrix mx, int m){ //快速幂
27.     Matrix res, base = mx;
28.     init(res); //初始为单位矩阵, 即除主对角线都是 1 外, 其他都是 0
29.     while(m){
30.         if(m & 1)
31.             res = multiplicative(res, base);
32.         base = multiplicative(base, base);
33.         m >>= 1;
34.     }
35.     return res;
36. }
37. int main(){
38.     int m;
39.     n = 2;
40.     scanf("%d", &m);
41.     Matrix base, res;
42.     base.matrix[0][0] = base.matrix[0][1] = base.matrix[1][0] = 1;
43.     base.matrix[1][1] = 0;
44.     res=pow(base, m-1);
45.     if(m == 0)
46.         puts("0");
47.     else if(m == 1)
48.         puts("1");
49.     else
50.         printf("%lld\n", res.matrix[0][0]%MOD);
51.     return 0;
52. }

```

小结: 同学们可以发现, 对于这一类有规律的题目关键在于构造出系数矩阵, 然后套上矩阵快速幂的模板即可求解。

## 4.5 带状态压缩的搜索

在高分篇中，我们已经学会了基础的搜索方法以及相关的剪枝技巧。

但是在有一类常见的搜索中，我们需要学会使用状态压缩的技巧去优化搜索过程。通俗点说，就是利用二进制的优越性可以节约空间，也可以加快运算时间，从而达到更优的效果。

### 胜利大逃亡(续)

#### 题目描述:

Ignatius 再次被魔王抓走了(搞不懂他咋这么讨魔王喜欢)……

这次魔王汲取了上次的教训，把 Ignatius 关在一个  $n*m$  的地牢里，并在地牢的某些地方安装了带锁的门，钥匙藏在地牢另外的某些地方。刚开始 Ignatius 被关在  $(sx, sy)$  的位置，离开地牢的门在  $(ex, ey)$  的位置。Ignatius 每分钟只能从一个坐标走到相邻四个坐标中的其中一个。魔王每  $t$  分钟回地牢视察一次，若发现 Ignatius 不在原位置便把他拎回去。经过若干次的尝试，Ignatius 已画出整个地牢的地图。现在请你帮他计算能否再次成功逃亡。只要在魔王下次视察之前走到出口就算离开地牢，如果魔王回来的时候刚好走到出口或还未到出口都算逃亡失败。

#### 输入描述:

每组测试数据的第一行有三个整数  $n, m, t$  ( $2 \leq n, m \leq 20, t > 0$ )。接下来的  $n$  行  $m$  列为地牢的地图，其中包括：

. 代表路

\* 代表墙

@ 代表 Ignatius 的起始位置

^ 代表地牢的出口

A-J 代表带锁的门，对应的钥匙分别为 a-j

a-j 代表钥匙，对应的门分别为 A-J

每组测试数据之间有一个空行。

### 输出描述:

针对每组测试数据, 如果可以成功逃亡, 请输出需要多少分钟才能离开, 如果不能则输出-1。

### 输入样例#:

```
4 5 17
```

```
@A. B.
```

```
a*. *.
```

```
*. * ^
```

```
c.. b*
```

```
4 5 16
```

```
@A. B.
```

```
a*. *.
```

```
*. * ^
```

```
c.. b*
```

### 输出样例#:

```
16
```

```
-1
```

### 题目来源:

DreamJudge 1725

**题目解析:** 主要问题就是状态表示。总共钥匙数不超过 10 把, 容易想到状态压缩。当遇到一把新钥匙时, 通过 “|” 运算来改变状态, 当遇到门时, 用 “&” 运算来判断是否已经拥有该门的钥匙。

### 参考代码

```
1. #include <stdio>
2. #include <cstring>
3. #include <queue>
4. #include <algorithm>
5. using namespace std;
6.
7. const int maxn = 25;
```

```

8.  char mpt[maxn][maxn];
9.  int vis[maxn][maxn][1 << 10]; //在普通搜索上增加一维表示带的钥匙情况
10. int dir[4][2] = {1, 0, 0, 1, 0, -1, -1, 0};
11. int n, m, t;
12. struct node {
13.     int x, y;
14.     int step;
15.     int state;
16. };
17.
18. int bfs(int sx, int sy) {
19.     queue<node> q;
20.     q.push(node{sx, sy, 0, 0});
21.     memset(vis, 0, sizeof(vis));
22.     vis[sx][sy][0] = 1;
23.     int ans = -1;
24.     while (!q.empty()) {
25.         node now = q.front();
26.         q.pop();
27.         if (now.step % t == 0 && (now.x != sx || now.y != sy)) continue;
28.         if (mpt[now.x][now.y] == '^') {
29.             ans = now.step; break;
30.         }
31.         for (int i = 0; i < 4; i++) {
32.             int nx = now.x + dir[i][0];
33.             int ny = now.y + dir[i][1];
34.             if (vis[nx][ny][now.state] == 1) continue; //已经走过这个状态
35.             if (mpt[nx][ny] == '*') continue; //墙不能走
36.             if (mpt[nx][ny] == '.' || mpt[nx][ny] == '^' \
37.                 || mpt[nx][ny] == '@') {
38.                 q.push(node{nx, ny, now.step + 1, now.state});
39.                 vis[nx][ny][now.state] = 1;
40.             }
41.             else if (mpt[nx][ny] >= 'A' && mpt[nx][ny] <= 'J') { //遇到门了
42.                 if (((1 << mpt[nx][ny] - 'A') & now.state) > 0) { //判断是否有对应的钥匙打开
43.                     q.push(node{nx, ny, now.step + 1, now.state});
44.                     vis[nx][ny][now.state] = 1;
45.                 }
46.             }
47.             else if (mpt[nx][ny] >= 'a' && mpt[nx][ny] <= 'j') { //遇到钥匙装进包里
48.                 q.push(node{nx, ny, now.step + 1, now.state | (1 << mpt[nx][ny] - 'a')});
49.                 vis[nx][ny][now.state | (1 << mpt[nx][ny] - 'a')] = 1;

```

```
50.     }
51.     }
52. }
53.     return ans;
54. }
55.
56. int main() {
57.     while (scanf("%d%d%d", &n, &m, &t) != EOF) {
58.         int sx = 1, sy = 1;
59.         memset(mpt, 0, sizeof(mpt));
60.         for (int i = 1; i <= n; i++) {
61.             scanf("%s", mpt[i] + 1);
62.             for (int j = 1; j <= m; j++) {
63.                 if (mpt[i][j] == '@') {
64.                     sx = i;
65.                     sy = j;
66.                 }
67.             }
68.         }
69.         int ans = bfs(sx, sy);
70.         printf("%d\n", ans);
71.     }
72.     return 0;
73. }
```

noobdream.com

## 4.6 数位类型动态规划

在考研机试中，往往会考察一类和数字相关的动态规划问题，即数位 DP。

数位 DP 有两种实现方式，一种是动态规划的常规的方式，另一种是使用记忆化搜索的方式。

一般而言，我们都使用记忆化搜索的方式来实现，因为它有固定的实现方式，同学们遇到其他题目的时候只需要修改一点地方即可套用，十分简洁方便。

### 模板代码

```

1.  typedef long long ll;
2.  int a[20];
3.  ll dp[20][state]; //不同题目状态不同
4.  ll dfs(int pos, /*state 变量*/, bool lead /*前导零*/, bool limit /*数位上界变量*/) //不是每个题都要判断前导零
5.  {
6.      //递归边界，既然是按位枚举，最低位是 0，那么 pos==-1 说明这个数我枚举完了
7.      if(pos==0) return 1; //这里一般返回 1，表示你枚举的这个数是合法的，那么这里就需要你在枚举时必须每一位都要满足题目条件，
8.      //也就是说当前枚举到 pos 位，一定要保证前面已经枚举的数位是合法的。不过具体题目不同或者写法不同的话不一定要返回 1
9.      //第二个就是记忆化(在此前可能不同题目还能有一些剪枝)
10.     if(!limit && !lead && dp[pos][state]!=-1) return dp[pos][state];
11.     //常规写法都是在没有限制的条件记忆化，这里与下面记录状态是对应，具体为什么是有条件的记忆化后面会讲
12.     int up=limit?a[pos]:9; //根据 limit 判断枚举的上界 up;
13.     ll ans=0;
14.     //开始计数
15.     for(int i=0; i<=up; i++) //枚举，然后把不同情况的个数加到 ans 就可以了
16.     {
17.         if() ...
18.         else if()...
19.         ans+=dfs(pos-1, /*状态转移*/, lead && i==0, limit && i==a[pos]) //最后两个变量传参都是这样写的
20.         /*这里还算比较灵活，不过做几个题就觉得这里也是套路了
21.         大概就是说，我当前数位枚举的数是 i，然后根据题目的约束条件分类讨论
22.         去计算不同情况下的个数，还要根据 state 变量来保证 i 的合法性，比如题目
23.         要求数位上不能有 62 连续出现，那么就是 state 就是要保存前一位 pre，然后分类，
24.         前一位如果是 6 那么这意味就不能是 2，这里一定要保存枚举的这个数是合法*/
25.     }

```



```

26. //计算完, 记录状态
27. if(!limit && !lead) dp[pos][state]=ans;
28. /*这里对应上面的记忆化, 在一定条件下时记录, 保证一致性, 当然如果约束条件不需要考虑 lead, 这里就
   是 lead 就完全不用考虑了*/
29. return ans;
30. }
31. ll solve(ll x){
32. int pos=0;
33. while(x)//把数位都分解出来
34. {
35. a[++pos]=x%10;
36. x/=10;
37. }
38. return dfs(pos-1/*从最高位开始枚举*/,/*一系列状态 */,true,true);//刚开始最高位都是有限制并
   且有前导零的, 显然比最高位还要高的一位视为 0 嘛
39. }
40. int main(){
41. ll le,ri;
42. while(~scanf("%lld%lld",&le,&ri))
43. {
44. //初始化 dp 数组为-1,计算[le,ri], 等价于[0,ri] - [0,le-1]
45. printf("%lld\n",solve(ri)-solve(le-1));
46. }
47. return 0;
48. }

```

noobdream.com

## 数字计数

### 题目描述:

给定两个正整数  $a$  和  $b$ , 求在  $[a, b]$  中的所有整数中, 每个数字 ( $0 \sim 9$ ) 各出现了多少次。

### 输入描述:

分别输入两个正整数  $a, b$ 。其中  $a < b$ 。

其中  $a, b \leq 10^{18}$

### 输出描述:

分别输出十个数字 ( $0 \sim 9$ ) 出现的次数。

### 输入样例#:

1 99

输出样例#:

9 20 20 20 20 20 20 20 20 20

题目来源:

DreamJudge 1707

题目解析: 数位 DP 裸题, 见参考代码。

参考代码

```
1. #include<bits/stdc++.h>
2. using namespace std;
3. typedef long long ll;
4.
5. ll a[20], dp[20][20];
6. //dp[i][j]:第 i 位, 已经有 j 个当前数字时共有多少当前数字
7. ll dfs(int now, int digit, ll sum, bool limit, bool zero) {
8.     //now==0 说明所有位数都搜过, 则返回当前 digit 出现次数 sum
9.     if(!now) return sum;
10.    //注意此处的判断条件
11.    if(!limit && !zero && dp[now][sum]!=-1) return dp[now][sum];
12.    ll up = 9, ans = 0;
13.    if(limit) up = a[now];
14.    //搜索下一位时, 注意 sum 要在无前导 0 时更新
15.    for(int i = 0; i <= up; i++)
16.        ans+=dfs(now-1,digit,sum+((!zero || i) && i==digit),limit && i==a[now],zero && !i);
17.    if(!limit && !zero) dp[now][sum] = ans;
18.    return ans;
19. }
20. ll DP(ll x, int digit) {
21.    //注意本题中每次 DP 都要 memset 一次
22.    memset(dp, -1, sizeof(dp));
23.    int cnt = 0;
24.    //把 x 拆分成 cnt 个位
25.    while(x) {
26.        a[++cnt] = x%10;
27.        x /= 10;
28.    }
```

```
29.     return dfs(cnt, digit, 0, true, true);
30. }
31. int main() {
32.     ll x, y;
33.     scanf("%lld%lld", &x, &y);
34.     //把[x,y]转化为[0,y]-[0,x-1]
35.     for(int i = 0; i <= 9; i++)
36.         printf("%lld ", DP(y,i) - DP(x-1,i));
37.     return 0;
38. }
```

### 题目练习

DreamJudge 1616 windy 数

DreamJudge 1618 萌数

DreamJudge 1641 love 数

noobdream.com

## 4.7 FFT 快速傅里叶变换

我们直接跳过 FFT（快速傅里叶变换）的原理部分，直接讲如何使用，有兴趣的同学可以考试结束之后再去研究原理。

相信所有同学都做过高精度的题目，比如大整数乘法，在以前我们做大整数乘法的时候使用的是模拟的方法，复杂度是  $O(N^2)$ ，那么有没有更快的方法呢？

于是 FFT 就出来了，它可以在  $O(N\log N)$  的复杂度内解决大整数乘法的问题，这是 FFT 的一个非常经典的应用，大家一定要记住。

通过大整数乘法推出来快速傅里叶变换可以让多项式相乘变得更快，我们直接给一个模板。

```
1. // 求高精度乘法
2. #include <stdio.h>
3. #include <string.h>
4. #include <iostream>
5. #include <algorithm>
6. #include <math.h>
7. using namespace std;
8.
9. const double PI = acos(-1.0);
10. //复数结构体
11. struct complex
12. {
13.     double r,i;
14.     complex(double _r = 0.0,double _i = 0.0)
15.     {
16.         r = _r; i = _i;
17.     }
18.     complex operator +(const complex &b)
19.     {
20.         return complex(r+b.r,i+b.i);
21.     }
22.     complex operator -(const complex &b)
23.     {
24.         return complex(r-b.r,i-b.i);
25.     }
26.     complex operator *(const complex &b)
27.     {
```

```
28.     return complex(r*b.r-i*b.i,r*b.i+i*b.r);
29.     }
30. };
31. /*
32.  * 进行 FFT 和 IFFT 前的反转变换。
33.  * 位置 i 和 (i 二进制反转后位置) 互换
34.  * len 必须去 2 的幂
35.  */
36. void change(complex y[],int len)
37. {
38.     int i,j,k;
39.     for(i = 1, j = len/2;i < len-1; i++)
40.     {
41.         if(i < j)swap(y[i],y[j]);
42.         //交换互为小标反转的元素, i<j 保证交换一次
43.         //i 做正常的+1, j 左反转类型的+1,始终保持 i 和 j 是反转的
44.         k = len/2;
45.         while( j >= k)
46.         {
47.             j -= k;
48.             k /= 2;
49.         }
50.         if(j < k) j += k;
51.     }
52. }
53. /*
54.  * 做 FFT
55.  * len 必须为 2^k 形式,
56.  * on==1 时是 DFT, on==-1 时是 IDFT
57.  */
58. void fft(complex y[],int len,int on)
59. {
60.     change(y,len);
61.     for(int h = 2; h <= len; h <= 1)
62.     {
63.         complex wn(cos(-on*2*PI/h),sin(-on*2*PI/h));
64.         for(int j = 0;j < len;j+=h)
65.         {
66.             complex w(1,0);
67.             for(int k = j;k < j+h/2;k++)
68.             {
69.                 complex u = y[k];
70.                 complex t = w*y[k+h/2];
```

```
71.         y[k] = u+t;
72.         y[k+h/2] = u-t;
73.         w = w*wn;
74.     }
75. }
76. }
77. if(on == -1)
78.     for(int i = 0;i < len;i++)
79.         y[i].r /= len;
80. }
81. const int MAXN = 200010;
82. complex x1[MAXN],x2[MAXN];
83. char str1[MAXN/2],str2[MAXN/2];
84. int sum[MAXN];
85. int main()
86. {
87.     while(scanf("%s%s",str1,str2)==2)
88.     {
89.         int len1 = strlen(str1);
90.         int len2 = strlen(str2);
91.         int len = 1;
92.         while(len < len1*2 || len < len2*2)len<=1;
93.         for(int i = 0;i < len1;i++)
94.             x1[i] = complex(str1[len1-1-i]-'0',0);
95.         for(int i = len1;i < len;i++)
96.             x1[i] = complex(0,0);
97.         for(int i = 0;i < len2;i++)
98.             x2[i] = complex(str2[len2-1-i]-'0',0);
99.         for(int i = len2;i < len;i++)
100.            x2[i] = complex(0,0);
101.         //求 DFT
102.         fft(x1,len,1);
103.         fft(x2,len,1);
104.         for(int i = 0;i < len;i++)
105.             x1[i] = x1[i]*x2[i];
106.         fft(x1,len,-1);
107.         for(int i = 0;i < len;i++)
108.             sum[i] = (int)(x1[i].r+0.5);
109.         for(int i = 0;i < len;i++)
110.         {
111.             sum[i+1]+=sum[i]/10;
112.             sum[i]%=10;
113.         }
```

```

114.     len = len1+len2-1;
115.     while(sum[len] <= 0 && len > 0)len--;
116.     for(int i = len;i >= 0;i--)
117.         printf("%c",sum[i]+'0');
118.     printf("\n");
119. }
120. return 0;
121. }

```

FFT 还有一个常见的题型，如果同学们遇到原题可以直接用下面的模板。

问：给出  $n$  条线段长度，问任取 3 根，组成三角形的概率

其中  $n \leq 10^5$  用 FFT 求可以组成三角形的取法有几种

```

1.  const int MAXN = 400040;
2.  complex x1[MAXN];
3.  int a[MAXN/4];
4.  long long num[MAXN];//100000*100000 会超 int
5.  long long sum[MAXN];
6.
7.  int main()
8.  {
9.      int T;
10.     int n;
11.     scanf("%d",&T);
12.     while(T--)
13.     {
14.         scanf("%d",&n);
15.         memset(num,0,sizeof(num));
16.         for(int i = 0;i < n;i++)
17.         {
18.             scanf("%d",&a[i]);
19.             num[a[i]]++;
20.         }
21.         sort(a,a+n);
22.         int len1 = a[n-1]+1;
23.         int len = 1;
24.         while( len < 2*len1 )len <= 1;

```

```
25.     for(int i = 0;i < len1;i++)
26.         x1[i] = complex(num[i],0);
27.     for(int i = len1;i < len;i++)
28.         x1[i] = complex(0,0);
29.     fft(x1,len,1);
30.     for(int i = 0;i < len;i++)
31.         x1[i] = x1[i]*x1[i];
32.     fft(x1,len,-1);
33.     for(int i = 0;i < len;i++)
34.         num[i] = (long long)(x1[i].r+0.5);
35.     len = 2*a[n-1];
36.     //减掉取两个相同的组合
37.     for(int i = 0;i < n;i++)
38.         num[a[i]+a[i]]--;
39.     //选择的无序, 除以 2
40.     for(int i = 1;i <= len;i++)
41.     {
42.         num[i]/=2;
43.     }
44.     sum[0] = 0;
45.     for(int i = 1;i <= len;i++)
46.         sum[i] = sum[i-1]+num[i];
47.     long long cnt = 0;
48.     for(int i = 0;i < n;i++)
49.     {
50.         cnt += sum[len]-sum[a[i]];
51.         //减掉一个取大, 一个取小的
52.         cnt -= (long long)(n-1-i)*i;
53.         //减掉一个取本身, 另外一个取其它
54.         cnt -= (n-1);
55.         //减掉大于它的取两个的组合
56.         cnt -= (long long)(n-1-i)*(n-i-2)/2;
57.     }
58.     //总数
59.     long long tot = (long long)n*(n-1)*(n-2)/6;
60.     printf("%.7lf\n",(double)cnt/tot);
61. }
62. return 0;
63. }
```



## 4.8 机试押题

很多同学都不知道该怎么去机试押题，这里给同学们简单介绍一下 N 诺技术团队的押题经验。

首先押题一定要有一个参考，即往年的机试真题，那如果没有真题怎么办呢？根据直觉去判断，比如报考学校是第一年考机试，那么题目难度肯定是偏低的，只需要把高分篇的前几章内容熟练掌握即可拿到高分。往年机试真题有一个非常重要的作用就是确定机试难度标准。

有的学校机试难度很低，只会出一些简单的题，有的学校机试难度很高，几乎都是难度较高的算法题。所以你需要确定报考学校机试难度大致在哪个梯度才能去押题。

确定难度之后就要开始分析题型，有的老师喜欢考动态规划，有的老师喜欢考图论，有的老师喜欢考数据结构，这个就要根据往年的题目类型来确定。把常考的题目类型和难度都确定下来之后就是对照着去练习，比如 xx 大学比较喜欢考动态规划和搜索，那么你就要多去做一些动态规划和搜索的题目来提升自己，如果往年难度不大就做一些简单的题目，如果往年难度大就要对应练一些复杂的题目。

总而言之，言而总之，同学们最好还是跟着 N 诺的教程学习，除非你像 N 诺技术团队的大佬们一样做过几千道各类型的题目，否则自己押题的意义不大。

不过同学们不用担心，在每年的复试机试之前，N 诺技术团队的大佬都会在直播间公开帮助各个学校的同学们进行押题，详细的消息会在计算机考研机试攻略交流群（960036920）里通知大家。

## 完结撒花

当你看到这里, 说明你已经读完了本书所有的内容。恭喜你, 学完计算机考研机试攻略 - 满分篇的全部内容, 我们相信你一定会在机试中取得非常不错的成绩。

如果本书对你有所帮助, 希望你能在复试之后将还能记得的机试题目发表在 N 诺的交流区里, 当然也可以直接在 N 诺官方群或机试群里联系管理员。N 诺会根据你提供题目描述进行数据还原, 继续帮助下一届学弟学妹, 让他们可以做到最新的真题, 少走一些弯路, 并且我们会将你的名字或 N 诺 ID 放在题目的后面进行特别鸣谢。

最后, 我们不仅希望你能在机试中取得满分的成绩, 也希望你能如愿以偿的考上心目中理想的院校, **加油! Go!Go!Go!**

一定要做的说明: N 诺出版的考研系列书籍都将以**电子版**的形式进行发布更新, 需要纸质版的同学自行打印学习即可。

## N 诺考研系列书籍为什么只发布电子版不发布纸质版？

原因一：发布纸质版需要提前很久将书籍整理成册，时间太赶容易敷衍了事，我们相信慢工出细活。

原因二：纸质版一经印刷，便无法修改，就算发现问题或者想对某些内容进行优化也没办法。而电子版可以随时勘误进行修改，灵光一现的时候还能对前面写的不够好的地方进行优化。

原因三：电子版少了中间商，可以给同学们节约更多的费用。纸质版的话出版社、印刷商都要从中获取利润，最终羊毛出在羊身上。



## 如何获取 N 诺考研系列书籍？

noobdream.com

访问 N 诺平台（[www.noobdream.com](http://www.noobdream.com)）的兑换中心即可兑换或购买各种你想要的书籍或资料。

另外，**本书会不断的进行更新，所以需要最新版的同学，请去官网兑换中心进行兑换**，只要一次兑换，后续版本更新都可看到，不用重复兑换。

本书每隔一段时间都会进行一次版本迭代，如果书中有错别字或者对本书有其他建议可以向官方群管理员反馈，在下一次版本迭代中就会进行修正更新。

## N 诺考研系列图书

《计算机考研报考指南》

《C 语言考研攻略》

《数据结构考研攻略》

《操作系统考研攻略》

《计算机网络考研攻略》

《计算机组成原理考研攻略》

《数据库考研攻略》

《计算机考研机试攻略 - 高分篇》

《计算机考研机试攻略 - 满分篇》

考研路上，N 诺与你携手同行。

## N 诺 Offer 训练营

感谢同学们一直以来对 N 诺不遗余力的支持，N 诺能快速发展起来离不开每一个 N 诺 er 的帮助。

计算机专业是一个靠技术实力说话的专业，很多同学的编程功底和项目水平都不是很好，在找工作的时候很容易处处碰壁。

N 诺里有非常多的大佬，N 诺技术团队有来自腾讯、阿里、字节、百度等各个大厂的同学，我们希望能帮助更多的同学提升自己的编程水平，最终拿到一个满意的 Offer。

同学们可以把参加 N 诺 Offer 训练营视为一次对自己的投资，投资未来，请相信自己的潜力也请相信 N 诺的能力。多年以后，回想起来，相信参加 N 诺 Offer 训练营会是你人生中一次重要的转折点。

### N 诺 Offer 训练营的面向人群

#### 1、面向就业

大学不是人生的终点，很多同学大学毕业之后就会面临着找工作的问题，提升自己的技术能力可以找到一份更好的工作，不用担心毕业即失业的烦恼。

N 诺 Offer 训练营致力于给有梦想、肯拼搏、敢奋斗的同学提高最好的平台！

#### 2、面向硕士

研究生也不是人生的终点，很多同学读研之后虽然在学历上进行了一次升级，但是如果技术能力不行，依然很难找到一份满意的工作。

所以趁着读研留出来的缓冲时间，通过 Offer 训练营提升自己的能力，让自己的未来可以自由选择！

#### 3、课程形式

由于有很多同学不方便参与线下的培训，所以 Offer 训练营有线上班和线下班两种供同学

们灵活选择。

#### 4、报名要求

具备本科学历，愿意通过奋斗去实现人生的价值。

参与线下班的同学需要完成开课前作业，用作业考察态度，筛选出有决心、有毅力改变自己的同学。

#### 5、你将获得

编程能力的迅速提升，结合项目实战，逐步打下坚实的编程基础，培养积极、主动的学习能力。同学们在训练营里从小白逐步成长为大佬，训练营中不少本科同学就拿到了阿里、腾讯、头条、百度、美团等一线互联网大厂的 Offer，研究生同学更是手握多个 Offer 可以挑选。

#### N 诺 Offer 训练营的优势

Offer 训练营的技术学习氛围浓厚，同学们乐于分享与交流，能更加专注于提升自身能力。

N 诺的技术团队都是顶级的大佬，具有多年的教学经验，能帮助零基础的同学找到快速提高编程能力的学习方式。

#### N 诺 Offer 训练营的课程信息

Offer 训练营分为线上和线下两种模式，开设 4 种班型：

- 实习一对一（针对大一、大二、研一的同学）
- 校招一对一（针对大三、大四、研二、研三的同学）
- 跨专业一对一（针对非计算机专业的在校学生）
- 社招一对一（针对已毕业的同学）

要想了解 N 诺 Offer 训练营的方方面面，可以登录 N 诺官网（noobdream.com）查看。

最后，感谢缘分让我们在 N 诺相遇，愿每一个 N 诺 er，所有的坚持都不被辜负，所有的梦想都能绽放出耀眼的光芒~