

N 诺考研系列

# 计算机考研机试攻略

高分篇

N 诺

noobdream.com

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/av81203473>

## 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](http://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
<b>第一章 从零开始 .....</b>	<b>8</b>
1.1 机试分析 .....	8
1.2 IDE 的选择与评测结果 .....	10
1.3 DreamJudge 的使用 .....	11
1.4 输入输出技巧 .....	12
1.5 头文件技巧 .....	15
1.6 数组使用技巧 .....	16
1.7 审时度势 — 复杂度与是否可做 .....	19
1.8 C++ STL 的使用 .....	21
1.9 多组输入的问题 .....	27
<b>第二章 入门经典 .....</b>	<b>29</b>
2.1 简单模拟 .....	30
2.2 进制转换类问题 .....	32
2.3 排版类问题 .....	39
2.4 日期类问题 .....	44
2.5 字符串类问题 .....	47
2.6 排序类问题 .....	49
2.7 查找类问题 .....	58
2.8 贪心类问题 .....	65
2.9 链表类问题 .....	69
<b>第三章 数学 .....</b>	<b>73</b>
3.1 同模余定理 .....	74
3.2 最大公约数 (GCD) .....	77
3.3 最小公倍数 (LCM) .....	79
3.4 斐波那契数列 .....	80
3.5 素数判定 .....	81
3.6 素数筛选 .....	83
3.7 分解素因数 .....	86
3.8 二分快速幂 .....	88
3.9 常见数学公式总结 .....	90

3.10 规律神器 OEIS .....	92
<b>第四章 高精度问题 .....</b>	<b>94</b>
4.1 Python 解法 .....	95
4.2 Java 解法 .....	96
4.3 C/C++解法 .....	97
<b>第五章 数据结构 .....</b>	<b>98</b>
5.1 栈的应用 .....	99
5.2 哈夫曼树 .....	102
5.3 二叉树 .....	108
5.4 二叉排序树 .....	117
5.5 hash 算法 .....	120
5.6 前缀树 .....	121
<b>第六章 搜索 .....</b>	<b>127</b>
6.1 暴力枚举 .....	128
6.2 广度优先搜索 (BFS) .....	130
6.3 递归及其应用 .....	133
6.4 深度优先搜索 (DFS) .....	136
6.5 搜索剪枝技巧 .....	141
6.6 终极骗分技巧 .....	144
<b>第七章 图论 .....</b>	<b>145</b>
7.1 理论基础 .....	146
7.2 图的存储 .....	151
7.3 并查集 .....	154
7.4 最小生成树问题 .....	157
7.5 最短路径问题 .....	161
7.6 拓扑排序 .....	170
<b>第八章 动态规划 .....</b>	<b>173</b>
8.1 递推求解 .....	174
8.2 最大子段和 .....	176
8.3 最长上升子序列 (LIS) .....	180
8.4 最长公共子序列 (LCS) .....	184
8.5 背包类问题 .....	187
8.6 记忆化搜索 .....	190
8.7 字符串相关的动态规划 .....	193
<b>完结撒花 .....</b>	<b>197</b>
<b>N 诺考研系列图书 .....</b>	<b>199</b>
<b>N 诺 Offer 训练营 .....</b>	<b>200</b>

# 第一章 从零开始

## 1.1 机试分析

首先我们来看一下机试是怎样的一种考核模式

全国所有院校的机试都大同小异, 大部分有自己 OJ (Online Judge 也就是在线代码测评平台) 的学校都会采用 OJ 上做题的方式来进行考核。这种考核方式的好处是公开透明, 机器进行判题并给分, 就像 N 诺的 DreamJudge 一样。没有 OJ 的学校只能人工进行判题, 人工判题的话, 一方面是主观性比较强, 可能还会对其他方面进行考量, 这个就需要自己去了解了。总的来说, 不论是 OJ 判题还是人工判题, 代码都要能通过测试用例才能得到分数。

针对机试我们应该怎么去训练提升自己呢

首先, 一定要做题, 在 N 诺上做题, 不要自己埋头看书, 在不断做题的过程中才能提升自己。如果非要用一个量化的标准来衡量的话, 至少要在 N 诺上做够 100 题 (也就是达到砖石 II 以上段位), 才能保证你机试达到自己满意的成绩。题量是很关键的, 看懂的题再多, 都不如自己实际敲代码去解决问题更稳妥。

解题速度也是很重要的

其实解题速度和题量是正相关的, 相信题量足够的同学, 解题的速度都不会太慢。机试一般 2-3 个小时左右, 要解决 5-8 道题。平均下来一道题最多半个小时, 从读题、分析题意、思考解法、敲代码、调试、测试数据到最后提交这整个流程下来, 如果你平时训练的少, 读题慢、理解题意慢、思考解法慢、敲代码慢、调试慢, 这样一算下来, 一道简单的题都可能要一个小时才能写出来, 说不定题目还有坑点, 再慢慢调试, 基本上就凉了。

## 多打比赛也是很重要的

很多同学平时做的题很多, 解题速度也挺快, 但是一到比赛或者考试的时候就会卡题, 在压力的情况下发挥失常比比皆是, 所以平时就要锻炼自己的抗压能力。

N 诺上除了每个周定期举办的小白赛, 还特别为大家准备了**考研机试冲刺八套卷**。

在本书的最后, 你可以看到关于考研机试冲刺八套卷的详细信息。通过这八套卷的练习, 相信会让你的水平产生一个脱胎换骨的变化。

## 准备好模板是至关重要的

一般来说, 机试都可以带书和纸质资料进入考场。所以提前把那些函数的用法和算法的模板准备好是很重要的, 一方面是增加自己的信心, 万一没记住还可以翻开来看一下。另外说不定考到原题或者类似的题, 就可以直接秒杀了。

**特别提醒:** 本书默认读者是会 C 语言的基本语法的, 比如 if 语句、for 语句等等。

noobdream.com

C 语言基本语法还未掌握的同学建议学习 N 诺的 C 语言快速入门课程(3 天学会 C 语言不是梦)

地址: <http://www.noobdream.com/Major/majorinfo/1/>

## 1.2 IDE 的选择与评测结果

建议选择 **CodeBlocks** 作为平时敲代码练习的 IDE

下载地址: <http://www.noobdream.com/Major/article/1/>

### 常见做题结果反馈

**Accepted:** 答案正确, 恭喜你正确通过了这道题目。

**Wrong Answer:** 答案错误, 出现这个错误的原因一般是你的程序实现或思路出现了问题, 或者数据范围边界没有考虑到。

**Runtime Error:** 运行时错误, 出现这个错误的原因一般是数组越界或者递归过深导致栈溢出。

**Presentation Error:** 输出格式错误, 出现这个错误的原因一般是末尾多了或少了空格, 多了或少了换行

**Time Limit Exceeded:** 程序运行超时, 出现这个错误的原因一般是你的算法不够优秀, 导致程序运行时间过长。

**Memory Limit Exceeded:** 运行内存超限, 出现这个错误的原因一般是你的程序申请太大了空间, 超过了题目规定的空间大小。

**Compile Error:** 编译错误, 这个不用说了吧, 就是你的代码存在语法错误, 检查一下是不是选择错误的语言提交了。

**Output Limit Exceeded:** 输出超限, 程序输出过多的内容, 一般是循环出了问题导致多次输出或者是调试信息忘记删除了。

**Submitting:** 提交中, 请等待题目结果的返回, 由于判题机有性能差异, 所以返回结果的速度也不一样, N 诺上做题一般瞬间就能出结果。

以上几种结果就是评判系统可能会返回的几种常见结果。若返回 **Accept**, 那么你就可以拿到该题所有分数, 如果返回其他结果, 则要看你报考学校的考试规则, 是根据通过测试点的百分比给分还是只要不是 **AC** 就得 0 分。

## 1.3 DreamJudge 的使用

DreamJudge 是一个在线代码测评的平台, 可以很方便的检验自身的学习情况。使用 DreamJudge 的方法很简单, 通过百度搜索 N 诺或者在浏览器中输入网址 [www.noobdream.com](http://www.noobdream.com) 进入 N 诺然后点击网站导航上方的 DreamJudge 就可以进去啦。

做题页面如下:

The screenshot shows the DreamJudge interface for a challenge titled "A+B问题". The left sidebar contains the N 诺 logo and navigation links: DreamJudge, 比赛, 题库 (selected), 提交状态, 历年真题, 和 Rating排行榜。在 "题库" 选项卡下，显示了 "A+B问题" 的题目描述、输入输出格式、样例输入输出，以及提交按钮。

**题目描述**

Time Limit: 1000 ms  
Memory Limit: 256 kb

输入A,B  
输出A+B  
-1,000,000,000 <= A,B <= 1,000,000,000

**输入输出格式**

输入描述: 输入包含两个整数a, b, 用一个空格分隔。  
输出描述: 输出一个整数, 表示a+b的值。

**输入输出样例**

输入样例#: 5 8      复制      输出样例#: 13      复制

右侧代码编辑器显示 Python 代码:

```
1 while True:
2     try:
3         a, b = map(int, input().split())
4         c = a+b
5         print(c)
6     except:
7         break
```

底部有 "提交代码" 按钮。

首先要登录我们的 N 诺账号, 然后开始做题。如果没有账号, 右上角点击注册, 然后注册一个账号就可以了。

然后将代码粘贴到右边的输入框里, 在上面选择使用哪种语言提交, C/C++/Java/Python, 建议选择 C++ 提交, 因为 C++ 可以编译 C 语言代码。我们一般写代码为了方便, 都会使用一点 C++ 的特性来帮助我们快速解决一道题目。如果代码里含有 C++ 的特性却选择了 C 语言提交的话, 会返回编译错误的提示信息。

## 1.4 输入输出技巧

输入 int 型变量 `scanf("%d", &x);`

输入 double 型变量 `scanf("%lf", &x);` 不用 float 直接 double

输入 char 类型变量 `scanf("%c", &x);`

输入字符串数组 `scanf("%s", s);`

输出与输入表示方式一致

`printf("%s\n", s);`

### scanf 输入解析

输入日期 2019-10-21

```
1. int year, month, day;
2. scanf("%d-%d-%d", &year, &month, &day);
3. printf("%d %d %d\n", year, month, day);
```

这样可以直接解析出来

输入时间 18:21:30

```
1. int hour, minute, second;
2. scanf("%d:%d:%d", &hour, &minute, &second);
3. printf("%d %d %d\n", hour, minute, second);
```

### scanf 和 gets

输入一行字符串带空格的话, 使用 gets, scanf 遇到空格会自动结束

```
1. char s[105];
2. gets(s); //例如输入 how are you?
3. printf("%s\n", s);
```

### getchar 和 putchar

读入单个字符和输出单个字符, 一般在 scanf 和 gets 中间使用 getchar 用于消除回车'\n'的影响

## 输出进制转换

```
1. int a = 10;
2. printf("%x\n", a); // 小写十六进制输出 答案 a
3. printf("%X\n", a); // 大写十六进制输出 答案 A
4. printf("%o\n", a); // 八进制输出 答案 12
```

## 输出增加前置 0

```
1. int a = 5;
2. printf("%02d\n", a); // 其中 2 代表宽度 不足的地方用 0 补充
3. 输出结果 05
4. printf("%04d\n", a);
5. 输出结果 0005
```

## 输出保留小数

```
1. double a = 3.6;
2. printf("%.2f\n", a); // 2 表示保留两位小数
```

输出结果 3.60

有小数输出小数, 没小数输出整数

%g

**特别注意:** 中文符号和英文符号要对应一致, 一般情况下都用英文符号 (如中文逗号, 和英文逗号, )

## long long 的使用

很多情况下的计算会超出 int, 比如求  $N!$ ,  $N$  较大的时候 int 就存不下了, 这时候我们就要用 long long。那么我们怎么去记 int 和 long long 的范围呢, 有一个简单的记法, int 范围-1e9 到 1e9, long long 范围-1e18 到 1e18, 这样就容易记了。

```
1. long long x;
2. scanf("%lld", &x);
3. printf("%lld\n", x);
```

## 字符的 ASCII 码

不要硬记, 直接输出来看

```
printf("%d\n", 'a');
```

输出结果 97

```
printf("%d\n", 'A');
```

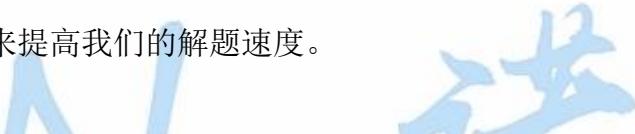
输出结果 65

**特别注意:** 如果遇到需要 ASCII 码的题目时候记住 `char` 字符和 `int` 值是可以相互转化的。

## cin 和 cout

很多时候使用 C++ 的输入输出写起来更简单, 在应对一些**输入输出量不是很大的**题目的时候, 我们会采用 `cin` 和 `cout` 来提高我们的解题速度。

比如求两个数的和:



```
1. #include <iostream> //输入输出函数的头文件
2.
3. int main() {
4.     int a, b;
5.     cin >> a >> b;
6.     cout << a + b; //输出两个数之和
7. }
```

可以发现, C++ 的输入输出敲起来更快, 这是我们会使用它来进行混合编程的原因之一。

另外, C++ 的 `string` 类对于字符串操作很方便, 但是输入输出只能用 `cin`、`cout`。

**特别注意:** 大家一定平时练习的时候不要排斥混合编程, 即 C 与 C++ 语法混用, 然后用 C++ 提交。这样可以极大的帮助你以更快的速度解决一道你用纯 C 写半天才能解决的题目, 留下充裕的时间去解决更多的题目。

**友情提示:** 当输入或输出格式有特殊要求的时候, `cin` 和 `cout` 不方便解决, 那么我们还是使用 `scanf` 和 `printf` 来解决问题。要注意的是 **printf 尽量不要和 cout 同时使用**, 会发生一些不可控的意外。

## 1.5 头文件技巧

这里推荐一个万能头文件给大家

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

不过要看考试的评测机支不支持, 绝大部分都是支持的。当然, 我们还可以留一手, 准备一个完整的头文件, 在考试开始前敲上去就行。

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <math.h>
4. #include <stdlib.h>
5. #include <time.h>
6. #include <algorithm>
7. #include <iostream>
8. #include <queue>
9. #include <stack>
10. #include <vector>
11. #include <string>
12. using namespace std;
13.
14. int main() {
15.
16.     return 0;
17. }
```

**特别注意:** 头文件可以多, 但是不能少, 但是有一些头文件是不允许的, 大部分 OJ 为了系统安全性考虑限制了一些特殊的 API, 导致一些头文件不能使用, 比如 windows.h。当然不同的 OJ 的安全策略也不尽相同, 一般不涉及到系统函数的头文件一般都是可以使用的。

## 1.6 数组使用技巧

数组除了可以存储数据以外, 还可以用来进行标记。

例题:

输入 N (N<=100) 个数, 每个数的范围> 0 并且 <= 100, 请将每个不同的数从小到大输出并且输出它对应的个数。

样例输入

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

样例输出

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

代码如下

noobdream.com

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int f[105]={0}; //注意, 尽量将数组开在全局
5. int main() {
6.     int n,x;
7.     scanf("%d", &n);
8.     for (int i = 0; i < n; i++) {
9.         scanf("%d", &x);
10.        f[x]++;
11.    }
12.    for (int i = 0; i <= 100; i++) {
13.        if (f[i] > 0) printf("%d %d\n", i, f[i]);
14.    }
15.    return 0;
16. }
```

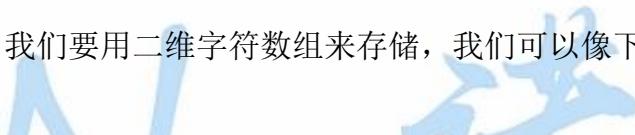
在这个程序中, 我们使用一个数组  $f$  记录每个值得个数,  $f[i]$  的值表示  $i$  这个数有多少个, 初始的时候每个值的个数都是 0。

数组的使用不一定从 0 开始, 可以从任意下标开始, 只要我们使用的时候对应上就行。

例如我们存储地图的时候

```
#####
#.##
##@#
####
```

假设一个地图是这样的, 我们要用二维字符数组来存储, 我们可以像下面这样做。



```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. char mpt[10][10];
5. int main() {
6.     for (int i = 1; i <= 4; i++) {
7.         scanf("%s", mpt[i] + 1);
8.         /* 不要用下面这种输入方式, 否则会出问题, 因为回车也算一个 char 字符
9.         for (int j = 1; j <= 4; j++) {
10.             scanf("%c", &mpt[i][j]);
11.         }
12.     */
13.     }
14.     for (int i = 1; i <= 4; i++) {
15.         for (int j = 1; j <= 4; j++) {
16.             printf("%c", mpt[i][j]);
17.         }
18.         printf("\n");
19.     }
20.     return 0;
21. }
```

## 数组还可以嵌套使用

我们将上面那题改进一下

### 例题:

输入 N (N<=100) 个数, 每个数的范围> 0 并且 <= 100, 请将每个不同的数输出并且输出它对应的个数。要求按值出现的次数从小到大排序, 如果多个值有相同的个数, 只用输出值最大的那个。

### 样例输入

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

### 样例输出

```
4 1
5 2
```

### 代码如下

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int f[105] = {0};
5. int p[105] = {0}; //p[i]表示有 i 个这样的数的最大值是多少
6. int main() {
7.     int n, x;
8.     scanf("%d", &n);
9.     for (int i = 0; i < n; i++) {
10.         scanf("%d", &x);
11.         f[x]++;
12.     }
13.     for (int i = 0; i <= 100; i++) p[f[i]] = i;
14.     for (int i = 1; i <= 100; i++) {
15.         if (p[i] > 0) printf("%d %d\n", p[i], i);
16.     }
17.     return 0;
18. }
```

## 1.7 审时度势 — 复杂度与是否可做

在做题之前, 我们要先判断这道题是否可做, 对于简单的模拟题, 大家肯定都知道, 我能写出来就是可做, 写不出来就是不可做。但是对于循环嵌套和算法题, 我们就需要去判断思考自己设计的算法是否可以通过这道题。

不懂复杂度计算的同学去看一下数据结构课程的第一章, 很简单的。

例如: 我们写一个冒泡排序, 它是两个 for 循环, 时间复杂度是  $O(N^2)$ , 那么在 1S 内我们最多可以对多少个数进行冒泡排序呢,  $N$  在 1000 – 3000 之间。一般情况下我们可以默认评测机一秒内可以运行  $1e7$  条语句, 当然这只是一个大概的估计, 实际上每个服务器的性能不同, 这个值都不同, 但是一般都相差不大, 差一个常数是正常的。因此, 我们可以这样做一个对应, 下面是时限 1S 的情况

$O(N)$	$N$ 最大在 500W 左右
$O(N \log N)$	$N$ 最大在 20W 左右
$O(N^2)$	$N$ 最大在 2000 左右
$O(N^2 \log N)$	$N$ 最大 700 左右
$O(N^3)$	$N$ 最大在 200 左右
$O(N^4)$	$N$ 最大在 50 左右
$O(2^N)$	$N$ 最大在 24 左右
$O(N!)$	$N$ 最大在 10 左右

如果是 2S、3S 对应的乘以 2 和 3 就可以。

**特殊技巧:** 如果发现自己设计的算法不能在题目要求的时限内解决问题, 不要着急, 可以先把这道题留一下, 继续做其他题, 然后看一下排行榜, 有多少人过了这道题, 如果过的人多, 那么说明这道题可能数据比较水, 直接暴力做, 不要怕复杂度的问题, 因为出题人可能偷懒或者失误了导致数据很水。**考研机试的题目数据大部分情况都比较水, 所以不要被复杂度吓唬住了, 后面的章节会教大家面对不会更好的算法那来解决题目时候, 如何用优雅的技巧水过去。**

举个简单的例子

题目要求你对 10W 个数进行排序

假设你只会冒泡排序, 但是冒泡排序很明显复杂度太高了, 但是有可能出题人偷懒, 他构造的测试数据最多只有 100 个, 根本没有 10W 个, 那么你就可以用冒泡排序通过这道题。

但是这种情况比较少见, 一般至少都会有一组极限数据, 所以可以先把这道题放着去做其他题, 然后再看看其他人能不能通过, 如果很多人都过了, 那么你就可以暴力试一下。

**特别注意:** 空间复杂度一般不会限制, 如果遇到了再想办法优化空间。



## 1.8 C++ STL 的使用

C++的算法头文件里有很多很实用的函数，我们可以直接拿来用。

```
#include <algorithm>
```

### 排序

sort() 函数：依次传入三个参数，要排序区间的起点，要排序区间的终点+1，比较函数。比较函数可以不填，则默认为从小到大排序。

### 使用示例

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int a[105];
5. int main() {
6.     int n;
7.     scanf("%d", &n);
8.     for (int i = 0; i < n; i++) {
9.         scanf("%d", &a[i]);
10.    }
11.    sort(a, a+n);
12.    for (int i = 0; i < n; i++)
13.        printf("%d ", a[i]);
14.    return 0;
15. }
```

### 查找

lower\_bound() 函数

upper\_bound() 函数

lower\_bound() 和 upper\_bound() 都是利用二分查找的方法在一个排好序的数组中进行查找的。

在从小到大的排序数组中,

`lower_bound( begin, end, num)`: 从数组的 `begin` 位置到 `end-1` 位置二分查找第一个大于或等于 `num` 的数字, 找到返回该数字的地址, 不存在则返回 `end`。通过返回的地址减去起始地址 `begin`, 得到找到数字在数组中的下标。

`upper_bound( begin, end, num)`: 从数组的 `begin` 位置到 `end-1` 位置二分查找第一个大于 `num` 的数字, 找到返回该数字的地址, 不存在则返回 `end`。通过返回的地址减去起始地址 `begin`, 得到找到数字在数组中的下标。

在从大到小的排序数组中, 重载 `lower_bound()` 和 `upper_bound()`

`lower_bound( begin, end, num, greater<type>()`: 从数组的 `begin` 位置到 `end-1` 位置二分查找第一个小于或等于 `num` 的数字, 找到返回该数字的地址, 不存在则返回 `end`。通过返回的地址减去起始地址 `begin`, 得到找到数字在数组中的下标。

`upper_bound( begin, end, num, greater<type>()`: 从数组的 `begin` 位置到 `end-1` 位置二分查找第一个小于 `num` 的数字, 找到返回该数字的地址, 不存在则返回 `end`。通过返回的地址减去起始地址 `begin`, 得到找到数字在数组中的下标。

## 使用示例

```
1. #include<bits/stdc++.h>
2. using namespace std;
3.
4. int cmp(int a,int b){
5.     return a>b;
6. }
7. int main(){
8.     int num[6]={1,2,4,7,15,34};
9.     sort(num,num+6); //按从小到大排序
10.    int pos1=lower_bound(num,num+6,7)-num;
11.    //返回数组中第一个大于或等于被查数的值
12.    int pos2=upper_bound(num,num+6,7)-num;
13.    //返回数组中第一个大于被查数的值
14.    cout<<pos1<<" "<<num[pos1]<<endl;
15.    cout<<pos2<<" "<<num[pos2]<<endl;
16.    sort(num,num+6,cmp); //按从大到小排序
17.    int pos3=lower_bound(num,num+6,7,greater<int>())-num;
```

```
18.    //返回数组中第一个小于或等于被查数的值
19.    int pos4=upper_bound(num,num+6,7,greater<int>())-num;
20.    //返回数组中第一个小于被查数的值
21.    cout<<pos3<<" "<<num[pos3]<<endl;
22.    cout<<pos4<<" "<<num[pos4]<<endl;
23.    return 0;
24. }
```

## 优先队列

通过 `priority_queue<int> q` 来定义一个储存整数的空的 `priority_queue`。当然 `priority_queue` 可以存任何类型的数据, 比如 `priority_queue<string> q` 等等。

### 示例代码

```
1. #include <iostream>
2. #include <queue>
3. using namespace std;
4. int main() {
5.     priority_queue<int> q;//定义一个优先队列
6.     q.push(1);//入队
7.     q.push(2);
8.     q.push(3);
9.     while (!q.empty()) {//判断队列不为空
10.         cout << q.top() << endl;//队首元素
11.         q.pop();//出队
12.     }
13.     return 0;
14. }
```

C++的 STL (标准模板库) 是一个非常重要的东西, 可以极大的帮助你更快速的解决题目。

## vector

通过 `vector<int> v` 来定义一个储存整数的空的 `vector`。当然 `vector` 可以存任何类型的数据, 比如 `vector<string> v` 等等。

## 示例代码

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4. int main() {
5.     vector<int> v; // 定义一个空的 vector
6.     for (int i = 1; i <= 10; ++i) {
7.         v.push_back(i * i); // 加入到 vector 中
8.     }
9.     for (int i = 0; i < v.size(); ++i) {
10.         cout << v[i] << " "; // 访问 vector 的元素
11.     }
12.     cout << endl;
13.     return 0;
14. }
```

queue



通过 `queue<int> q` 来定义一个储存整数的空的 queue。当然 queue 可以存任何类型的数据，比如 `queue<string> q` 等等。

## 示例代码

noobdream.com

```
1. #include <iostream>
2. #include <queue>
3. using namespace std;
4. int main() {
5.     queue<int> q; // 定义一个队列
6.     q.push(1); // 入队
7.     q.push(2);
8.     q.push(3);
9.     while (!q.empty()) { // 当队列不为空
10.         cout << q.front() << endl; // 取出队首元素
11.         q.pop(); // 出队
12.     }
13.     return 0;
14. }
```

## stack

通过 `stack<int> S` 来定义一个全局栈来储存整数的空的 `stack`。当然 `stack` 可以存任何类型的数据, 比如 `stack<string> S` 等等。

### 示例代码

```
1. #include <iostream>
2. #include <stack>
3. using namespace std;
4. stack<int> S;//定义一个栈
5. int main() {
6.     S.push(1);//入栈
7.     S.push(10);
8.     S.push(7);
9.     while (!S.empty()) {//当栈不为空
10.         cout << S.top() << endl;//输出栈顶元素
11.         S.pop();//出栈
12.     }
13.     return 0;
14. }
```

## map

通过 `map<string, int> dict` 来定义一个 `key:value` 映射关系的空的 `map`。当然 `map` 可以存任何类型的数据, 比如 `map<int, int> m` 等等。

### 示例代码

```
1. #include <iostream>
2. #include <string>
3. #include <map>
4. using namespace std;
5. int main() {
6.     map<string, int> dict;//定义一个 map
7.     dict["Tom"] = 1;//定义映射关系
8.     dict["Jone"] = 2;
9.     dict["Mary"] = 1;
10.    if (dict.count("Mary")) {//查找 map
11.        cout << "Mary is in class " << dict["Mary"];
12.    }
13. }
```

```
13. //使用迭代器遍历 map 的 key 和 value
14. for (map<string, int>::iterator it = dict.begin(); it != dict.end(); ++it) {
15.     cout << it->first << " is in class " << it->second << endl;
16. }
17. dict.clear(); //清空 map
18. return 0;
19. }
```

set

通过 `set<string> country` 来定义一个储存字符串的空的 set。当然 set 可以存任何类型的数据, 比如 `set<int> s` 等等。

## 示例代码

```
1. #include <iostream>
2. #include <set>
3. using namespace std;
4. int main() {
5.     set<string> country; // 定义一个存放 string 的集合
6.     country.insert("China"); // 插入操作
7.     country.insert("America");
8.     country.insert("France");
9.     set<string>::iterator it;
10.    // 使用迭代器遍历集合元素
11.    for (it = country.begin(); it != country.end(); ++it) {
12.        cout << *it << " ";
13.    }
14.    cout << endl;
15.    country.erase("American"); // 删除集合内的元素
16.    country.erase("England");
17.    if (country.count("China")) { // 统计元素个数
18.        cout << "China in country." << endl;
19.    }
20.    country.clear(); // 清空集合
21.    return 0;
22. }
```

## 1.9 多组输入的问题

对有的题目来说, 可能需要多组输入。

多组输入是什么意思呢? 一般的题目我们输入一组数据, 然后直接输出程序就结束了, 但是多组输入的话要求我们可以循环输入输出结果。

例题:

输入两个数, 输出两个数的和, 要求多组输入。

样例输入

```
1 2  
3 7  
10 24
```

样例输出

```
3  
10  
34
```

C 循环读入代码如下

```
1. #include <bits/stdc++.h>  
2. using namespace std;  
3.  
4. int main() {  
5.     int a, b;  
6.     while (scanf("%d%d", &a, &b) != EOF) {  
7.         printf("%d\n", a+b);  
8.     }  
9.     return 0;  
10. }
```

**特别注意:** 不能使用 `while(1)` 这样死循环, `!=EOF` 的意思一直读取到文件末尾 (End of file) 另外, 多组输入一定要注意初始化问题, 数组和变量的初始化要放在 `while` 循环内, 否则上一次的运算的结果会影响当前的结果。

C++循环读入代码如下

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int main() {
5.     int a, b;
6.     while (cin >> a >> b) {
7.         cout << a + b << endl;
8.     }
9.     return 0;
10. }
```

Java 循环读入代码如下

```
1. Scanner stdin = new Scanner(System.in);
2. while (stdin.hasNext()) {
3.     String s = stdin.next();
4.     int n = stdin.nextInt();
5.     double b = stdin.nextDouble();
6. }
```

Python 循环读入代码如下

```
1. while True:
2.     try:
3.         a, b = map(int, input().split())
4.         c = a+b
5.         print(c)
6.     except: #读到文件末尾抛出异常结束循环
7.         break
```

## 第二章 入门经典

我们根据全国上百所院校的历年真题进行分析, 总结了其中常见的题型, 最常考的知识点。

学会这一章, 在机试难度较低的学校基本上可以拿到 60-80 分左右的成绩, 在机试难度中等的学校, 也可拿到 40-60 分左右的成绩, 在机试难度高的学校亦可将签到题做出来, 拿到 20-40 分的成绩。

所以, 认真看完这一章的内容对你的帮助会很大, 加油, fighting!



本书配套视频精讲: <https://www.bilibili.com/video/av81203473>

## 2.1 简单模拟

在考研机试中, 有一类很常见的题型叫做简单模拟。顾名思义, 就是不需要去考虑什么算法, 直接按照题目的意思进行模拟计算就行。

### 促销计算

#### 题目描述:

某百货公司为了促销, 采用购物打折的优惠方法, 每位顾客一次购物: 在 1000 元以上者, 按 9.5 折优惠; 在 2000 以上者, 按 9 折优惠; 在 3000 以上者, 按 8.5 折优惠; 在 5000 以上者, 按 8 折优惠; 编写程序, 购物款数, 计算并输出优惠价。

#### 输入样例#:

```
850
1230
5000
3560
```

#### 输出样例#:

```
discount=1, pay=850
discount=0.95, pay=1168.5
discount=0.8, pay=4000
discount=0.85, pay=3026
```

#### 题目来源:

```
DreamJudge 1091
```

**解题分析:** 根据题目的意思, 我们知道就是按照题意去进行打折优惠的计算, 只需要判断输入的数值在哪个区间该用什么优惠去计算就好了。

#### 参考代码

```
1. #include <bits/stdc++.h> // 万能头文件
2. using namespace std;
3.
```

```
4. int main() {  
5.     double a;  
6.     scanf("%lf", &a);  
7.     //使用%g 可以自动去掉小数点后多余的 0 如果是整数则显示整数  
8.     if (a < 1000) printf("discount=1, pay=%g\n", a);  
9.     if (a >= 1000 && a < 2000) printf("discount=0.95, pay=%g\n", a*0.95);  
10.    if (a >= 2000 && a < 3000) printf("discount=0.9, pay=%g\n", a*0.9);  
11.    if (a >= 3000 && a < 5000) printf("discount=0.85, pay=%g\n", a*0.85);  
12.    if (a >= 5000) printf("discount=0.8, pay=%g\n", a*0.8);  
13.    return 0;  
14. }
```

## 题型总结

简单模拟这类题目在考试中很常见, 属于送分签到的题目。所有的考生, 注意了, 这类题必须会做。

对于简单模拟这一类的题目, 怎么去练习提高呢?

很简单, 在 DreamJudge 上**多做题**就行了。那么要达到什么样的标准呢?

如果你想拿高分甚至满分, 平时训练的时候, 这类题尽量要在 8 分钟内解决。

如果你只是想拿个还不错的成绩, 这类题 AC 的时间尽量不要超过 15 分钟, 一定要记住, 最坏情况不能超过 20 分钟, 如果超过了, 说明你平时做的题还是太少了。

在考试的过程中, 大多数考生都会紧张, 有些考生甚至会手抖, 导致敲多某个字母, 然后又调试半天, 找半天错, 会导致比平时解决同样难度的问题时长多一倍甚至更多, 所以平时就要注意, 做题千万不能太慢了, 不然没有足够的时间来解决其他的题目哦。

## 练习题目

DreamJudge 1133 求 1 到 n 的和

DreamJudge 1043 计算  $S_n$

DreamJudge 1040 利润提成

DreamJudge 1722 身份证校验

## 2.2 进制转换类问题

进制转换类的题目在绝大多数学校都是必考题目之一, 这类题目的既基础又灵活, 能看出学生的编程功底, 所以这类题目一定要掌握。

总的来说, 跟进制相关的题目可以分为以下几种题型

1、**反序数**: 输入一个整数如 123, 将其转换为反序之后的整数 321

2、**10 进制转 2 进制**: 将一个 10 进制整数转化为一个 2 进制的整数

例如: 7 转换为 111

3、**10 进制转 16 进制**: 将一个 10 进制整数转化为一个 16 进制的整数

例如: 10 转换为 A

4、**10 进制转 x 进制**: 将一个 10 进制整数转化为一个 x 进制的整数

解析: 这是前面两个的一种通解, 如果会前面两种那么这个自然也触类旁通。

5、**x 进制转 10 进制**: 将一个 x 进制整数转化为一个 10 进制的整数

解析: 这是上一种情况的反例, 看代码之后相信也能容易理解。

6、**x 进制转 y 进制**: 将一个 x 进制整数转化为一个 y 进制的整数

解析: 遇到这种情况, 可以拆解为 x 先转为 10 进制, 然后再将 10 进制转为 y 进制。

### 7、字符串转浮点数

例如: 有一串字符串 31.25 将其转换为一个浮点数, 可以先转整数部分, 再转小数部分, 最后相加即可。

### 8、浮点数转字符串

例如: 有一个浮点数 23.45 将其转换为一个字符串进行存储, 可以将整数和小数拆开再合并成一个字符串。

### 9、字符串转整型和整形转字符串

解析: 直接用 atoi 函数和 itoa 函数即可。

## 反序数代码

```
1. #include <stdio.h>
2.
3. int main() {
4.     int n;
5.     scanf("%d", &n);
6.     int ans = 0;//将反序之后的答案存在这里
7.     while (n > 0) {//将 n 逐位分解
8.         ans *= 10;
9.         ans += (n % 10);
10.        n /= 10;
11.    }
12.    printf("%d\n", ans);
13.    return 0;
14. }
```

## 10 进制转 x 进制代码 (x 小于 10 的情况)

```
1. #include <stdio.h>
2.
3. int main() {
4.     int n, x;
5.     int s[105];
6.     //输入 10 进制 n 和 要转换的进制 x
7.     scanf("%d%d", &n, &x);
8.     int cnt = 0;//数组下标
9.     while (n > 0) {//将 n 逐位分解
10.         int w = (n % x);
11.         s[cnt++] = w;
12.         n /= x;
13.     }
14.     //反序输出
15.     for (int i = cnt - 1; i >= 0; i--) {
16.         printf("%d", s[i]);
17.     }
18.     printf("\n");
19.     return 0;
20. }
```

## 10 进制转 x 进制代码 (通用版)

```
1. #include <stdio.h>
2.
3. int main() {
4.     int n, x;
5.     char s[105];//十进制以上有字符, 所以用 char 存储
6.     //输入 10 进制 n 和 要转换的进制 x
7.     scanf("%d%d", &n, &x);
8.     int cnt = 0;//数组下标
9.     while (n > 0) {//将 n 逐位分解
10.         int w = (n % x);
11.         if (w < 10) s[cnt++] = w + '0';//变成字符需要加'0'
12.         else s[cnt++] = (w - 10) + 'A';//如果转换为小写则加'a'
13.         //如果大于 10 则从 A 字符开始
14.         n /= x;
15.     }
16.     //反序输出
17.     for (int i = cnt - 1; i >= 0; i--) {
18.         printf("%c", s[i]);
19.     }
20.     printf("\n");
21.     return 0;
22. }
```

## x 进制转 10 进制 (x 为 2 时)

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char s[105];
6.     //输入二进制字符串
7.     scanf("%s", &s);
8.     int ans = 0;//
9.     int len = strlen(s);
10.    for (int i = 0; i < len; i++) {
11.        if (s[i] == '0') {
12.            ans = ans * 2;
13.        }
14.        else {
15.            ans = ans * 2 + 1;
```

```
16.     }
17. }
18. printf("%d\n", ans);
19. return 0;
20. }
```

## x 进制转 10 进制 (通用版)

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char s[105];
6.     int x;
7.     //输入 x 进制字符串 和 代表的进制 x
8.     scanf("%s%d", &s, &x);
9.     int ans = 0;//
10.    int len = strlen(s);
11.    for (int i = 0; i < len; i++) {
12.        ans = ans * x;
13.        if (s[i] >= '0' && s[i] <= '9') ans += (s[i] - '0');
14.        else ans += (s[i] - 'A') + 10;
15.    }
16.    printf("%d\n", ans);
17.    return 0;
18. }
```

## x 进制转 y 进制 (通用版)

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char s[105];
6.     int x, y;
7.     //输入二进制字符串 和 代表的进制 x 以及要转换的进制 y
8.     scanf("%s%d%d", &s, &x, &y);
9.     int ans = 0;
10.    int len = strlen(s);
11.    for (int i = 0; i < len; i++) {
```

```
12.     ans = ans * x;
13.     if (s[i] >= '0' && s[i] <= '9') ans += (s[i] - '0');
14.     else ans += (s[i] - 'A') + 10;
15. }
16. char out[105];
17. int cnt = 0;
18. while (ans > 0) {
19.     int w = (ans % y);
20.     if (w < 10) out[cnt++] = w + '0';
21.     else out[cnt++] = (w-10) + 'A';
22.     ans /= y;
23. }
24. for (int i = cnt - 1; i >= 0; i--) {
25.     printf("%c", out[i]);
26. }
27. printf("\n");
28. return 0;
29. }
```

还有一类进制转换的题目是大数的进制转换, 建议同学们学完第四章高精度问题, 学会大数的加减乘除法再看下面这类题。

## 进制转换

### 题目描述:

将一个长度最多为 30 位数字的十进制非负整数转换为二进制数输出

### 输入描述:

多组数据, 每行为一个长度不超过 30 位的十进制非负整数。

(注意是 10 进制数字的个数可能有 30 个, 而非 30bits 的整数)

### 输出描述:

每行输出对应的二进制数。

### 输入样例#:

```
0
1
3
8
```

输出样例#:

```
0
1
11
1000
```

题目来源:

DreamJudge 1178

题目解析: 这个题和一般的 10 进制转二进制的区别在于它的数是一个很大的整数。对于一个很大的数我们做法和普通的 10 进制转二进制是一样的, 就是不断的%2 然后除/2, 唯一区别在于要用大数进行模拟。

参考代码



```
1. #include<bits/stdc++.h>
2. using namespace std;
3.
4. //十进制转二进制
5. char s[40], buf[200];
6. int main(){
7.     int num[40];
8.     while (scanf("%s", s) != EOF){
9.         int len = strlen(s);
10.        for (int i = 0; i < len; i++){//字符串转成 int 数组
11.            num[i] = s[i] - '0';
12.        }
13.        int i = 0, len_str = 0;
14.        while (i < len){//除 2 取余法
15.            buf[len_str++] = num[len - 1] % 2 + '0';//余数
16.            // 大数除法, 更新 num[] 数组
17.            int c = 0;
18.            for (int j = i; j < len; j++){
19.                int tmp = num[j];
20.                num[j] = (num[j] + c) / 2;//高位除 2 (数的高位对应数组低位
21.                if (tmp % 2 == 1){//判断 tmp 是否为奇数
22.                    c = 10;// 若 tmp 为奇数, 则该位必有余数 10
23.                }
24.            }
25.        }
26.    }
27. }
```

```
23.         }
24.         else c = 0;
25.     }
26.     if (num[i] == 0) i++;//高位变为0
27. }
28. for (int j = len_str - 1; j >= 0; j--){
29.     printf("%c", buf[j]);
30. }
31. printf("\n");
32. }
33. return 0;
34. }
```

## 题型总结

这类题目任他千变万化, 本质上都是不变的。就是数位的拆解与合并, 拆解很明显就是两步, 先取模然后除取整, 合并就是先乘后加。只要掌握了以上的几类变化, 不管题目如何变化, 你都立于了不败之地。

## 题目练习

DreamJudge 1454 反序数

DreamJudge 1259 进制转换 2

DreamJudge 1176 十进制和二进制

DreamJudge 1380 二进制数

DreamJudge 1417 八进制

DreamJudge 1422 进制转换 3

DreamJudge 1097 负二进制

## 2.3 排版类问题

排版类问题也是机试中经常出现的题目, 这类题目主要考验考生对代码的掌控程度。表面上看起来很简单, 但是对于大部分没有认真研究过的同学来说, 这些题可能会搞半天才能搞出来。

总的来说, 排版类的题目可以以下几种题型为代表。

1、输出字符棱形 DreamJudge 1473

这类题目的变形可以是输出长方形、三角形、梯形等形状。

2、旋转数字输出

3、矩阵顺/逆指针旋转

4、矩阵翻转

这类题目的变形可以是轴对称翻转、中心对称翻转等。

5、杨辉三角形

6、2048 问题



以上, 我们选择其中输出字符棱形和杨辉三角形进行详细讲解, 其他题型我们给出解题思路以及题目编号, 大家可以在本节后面的练习题目里找到并完成。如果自己无法理解并完成题目, 请加入我们的机试交流群进行提问交流。

### 字符棱形

**题目描述:**

输入一个整数  $n$  表示棱形的对角半长度, 请你用\*把这个棱形画出来。

输入: 3

输出:

\*

\*\*\*

\*\*\*\*\*

\*\*\*

\*

输入样例#:

1

输出样例#:

\*

题目来源:

DreamJudge 1473

**解题分析:** 对于这类题目, 我们可以将它进行分解。从中间切开, 上面一个三角形, 下面一个三角形。那么问题就转化为了如何输出三角形, 我们可以利用两个 for 循环控制来输出三角形。



参考代码

```
1. #include <stdio.h>
2.
3. int main() {
4.     int n;
5.     scanf("%d", &n);
6.     //上三角
7.     for (int i = 1; i <= n; i++) {
8.         for (int j = 1; j <= n - i; j++) {
9.             printf(" ");
10.        }
11.        for (int j = n - i + 1; j < n + i; j++) {
12.            printf("*");
13.        }
14.        printf("\n");
15.    }
16.    //下三角 下三角只需要将上三角反过来输出就行
17.    for (int i = n - 1; i >= 1; i--) {
18.        for (int j = 1; j <= n - i; j++) {
19.            printf(" ");
20.        }
21.        for (int j = n - i + 1; j < n + i; j++) {
22.            printf("*");
23.        }
24.    }
25. }
```

```
24.     printf("\n");
25. }
26. return 0;
27. }
```

## 杨辉三角形

### 题目描述:

提到杨辉三角形. 大家应该都很熟悉. 这是我国宋朝数学家杨辉在公元 1261 年著书《详解九章算法》提出的。 1 1 1 1 2 1 1 3 3 1 1 4 6 4 1 1 5 10 10 5 1 1 6 15 20 15 6 1 我们不难其规律: S1: 这些数排列的形状像等腰三角形, 两腰上的数都是 1 S2: 从右往左斜着看, 第一列是 1, 1, 1, 1, 1, 1; 第二列是, 1, 2, 3, 4, 5, 6; 第三列是 1, 3, 6, 10, 15; 第四列是 1, 4, 10, 20; 第五列是 1, 5, 15; 第六列是 1, 6……。 从左往右斜着看, 第一列是 1, 1, 1, 1, 1, 1; 第二列是 1, 2, 3, 4, 5, 6……和前面的看法一样。我发现这个数列是左右对称的。 S3: 上面两个数之和就是下面的一行的数。 S4: 这行数是第几行, 就是第二个数加一。…… 现在要求输入你想输出杨辉三角形的行数 n; 输出杨辉三角形的前 n 行.

### 输入描述:

输入你想输出杨辉三角形的行数 n (n<=20); 当输入 0 时程序结束.

### 输出描述:

noobdream.com

对于每一个输入的数, 输出其要求的三角形. 每两个输出数中间有一个空格. 每输完一个三角形换行.

### 输入样例#:

5

### 输出样例#:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

### 题目来源:

DreamJudge 1062

**解题分析:** 这是一道特别经典的题目, 我们只需要按照题意用二维数组去计算即可。对于任意一个数  $a[i][j]$ , 都有  $a[i][j] = a[i-1][j] + a[i-1][j-1]$ ;

## 参考代码

```
1. #include <stdio.h>
2. int main() {
3.     int a[21][21] = {0}; // 数组里的所有值初始化为 0
4.     int n;
5.     while (scanf("%d", &n) != EOF) {
6.         if (n == 0) break;
7.         a[1][1] = 1;
8.         for (int i = 2; i <= n; i++) {
9.             for (int j = 1; j <= i; j++) {
10.                 a[i][j] = a[i-1][j] + a[i-1][j-1];
11.             }
12.         }
13.         for (int i = 1; i <= n; i++) {
14.             for (int j = 1; j <= i; j++) {
15.                 printf("%d ", a[i][j]);
16.             }
17.             printf("\n");
18.         }
19.     }
20.     return 0;
21. }
```

**题型总结:** 这类题目尽量在平时练习, 解法主要就是把一个大问题进行分解, 一部分一部分的实现。在考试的时候遇到, 千万不要急, 将问题进行分解, 找到其中的规律, 然后再写出来。当然, 如果平时就有练习, 那就不用担心了。

### 练习题目

DreamJudge 1392 杨辉三角形 - 西北工业大学

DreamJudge 1377 旋转矩 - 北航

DreamJudge 1216 旋转方阵

DreamJudge 1221 旋转矩阵

DreamJudge 1472 2048 游戏



## 2.4 日期类问题

日期类的题目也是常考的题目, 这类题目一般都为以下几种考法。

1、判断某年是否为闰年

2、某年某月某日是星期几

变形问法: 某日期到某日期之间有多少天

3、某天之后 x 天是几月几日

4、10:15 分之后 x 分钟是几点几分

变形问法: 某点到某点之间有多少分或多少秒

注意输入时候一般用 `scanf` 解析输入值

如: 2019-11-8 2019-11-08 2019/11/8 10:10



```
1. int year, month, day;
2. scanf("%d-%d-%d", &year, &month, &day);
3. scanf("%d/%d/%d", &year, &month, &day);
4. int hour, minute;
5. scanf("%d:%d", &hour, &minute);
```

noobdream.com

### 日期

**题目描述:**

定义一个结构体变量 (包括年、月、日) , 编程序, 要求输入年月日, 计算并输出该日在本年中第几天。

**输入描述:**

输入三个整数(并且三个整数是合理的, 既比如当输入月份的时候应该在 1 至 12 之间, 不应该超过这个范围)否则输出 Input error!

**输出描述:**

输出一个整数. 既输入的日期是本月的第几天。

**输入样例#:**

1985 1 20

2006 3 12

输出样例#:

20

71

题目来源:

DreamJudge 1051

**解题分析:** 这个题目的考点在于两个地方, 一个是每个月的天数都不一样, 另一个是2月如果是闰年则多一天, 最后我们还要判断输入的日期是否存在, 如果不存在则输出 Input error!

参考代码



```
1. #include <stdio.h>
2.
3. struct node {
4.     int year, month, day;
5. }p;
6. int f[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
7. int main() {
8.     while (scanf("%d%d%d", &p.year, &p.month, &p.day) != EOF) {
9.         //判断是否闰年
10.         if ((p.year%400== 0)|| (p.year%4==0)&&(p.year%100!=0)) {
11.             f[2] = 29;
12.         }
13.         else f[2] = 28;
14.         int flag = 0;
15.         //判断月份输入是否合法
16.         if (p.month < 1 || p.month > 12) flag = 1;
17.         //判断天的输入是否合法
18.         if (p.day < 0 || p.day > f[p.month]) flag = 1;
19.         if (flag) {
20.             printf("Input error!\n");
21.             continue;
22.         }
```

```
23.     int ans = p.day;
24.     for (int i = 1; i < p.month; i++) {
25.         ans += f[i];
26.     }
27.     printf("%d\n", ans);
28. }
29. return 0;
30. }
```

## 题型总结

日期类的题目就是要特别注意闰年的判断, 这些题目一般都是考察代码细节的把握, 时间类的题目注意时间的转换, 1 天=24 小时, 1 小时=60 分, 1 分=60 秒。

特别注意: 一天之内时针和分针会重合 22 次, 而不是 24 次。



noobdream.com

## 练习题目

DreamJudge 1011 日期

DreamJudge 1290 日期差值

DreamJudge 1410 打印日期

DreamJudge 1437 日期类

DreamJudge 1446 日期累加

DreamJudge 1053 偷菜时间表

## 2.5 字符串类问题

字符串类的问题也是各个院校必考的题型之一, 基本上有以下这些考点:

1、统计字符个数

2、单词首字母大写

3、统计子串出现次数

解析: 考察大家基础的字符串遍历能力。

4、文本加密/解密

解析: 通过循环往后移动 x 位或直接给一个映射表是比较常见的考法。

5、文本中的单词反序

解析: 灵活使用 string 可以秒杀这类题目, 当然也可以用字符串一步步解析。

6、删除字符串 (大小写模糊)

解析: 如果大小写不模糊, 那么就是直接找到之后删除。大小写模糊的话, 只是多一个判断。

### 加密算法

**题目描述:**

编写加密程序, 加密规则为: 将所有字母转化为该字母后的第三个字母, 即 A->D、B->E、C->F、.....、Y->B、Z->C。小写字母同上, 其他字符不做转化。输入任意字符串, 输出加密后的结果。

例如: 输入 "I love 007", 输出 "L oryh 007"

**输入描述:**

输入一行字符串, 长度小于 100。

**输出描述:**

输出加密之后的结果。

**输入样例#:**

I love 007

**输出样例#:**

L oryh 007

**题目来源:**

DreamJudge 1014

**题目解析:** 这是一道很常见的加解密考法, 往后移动 3 位是这道题的核心, 我们只需要按照题意将大写字母、小写字母、和其他分开进行处理就可以, 具体看代码。

**参考代码**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char s[105];
6.     gets(s); //输入一行文本用 gets
7.     int len = strlen(s);
8.     for (int i = 0; i < len; i++) {
9.         if (s[i] >= 'A' && s[i] <= 'Z') {
10.             s[i] += 3;
11.             if (s[i] > 'Z') s[i] -= 26; //溢出循环
12.         }
13.         else if (s[i] >= 'a' && s[i] <= 'z') {
14.             s[i] += 3;
15.             if (s[i] > 'z') s[i] -= 26; //溢出循环
16.         }
17.         else {
18.             continue;
19.         }
20.     }
21.     puts(s);
22.     return 0;
23. }
```

**练习题目**

DreamJudge 1012 字符移动

DreamJudge 1292 字母统计

DreamJudge 1240 首字母大写

DreamJudge 1394 统计单词

DreamJudge 1027 删除字符串 2

## 2.6 排序类问题

排序类的问题基本上是每个学校必考的知识点, 所以它的重要性不言而喻。

如果你在网上一查, 或者看数据结构书, 十几种排序算法可以把你吓的魂不守舍。表面上看各种排序都有其各自的特点, 那是不是我们需要掌握每一种排序呢?

答案自然是否定的。我们一种排序也不需要掌握, 你需要会用一个 sort 函数就可以了, 正所谓一个 sort 走天下。

sort 函数本质上是封装了快速排序, 但是它做了一些优化, 所以你只管用它就行了。

复杂度为:  $n \log n$

所以 sort 可以对最大 30W 个左右的元素进行排序, 可以应对考研机试中的 99. 9% 的情况。

sort 函数的用法

sort() 函数: 依次传入三个参数, 要排序区间的起点, 要排序区间的终点+1, 比较函数。比较函数可以不填, 则默认为从小到大排序。

sort 函数有两个常见的应用场景

1、自定义函数排序

noobdream.com

2、多级排序

### 成绩排序

题目描述:

输入任意 (用户, 成绩) 序列, 可以获得成绩从高到低或从低到高的排列, 相同成绩都按先录入排列在前的规则处理。

示例:

jack 70

peter 96

Tom 70

smith 67

从高到低 成绩

peter 96

jack 70

Tom 70

smith 67

从低到高

smith 67

jack 70

Tom 70

peter 96

### 输入描述:

输入多行, 先输入要排序的人的个数, 然后输入排序方法 0 (降序) 或者 1 (升序) 再分别输入他们的名字和成绩, 以一个空格隔开

### 输出描述:

按照指定方式输出名字和成绩, 名字和成绩之间以一个空格隔开

### 输入样例#:

3

0

fang 90

yang 50

ning 70

### 输出样例#:

fang 90

ning 70

yang 50

### 题目来源:

DreamJudge 1151

**题目解析:** 这题唯一的一个考点在于稳定排序, sort排序是不稳定的, 排序之后相对次序有可能发生改变。解决这个问题有两个方法, 一个是用 stable\_sort 函数, 它的用法和 sort 一样, 但是它是稳定的, 所以如果我们遇到有稳定的需求的排序时, 可以用它。另一个方法是给每一个输入增加一个递增的下标, 然后二级排序, 当值相同时, 下标小的排在前面。

### 参考代码 (稳定排序)

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. struct Student {
5.     string name;
6.     int grade;
7. }stu[1005];
8. //从大到小排序
9. bool compareDesc(Student a,Student b) {
10.     return a.grade > b.grade;
11. }
12. //从小到大排序
13. bool compareAsc(Student a,Student b) {
14.     return a.grade < b.grade;
15. }
16. int main() {
17.     int n,order;
18.     while(cin>>n) {
19.         cin>>order;
20.         for(int i=0;i<n;i++) {
21.             cin>>stu[i].name>>stu[i].grade;
22.         }
23.         if(order==0)
24.             stable_sort(stu,stu+n,compareDesc);
25.         else
26.             stable_sort(stu,stu+n,compareAsc);
27.         for(int i=0;i<n;i++) {
28.             cout<<stu[i].name<< " "<<stu[i].grade<<endl;
29.         }
30.     }
31.     return 0;
32. }
```

## 参考代码 (标记 id)

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. struct Student {
5.     string name;
6.     int grade, id;
7. }stu[1005];
8. //从大到小排序
9. bool compareDesc(Student a,Student b) {
10.     if (a.grade == b.grade) return a.id < b.id;
11.     return a.grade > b.grade;
12. }
13. //从小到大排序
14. bool compareAsc(Student a,Student b) {
15.     if (a.grade == b.grade) return a.id < b.id;
16.     return a.grade < b.grade;
17. }
18. int main() {
19.     int n,order;
20.     while(cin>>n) {
21.         cin>>order;
22.         for(int i=0;i<n;i++) {
23.             cin>>stu[i].name>>stu[i].grade;
24.             stu[i].id = i;//通过标记 ID 进行判断
25.         }
26.         if(order==0)
27.             sort(stu,stu+n,compareDesc);
28.         else
29.             sort(stu,stu+n,compareAsc);
30.         for(int i=0;i<n;i++) {
31.             cout<<stu[i].name<<" "<<stu[i].grade<<endl;
32.         }
33.     }
34.     return 0;
35. }
```

## 排序

### 题目描述:

输入  $n$  个数进行排序, 要求先按奇偶后按从小到大的顺序排序。

### 输入描述:

第一行输入一个整数  $n$ , 表示总共有多少个数,  $n \leq 1000$ 。

第二行输入  $n$  个整数, 用空格隔开。

### 输出描述:

输出排序之后的结果。

### 输入样例#:

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

### 输出样例#:

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

### 题目来源:

DreamJudge 1010

**题目解析:** 题目要求我们按照奇数在前偶数在后的排序方法, 同为奇数或同为偶数再从小到大排序。我们有两种简便的方法可以解决这个问题, 其一是我们将奇数和偶数分离开来, 然后分别排好序, 再合并在一起。其二是使用 `sort` 进行二级排序, 这里我们采用第二种方法进行演示。

### 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. bool cmp(int a,int b){
5.     if(a % 2 == b % 2)//如果同奇同偶
6.         return a < b;//直接从小到大排序
7.     else//如果奇偶性不同
8.         return (a%2) > (b%2); //奇数在偶数前
```

```
9. }
10. int main() {
11.     int n;
12.     int a[1005] = {0};
13.     cin >> n;
14.     for (int i = 0; i < n; i++) {
15.         cin >> a[i];
16.     }
17.     sort(a, a+n, cmp);
18.     for(int i = 0; i < n; i++) {
19.         cout << a[i] << " ";
20.     }
21.     cout << endl;
22.     return 0;
23. }
```

小结：由上面可以看出，只要我们掌握好 sort 的用法，不管什么样的花里胡哨的排序，我们都可以一力破之。



### 一些特殊的排序题

1、如果题目给的数据量很大，上百万的数据要排序，但是值的区间范围很小，比如值最大只有 10 万，或者值的范围在 1000W 到 1010W 之间，对于这种情况，我们可以采用空间换时间的计数排序。

2、字符串的字典序排序是一个常见的问题，需要掌握，也是用 sort。

下面两种情况了解即可，追求满分的同学需要掌握

3、如果题目给你一个数的序列，要你求逆序数对有多少，这是一个经典的问题，解法是在归并排序合并时进行统计，复杂度可以达到  $n \log n$ 。如果数据量小，直接冒泡排序即可。

4、如果题目让你求 top10，即最大或最小的 10 个数，如果数据量很大，建议使用选择排序，也就是一个一个找，这样复杂度比全部元素排序要低。

5、如果题目给的数据量有几百万，让你从中找出第  $K$  大的元素，这时候 sort 是会超时的。解法是利用快速排序的划分的性质，进入到其中一个分支继续寻找，

以上都是一些数据很特殊且数据量非常大的情况下的解决方案。

由于部分同学只能用纯 C 语言机试, 这里给出纯 C 语言实现的排序代码模板

普通排序 (适合 n 在 2000 以内的题目)

```
1. #include <stdio.h>
2.
3. const int maxn = 1005;
4. int a[maxn];
5.
6. int main() {
7.     int n;
8.     scanf("%d", &n);
9.     for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
10.    for (int i = 1; i <= n; i++) { //两个 for 都是 1 到 n 方便好记
11.        for (int j = 1; j < n; j++) {
12.            if (a[j] > a[j + 1]) { //交换 a[j] 和 a[j+1]
13.                int temp = a[j];
14.                a[j] = a[j+1];
15.                a[j+1] = temp;
16.            }
17.        }
18.    }
19.    for (int i = 1; i <= n; i++) {
20.        printf("%d ", a[i]);
21.    }
22.    printf("\n");
23.    return 0;
24. }
```

快速排序 (适合 n 在 50W 以内的题目)

```
1. #include <stdio.h>
2.
3. const int maxn = 100005;
4. int a[maxn];
5. //快速排序
6. void Quick_Sort(int l, int r) {
7.     if (l >= r) return;
8.     int i = l, j = r, x = a[l];
9.     while (i < j) {
10.         while (i < j && a[j] >= x) j--;
11.         if (i < j) a[i++] = a[j];
```

```
12.     while (i < j && a[i] < x) i++;
13.     if (i < j) a[j--] = a[i];
14. }
15. a[i] = x;
16. Quick_Sort(l, i - 1);
17. Quick_Sort(i + 1, r);
18. }
19.
20. int main() {
21.     int n;
22.     scanf("%d", &n);
23.     for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
24.     Quick_Sort(1, n); //传入左边界下标和右边界下标
25.     for (int i = 1; i <= n; i++) {
26.         printf("%d ", a[i]);
27.     }
28.     printf("\n");
29.     return 0;
30. }
```



## 练习题目

DreamJudge 1106 排序 2  
DreamJudge 1159 成绩排序 2. 0  
DreamJudge 1217 国名排序  
DreamJudge 1227 日志排序  
DreamJudge 1248 整数奇偶排序  
DreamJudge 1254 字符串排序  
DreamJudge 1255 字符串排序 2  
DreamJudge 1261 字符串排序 3  
DreamJudge 1294 后缀子串排序  
DreamJudge 1310 奥运排序问题  
DreamJudge 1338 EXCEL 排序  
DreamJudge 1360 字符串内排序  
DreamJudge 1399 排序 - 华科  
DreamJudge 1400 特殊排序  
DreamJudge 1404 成绩排序 - 华科  
DreamJudge 1412 大整数排序  
DreamJudge 1817 成绩再次排序  
DreamJudge 1798 数组排序

## 2.7 查找类问题

查找是一类我们必须掌握的算法, 它不仅会在题目中直接考察, 同时也可能是其他算法中的重要组成部分。本章中介绍的查找类问题都是单独的基础查找问题, 对于这类基础查找的问题, 我们应该将它完全掌握。

查找类题目一般有以下几种考点

1、**数字查找**: 给你一堆数字, 让你在其中查找  $x$  是否存在

题目变形: 如果  $x$  存在, 请输出有几个。

2、**字符串查找**: 给你很多个字符串, 让你在其中查找字符串  $s$  是否存在

顺序查找就不说了, 这个大家会。

什么时候不能用顺序查找呢?

很明显, 当满足下面这种情况的时候

1、数据量特别大的时候, 比如有 10W 个元素。

2、查询次数很多的时候, 比如要查询 10W 次。

遇到这类题大多数人的想法是先 sort 排序, 然后二分查找, 这是一个很常规的解决这类问题的方法。

但是, 我们不推荐你这么做, 我们有更简单易用且快速的方法。我们推荐你了解并使用 map 容器。

前面介绍过 map, 它是 STL 的一种关联式容器, 它的底层是红黑树实现的, 也就意味着它的插入和查找操作都是  $\log$  级别的。

相信每一个用过 map 的同学, 都会情不自禁的说一句, map 真香!

## 查找学生信息 2

### 题目描述:

输入  $N$  个学生的信息, 然后进行查询。

### 输入描述:

输入的第一行为  $N$ , 即学生的个数 ( $N \leq 1000$ )

接下来的  $N$  行包括  $N$  个学生的信息, 信息格式如下:

01 李江 男 21

02 刘唐 男 23

03 张军 男 19

04 王娜 女 19

然后输入一个  $M$  ( $M \leq 10000$ ), 接下来会有  $M$  行, 代表  $M$  次查询, 每行输入一个学号, 格式如下:

02

03

01

04

### 输出描述:

输出  $M$  行, 每行包括一个对应于查询的学生的信息。

如果没有对应的学生信息, 则输出 “No Answer!”

### 输入样例#:

4

01 李江 男 21

02 刘唐 男 23

03 张军 男 19

04 王娜 女 19

5

02

03

01

04

03

输出样例#:

```
02 刘唐 男 23
03 张军 男 19
01 李江 男 21
04 王娜 女 19
03 张军 男 19
```

题目来源:

DreamJudge 1476

**题目解析:** 对于这类查询量大的题目, 我们有两种方法来解决这个问题。第一是将学号先排好序, 然后使用二分查找, 但是很多同学写二分的时候容易出现问题, 而且代码量也比较大, 我们不推荐这种做法。推荐大家使用 map 来解决这类问题, 基本上 map 可以通过 99. 9% 的这类题目。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. struct node{
5.     string num;
6.     string name;
7.     string sex;
8.     int age;
9. };
10. int main(){
11.     int n,q;
12.     map<string, node> M;//定义一个 map 映射
13.     while(scanf("%d", &n)!=EOF){
14.         for(int i=0;i<n;i++){
15.             node tmp;
16.             cin>>tmp.num>>tmp.name>>tmp.sex>>tmp.age;
17.             M[tmp.num] = tmp;//将学号指向对应的结构体
18.         }
19.         scanf(" %d", &q);
```

```
20.     for(int i=0;i<q;i++){
21.         string num;
22.         cin>>num;
23.         if((M.find(num))!=M.end())//find 查找 如果找不到则返回末尾
24.             cout<<M[num].num<< " "<<M[num].name<< " "<<M[num].sex<< " "<<M[num].age<<endl;
25.     else
26.         cout<<"No Answer!"<<endl;
27.     }
28. }
29. return 0;
30. }
```

可以发现, 用 map 解决这个题目的时候, 不用去考虑字符串排序的问题, 也不用想二分查找会不会写出问题, 直接用 map, 所有的烦恼都没有了, 而且它的复杂度和二分查找是一个量级的。

上面讲的是一类静态查找的问题, 实际中为了增加思维难度或代码难度, 会经过一定的改变变成动态查找问题。



## 动态查找问题

### 题目描述:

有  $n$  个整数的集合, 想让你从中找出  $x$  是否存在。

### 输入描述:

第一行输入一个正整数  $n$  ( $n < 100000$ )

第二行输入  $n$  个正整数, 用空格隔开。

第三行输入一个正整数  $q$  ( $q < 100000$ ), 表示查询次数。

接下来输入  $q$  行, 每行一个正整数  $x$ , 查询  $x$  是否存在。

### 输出描述:

如果  $x$  存在, 请输出  $find$ , 如果不存在, 请输出  $no$ , 并将  $x$  加入到集合中。

### 输入样例#:

5

1 2 3 4 5

3  
6  
6  
3

输出样例#:

no  
find  
find

题目来源:

DreamJudge 1477

**题目解析:** 通过分析题目我们可以发现, 这道题有一个特点就是, 数的集合在不断的改变。如果我们用先排序再二分的方法就会遇到困难, 因为加入新的数的时候我们需要去移动多次数组, 才能将数插入进去, 最坏情况每次插入都是  $O(n)$  的复杂度, 这是无法接受的。当然也不是说就不能用这样的方法来解决了, 可以用离线的方法来解决这个问题, 但是这样做太复杂, 不适合在考试中使用。那么我们考虑用 map 来解决这个问题。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int main(){
5.     int n,q,x;
6.     map<int, int> M;//定义一个 map 映射
7.     scanf("%d", &n);
8.     for (int i = 0; i < n; i++) {
9.         scanf("%d", &x);
10.        M[x]++;
11.    }
12.    scanf("%d", &q);
13.    for (int i = 0; i < q; i++) {
14.        scanf("%d", &x);
```

```
15.     if (M[x] == 0) {//如果 x 的个数为 0
16.         printf("no\n");
17.         M[x]++;
18.     }
19.     else printf("find\n");
20. }
21. return 0;
22. }
```

看了上面的代码, 是不是发现用 map 来解决问题真是超级简单。所以学会灵活使用 map 将能极大的拉近你和大佬之间的距离, 我们一起来学 map 吧!

当然不是说二分查找就没用了, 我们也需要了解二分查找的原理, 只不过。二分的前提是单调性, 只要满足单调性就可以二分, 不论是单个元素还是连续区间。下面我们也给出一个基本的二分查找代码, 供大家参考。

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int a[10005];
5. int main(){
6.     int n,x;
7.     scanf("%d", &n); //输入 n 个数
8.     for (int i = 1; i <= n; i++) {
9.         scanf("%d", &a[i]);
10.    }
11.    sort(a+1, a+1+n); //排序保持单调性
12.    scanf("%d", &x); //要查找的数 x
13.    int l = 1, r = n;
14.    while (l <= r) {
15.        int mid = (l + r) / 2;
16.        if (a[mid] == x) {
17.            printf("find\n");
18.            return 0;
19.        }
20.        if (a[mid] > x) { //如果 x 比中间数小
21.            r = mid - 1; //说明在左区间
22.        }
23.        else l = mid + 1; //否则在右区间内
24.    }
25. }
```

```
24. }
25. printf("not find\n");
26. return 0;
27. }
```

注: 由于部分同学只能用纯 C 语言机试, 这里给出纯 C 语言实现查找的方法

普通查找即顺序查找, 相信同学们都会

二分查找即将上面的 sort 排序改为上一节纯 C 语言实现的快速排序即可

## 练习题目

DreamJudge 1177 查找学生信息

DreamJudge 1388 查找 1

DreamJudge 1387 查找 - 北邮

DreamJudge 1383 查找第 K 小数

## 2.8 贪心类问题

贪心类问题是常见的考点, 贪心算法更重要的是贪心的思想, 它追求的是当前最优解, 从而得到全局最优解。贪心类问题基本上算是必考题型之一, 它能更好的考察出学生的思维能力以及对问题的分析能力, 很多学校的出题人都非常爱出贪心类的题目。

贪心算法的定义:

贪心算法是指在对问题求解时, 总是做出在当前看来是最好的选择。也就是说, 不从整体最优上加以考虑, 只做出在某种意义上的局部最优解。

贪心算法不是对所有问题都能得到整体最优解, 关键是贪心策略的选择, 选择的贪心策略必须具备无后效性, 即某个状态以前的过程不会影响以后的状态, 只与当前状态有关。

贪心可以很简单, 简单到让所有人一眼就能看出来该怎么做。贪心也可以很难, 难到让你没办法去证明这样贪心的正确性。所以要想解决贪心这类问题, 主要还是看你的悟性, 看你对题目的分析能力如何, 下面我们举例说明。

**例子 1:** 地上有 3 张纸币, 分别是 5 元、1 元和 10 元, 问你只能拿一张, 最多能拿多少钱?

解析: 很明显, 10 元。

**例子 2:** 地上有  $n$  张纸币, 有 1 元的, 有 5 元的, 还要 10 元的, 问你只能拿一张, 最多能拿多少钱?

解析: 很明显, 还是 10 元。

**例子 3:** 地上有很多纸币, 有  $a$  张 1 元的, 有  $b$  张 5 元的, 还要  $c$  张 10 元的, 问你从中拿  $x$  张, 最多能拿多少钱?

解析: 大家应该都能想到, 肯定是优先拿 10 元的, 如果 10 元的拿完了, 再拿 5 元的, 最后才会拿 1 元的。这就是贪心的思想, 所以贪心其实很容易想到的。

**例子 4:** 有  $n$  个整数构成的集合, 现在从中拿出  $x$  个数来, 问他们的和最大能是多少?

解析: 相信大家都能想到, 优先拿大的, 从大到小一个个拿, 这样组成的和最大。那么在解决这个问题之前, 我们需要先排序, 从大到小的排好序, 然后将前  $x$  个数的和累加起来就是答案。

从上面几个例子中, 相信大家对贪心已经有了初步的了解。我们使用贪心的时候, 往往需要先按照某个特性先排好序, 也就是说贪心一般和 sort 一起使用。

### 喝饮料

#### 题目描述:

商店里有  $n$  中饮料, 第  $i$  种饮料有  $m_i$  毫升, 价格为  $w_i$ 。

小明现在手里有  $x$  元, 他想吃尽量多的饮料, 于是向你寻求帮助, 怎么样买才能吃的最多。

请注意, 每一种饮料都可以只买一部分。

#### 输入描述:

有多组测试数据。

第一行输入两个非负整数  $x$  和  $n$ 。

接下来  $n$  行, 每行输入两个整数, 分别为  $m_i$  和  $w_i$ 。

所有数据都不大于 1000。

$x$  和  $n$  都为-1 时程序结束。

#### 输出描述:

请输出小明最多能喝到多少毫升的饮料, 结果保留三位小数。

#### 输入样例#:

233 6

6 1

23 66

32 23

66 66

1 5

8 5

-1 -1

#### 输出样例#:

136.000

#### 题目来源:

DreamJudge 1478

**题目解析:** 通过分析之后我们可以发现, 小明想要喝尽量多的饮料的话, 肯定优先选择性价比最高的饮料喝, 也就是说 1 毫升的价格最低的饮料先喝, 那么我们就需要去比较, 每种饮料 1 毫升的价格是多少。然后按照这个单价从低到高依次排序, 然后一个一个往后喝, 这样可以保证小明能喝到最多的饮料。

## 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. struct node {
5.     double w, m;
6. }p[1005];
7. bool cmp(node a, node b) {
8.     //按照每毫升的价格从低到高排序
9.     return a.w/a.m < b.w/b.m;
10. }
11. int main(){
12.     int n,x;
13.     while (scanf("%d%d", &x, &n) != EOF) {
14.         if (x == -1 && n == -1) break;
15.         for (int i = 1; i <= n; i++) {
16.             scanf("%lf%lf", &p[i].m, &p[i].w);
17.         }
18.         sort(p+1, p+1+n, cmp);
19.         double ans = 0;
20.         for (int i = 1; i <= n; i++) {
21.             if (x >= p[i].w) {//如果剩余的钱能全买
22.                 ans += p[i].m;
23.                 x -= p[i].w;
24.             }
25.             else {//如果剩余的钱买不完这种饮料
26.                 ans += (p[i].m*x/p[i].w);
27.                 break;//到这里 x 已经为 0 了
28.             }
29.         }
30.         printf("%.3lf\n", ans);
31.     }
32.     return 0;
}
```

33. }

## 解题的通用步骤

- 1、建立数学模型来描述问题；
- 2、把求解的问题分成若干个子问题；
- 3、对每一子问题求解，得到子问题的局部最优解；
- 4、把子问题的局部最优解合成原来问题的一个解。

## 题型总结

贪心问题在很多机试难度低的学校，可以成为压轴题，也就是通过人数最少的题目。在机试难度高的学校也是中等难度及以上的题目，为什么明明贪心看起来这么容易的题目，却成为大多数学生过不去的坎呢？原因有二，一是很多同学根本就没有想到这个题目应该用贪心算法，没能将题目抽象成数学模型来分析，简单说就是没有读懂题目隐藏的意思。二是读懂题了，知道应该是贪心算法解这个题目，但是排序的特征点却没有找准，因为不是所有题目都是这么明显的看出来从小到大排序，有的题目可能隐藏的更深，但是这种难度的贪心不常见。所以机试中的贪心题，你要你反应过来这是一个贪心，99%的情况下都能解决。

noobdream.com

## 练习题目

DreamJudge 1307 组队刷题

DreamJudge 1347 To Fill or Not to Fill

## 2.9 链表类问题

链表类问题属于选读章节, 对于使用 OJ 测评的院校的同学来说, 这类问题可以用数组来实现, 没有必要用链表去实现, 写起来慢不说, 还容易出错, 所以我们一般都直接用数组来实现, 反正最后 OJ 能 AC 就行, 建议这类同学跳过本节或仅做了解即可。但是对于非 OJ 测评的院校来说, 链表类问题可以说是必考的题型。

一般来说有以下三种常见考点

### 1、猴子报数

解析: 循环链表建立之后, 按照题意删除节点。

### 2、两个有序链表合并为一个

解析: 这个和两个有序数组合并为一个有序数组原理一样。

### 3、链表排序

解析: 使用冒泡排序进行链表排序, 因为冒泡排序是相邻两个元素进行比较交换, 适合链表。



### 猴子报数

#### 题目描述:

$n$  个猴子围坐一圈并按照顺时针方向从 1 到  $n$  编号, 从第  $s$  个猴子开始进行 1 到  $m$  的报数, 报数到第  $m$  的猴子退出报数, 从紧挨它的下一个猴子重新开始 1 到  $m$  的报数, 如此进行下去知道所有的猴子都退出为止。求给出这  $n$  个猴子的退出的顺序表。

#### 输入描述:

有做组测试数据. 每一组数据有两行, 第一行输入  $n$  (表示猴子的总数最多为 100) 第二行输入数据  $s$  (从第  $s$  个猴子开始报数) 和数据  $m$  (第  $m$  个猴子退出报数). 当输入 0 0 0 时表示程序结束.

#### 输出描述:

每组数据的输出结果为一行, 中间用逗号间隔。

#### 输入样例#:

10

2 5

5  
2 3  
0  
0 0

输出样例#:

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

题目来源:

DreamJudge 1081

**题目解析:** 我们需要创建一个首尾相连的循环链表, 然后先走  $s$  步, 再开始循环遍历链表, 每走  $m$  步删除一个节点, 知道链表中只能下一个节点时结束循环。只能一个节点的判断条件是, 它的下一个指针指向的是它, 说明它自循环了。

参考代码

```
1. #include <stdio.h>
2. #include <malloc.h>
3.
4. struct node {
5.     int num;
6.     struct node *next;
7. };
8. int n, s, m;
9. //创建循环链表
10. struct node* create() {
11.     struct node *head, *now, *pre;
12.     for (int i = 1; i <= n; i++) {
13.         now = (struct node *)malloc(sizeof(node));
14.         if (i == 1) {
15.             head = now;
16.             pre = now;
17.         }
18.         now->num = i;
19.         now->next = head;
20.         pre->next = now;
```

```
21.         pre = now;
22.     }
23.     return head;
24. };
25. //按照题目要求输出
26. void print(struct node *head) {
27.     struct node *p, *pre;//pre 是前一个节点
28.     p = head;
29.     s -= 1;//因为起点在第一个 所以要少走一步
30.     while (s--) {//先走 s 步
31.         pre = p;
32.         p = p->next;
33.     }
34.     int i = 1;
35.     while (p != NULL) {
36.         if (p == p->next) {//只剩最后一个
37.             printf("%d\n", p->num);
38.             break;
39.         }
40.         if (i % m == 0) {//找到第 m 个
41.             printf("%d,", p->num);//输出它
42.             pre->next = p->next;//删除它
43.         }
44.         else pre = p;//这里一定要用 else 如果是删除的话 pre 不变
45.         p = p->next;
46.         i++;
47.     }
48. }
49.
50. int main(){
51.     while (scanf("%d%d%d", &n, &s, &m) != EOF) {
52.         if (n==0&&s==0&&m==0) break;
53.         struct node *head;
54.         head = create();
55.         print(head);
56.     }
57.     return 0;
58. }
```

## 练习题目

DreamJudge 1015 单链表

DreamJudge 1018 击鼓传花

DreamJudge 1025 合并链表

DreamJudge 1405 遍历链表

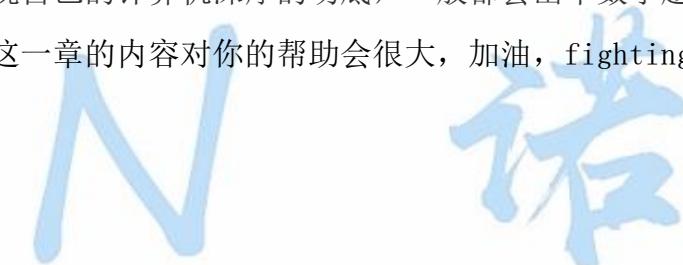


## 第三章 数学

本章我们重点讲解一些常见的数学题型, 包括同模余定理、最大公约数 (GCD)、最小公倍数 (LCM)、斐波那契数列、素数判定、素数筛选、分解素因数、二分快速幂、常见数学公式总结、规律神器 OEIS 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数学问题。

数学算是一个很常考的类别, 毕竟计算机的基础是数学, 就如人类的本质是复读机一样。所以, 出题人为了展现自己的计算机深厚的功底, 一般都会出个数学题来刷一下存在感。

所以, 认真看完这一章的内容对你的帮助会很大, 加油, fighting!



noobdream.com

本书配套视频精讲: <https://www.bilibili.com/video/av81203473>

### 3.1 同模余定理

同余模是一个大家容易忽视的点, 它一般不会单独作为考点出现, 而是在其他类型的题目中作为一个常识的形式出现。

#### 定义

所谓的同余, 顾名思义, 就是许多的数被一个数  $d$  去除, 有相同的余数。 $d$  数学上的称谓为模。如  $a = 6$ ,  $b = 1$ ,  $d = 5$ , 则我们说  $a$  和  $b$  是模  $d$  同余的。因为他们都有相同的余数 1。

数学上的记法为:

$$a \equiv b \pmod{d}$$

可以看出当  $n < d$  的时候, 所有的  $n$  都对  $d$  同商, 比如时钟上的小时数, 都小于 12, 所以小时数都是模 12 的同余。对于同余有三种说法都是等价的, 分别为:

- (1)  $a$  和  $b$  是模  $d$  同余的。
- (2) 存在某个整数  $n$ , 使得  $a = b + nd$ 。
- (3)  $d$  整除  $a - b$ 。

可以通过换算得出上面三个说话都是正确而且是等价的。

#### 定律

同余公式也有许多我们常见的定律, 比如相等律, 结合律, 交换律, 传递律…。如下面的表示:

- 1)  $a \equiv a \pmod{d}$
- 2)  $a \equiv b \pmod{d} \rightarrow b \equiv a \pmod{d}$
- 3)  $(a \equiv b \pmod{d}), (b \equiv c \pmod{d}) \rightarrow a \equiv c \pmod{d}$

如果  $a \equiv x \pmod{d}$ ,  $b \equiv m \pmod{d}$ , 则

- 4)  $a + b \equiv x + m \pmod{d}$
- 5)  $a - b \equiv x - m \pmod{d}$
- 6)  $a * b \equiv x * m \pmod{d}$

#### 应用

$$\begin{aligned}(a+b) \% c &= (a \% c + b \% c) \% c; \\ (a-b) \% c &= (a \% c - b \% c) \% c; \\ (a * b) \% c &= (a \% c * b \% c) \% c;\end{aligned}$$

## 求 $S(n)$

### 题目描述:

$S(n) = n^5$

求  $S(n)$  除以 3 的余数

### 输入描述:

每行输入一个整数  $n$ , ( $0 < n < 1000000$ )

处理到文件结束

### 输出描述:

输出  $S(n) \% 3$  的结果并换行

### 输入样例#:

1

2

### 输出样例#:

1

2

### 题目来源:

DreamJudge 1500

**题目解析:** 通过分析我们可以发现,  $n$  虽然不大, 但是  $n^5$  却超过 long long 的范围, 所幸的是题目只要我们对答案%3, 这时候我们就可以运用同余模定理。

$S(n) \% 3 = (n^5 \% 3) \% 3 = (n \% 3)^5 \% 3 = ((n \% 3) * (n \% 3) * (n \% 3) * (n \% 3) * (n \% 3)) \% 3$

### 参考代码

```
1. #include<stdio.h>
2. #include<iostream>
3. using namespace std;
4.
5. int main() {
6.     long long int n;
7.     while(~scanf("%lld",&n)) {
8.         long long int s=n;
9.         // 同余模定理:
10.        for(int i=1;i<5;i++) {
```

```
11.         s=((s%3)*(n%3));  
12.     }  
13.     printf("%lld\n",s%3);  
14. }  
15. return 0;  
16. }
```

另一类考点就是对大数进行取模

对于大数的求余, 联想到进制转换时的方法, 得到

举例如下, 设大数  $m=1234$ , 模  $n$

就等于

$((((1 * 10) \% n + 2 \% n) \% n * 10 \% n + 3 \% n) \% n * 10 \% n + 4 \% n) \% n$

参考代码 (大数取余)

```
1. #include <stdio.h>  
2.  
3. char s[1000];  
4. int main() {  
5.     int i, j, k, m, n;  
6.     while(~scanf("%s%d", s, &n)) {  
7.         m = 0;  
8.         for(i = 0; s[i] != '\0'; i++)  
9.             m = ((m * 10) % n + (s[i] - '0') % n) % n;  
10.        printf("%d\n", m);  
11.    }  
12.    return 0;  
13. }
```

**特别提醒:** 同余模定理的运算不适用于除法, 对于除法取模的运算我们一般使用逆元来解决问题, 后面的章节中会详细给大家讲解。

## 3.2 最大公约数 (GCD)

最大公约数, 相信大家高中的时候就知道了, 如何用计算机来快速求解两个数的最大公约数是一个很常见的数学问题。

如果题目要求我们求两个数  $x$  和  $y$  的最大公约数, 我们只需要记住下面这种方法就行了。

```
1. #include <stdio.h>
2.
3. int gcd(int a, int b) {
4.     if (b == 0) return a;
5.     else return gcd(b, a % b);
6. }
7. int main() {
8.     int x, y;
9.     scanf("%d%d", &x, &y);
10.    printf("%d\n", gcd(x, y));
11. }
```

显然, 大部分时候题目都不会出的这么直接, 那么对于最大公约数一般会有哪些变形考法呢?

### 最简真分数

#### 题目描述:

给出  $n$  个正整数, 任取两个数分别作为分子和分母组成最简真分数, 编程求共有几个这样的组合。

#### 输入描述:

每组包含  $n$  ( $n \leq 600$ ) 和  $n$  个数, 整数大于 1 且小于等于 1000。

#### 输出描述:

每行输出最简分数组合的个数。

#### 输入样例#:

```
7
3 5 7 9 11 13 15
```

#### 输出样例#:

```
17
```

#### 题目来源:

DreamJudge 1180

题目解析：通过分析题意可以发现，最简真分数的必要条件就是不可以继续约分，那么不可以继续约分，就说明分子和分母的最大公约数为 1。因此，我们只需要枚举所有组合的情况然后判断 GCD 即可。

## 参考代码

```
1. #include <stdio.h>
2.
3. int gcd(int a, int b) {
4.     if(b==0) return a;
5.     else return gcd(b, a%b);
6. }
7. int main() {
8.     int buf[605];
9.     int ans, n;
10.    while(scanf("%d", &n)!=EOF) {
11.        for(int i=0; i<n; i++)
12.            scanf("%d", &buf[i]);
13.        ans=0;//答案个数
14.        for(int i=0; i<n; i++)
15.            for(int j=i+1; j<n; j++)
16.                if (gcd(buf[i], buf[j])==1) ans++;
17.        printf("%d\n", ans);
18.    }
19.    return 0;
20. }
```

## 另一种变形考法是：分数化简

例如：给你一个分数  $12/30$ ，让你将它化简，很明显，我们都知道它的答案是  $2/5$ 。  
那么：如果给你一个分数  $x/y$  呢？如何化简？

我们可以得出： $x/y = (x/\text{gcd}(x, y))/(y/\text{gcd}(x, y))$   
那么只用求出他们的最大公约数，然后除一下就可以得到答案了。

## 练习题目

DreamJudge 1426 最大公约数 1

### 3.3 最小公倍数 (LCM)

对于求两个数的最小公倍数, 只需要记住下面这个公式即可。

$$\text{LCM}(x, y) = x * y / \text{GCD}(x, y)$$

翻译一下就是: 两个数的最小公倍数等于两个数的乘积除以两个数的最大公约数。

所以要求两个数的最小公倍数, 我们只需求出他们的最大公约数即可。

上面的式子经过变形, 可以很容易得到下面这个式子

$$x * y = \text{LCM}(x, y) * \text{GCD}(x, y)$$

延伸出考点:

1、给你两个数的乘积和这两个数的最小公倍数, 问你这两个数的最大公约数是多少?

解析: 很明显, 我们通过上面的公式可知, 乘积除以最小公倍数就是答案。

2、给你两个数的最大公约数和最小公倍数, 问你这两个数的和最大和最小可能是多少?

解析: 这个问题留给读者思考。

### 3.4 斐波那契数列

#### 定义

斐波那契数列指的是这样一个数列 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368.....这个数列从第 3 项开始, 每一项都等于前两项之和。

#### 公式

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

斐波那契一般不会作为一个直接考点出现, 都会结合一些规律让你去推导, 然后发现题目的规律原来是一个斐波那契数列。

在本章最后一个小节中, 用了一个斐波那契的题目作为例子。

我们需要注意的考点是



1、这个数列的上升速度非常快, 很容易超过 int 和 long long 的范围

2、如果对答案取模, 题目就可能要求我们计算第 10000000 项的值, 我们就可以直接使用公式求解, 后面的章节也会讲到用矩阵快速幂求解的方法。

$$\therefore F(n) = 1/\sqrt{5} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right]$$

3、如果给你一个数列:  $a(1) = 1$ ,  $a(n+1) = 1 + 1/a(n)$ 。  
那么它的通项公式为:  $a(n) = \text{fib}(n+1) / \text{fib}(n)$ 。

## 3.5 素数判定

如何判断一个数  $x$  是不是素数呢?

我们可以根据素数的定义从 2 到小于这个数  $x$  的每个数去除, 看是否能除尽。  
一般情况下, 我们没有必要判断这么多个数, 只用判断到  $\sqrt{x}$  就停止了。  
因为: 如果比  $\sqrt{x}$  大的数能除尽的话, 就必然存在一个比  $\sqrt{x}$  小的数能被除尽。

### 判断素数

**题目描述:**

输入一个整数, 判断该整数是否为素数, 若是, 输出该整数, 若否, 输出大于该整数的第一个素数。(例如, 输入为 14, 输出 17, 因为 17 是大于 14 的第一个素数)

**输入描述:**

输入一个整数  $n$ ,  $n$  最大为 10000。

**输出描述:**

按题意输出。

**输入样例#:**

14

**输出样例#:**

17

**题目来源:**

DreamJudge 1013

**题目解析:** 我们首先判断输入的  $n$  是不是一个素数, 如果是的话就直接输出。如果不是的话, 我们从  $n+1$  开始, 对每个数去判断它是不是一个素数, 直到找到一个素数的时候终止。

**参考代码**

```
1. #include <stdio.h>
2. #include <math.h>
3.
4. int main() {
5.     int n;
6.     scanf("%d", &n);
7.     if (n == 1) n++; // 1 不是素数
```

```
8.     for (int i = n; ; i++) {  
9.         int flag = 0;  
10.        for (int j = 2; j <= sqrt(i); j++) {  
11.            if (i % j == 0) {//如果找到了约数  
12.                flag = 1;//说明不是素数  
13.                break;  
14.            }  
15.        }  
16.        if (flag == 0) {  
17.            printf("%d\n", i);  
18.            break;  
19.        }  
20.    }  
21.    return 0;  
22. }
```

练习题目



DreamJudge 1355 素数判定 – 哈尔滨工业大学

noobdream.com

## 3.6 素数筛选

有的时候, 题目要求我们筛选出一段区间内的素数, 我们就需要掌握一种素数筛选的方法。

如果用上一节素数判定的方法去判定每一个数是不是素数的话。

复杂度是  $O(n * \sqrt{n})$ , 大概能处理到 10000 以内的数。

如果题目要求的范围更大, 那么我们就需要一种更为高效的筛选法。

掌握下面这一种线性复杂度的筛选方法就足够我们应对任何情况。

```
1. // 线性素数筛选 prime[0]存的是素数的个数
2. const int maxn = 1000000 + 5;
3. int prime[maxn];
4. void getPrime() {
5.     memset(prime, 0, sizeof(prime));
6.     for (int i = 2; i <= maxn; i++) {
7.         if (!prime[i]) prime[++prime[0]] = i;
8.         for (int j = 1; j <= prime[0] && prime[j] * i <= maxn; j++) {
9.             prime[prime[j] * i] = 1;
10.            if (i % prime[j] == 0) break;
11.        }
12.    }
13. }
```

### 素数判定

**题目描述:**

给你两个数 a、b, 现在的问题是要判断这两个数组成的区间内共有多少个素数

**输入描述:**

多组测试数据。 每个测试数据输入两个数 a、b。 ( $2 \leq a, b \leq 1000$ )

**输出描述:**

输出该区间内素数的个数。

**输入样例#:**

2 4

4 6

输出样例#:

2

1

题目来源:

DreamJudge 1102

**题目解析:** 这道题的数据范围不大, 我们可以用挨个暴力判断的方法来解决。我们假设这道题的数据范围很大, 使用素数筛选的方法来解决这个问题。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. // 线性素数筛选 prime[0]存的是素数的个数
5. const int maxn = 1000000 + 5;
6. int prime[maxn];
7. void getPrime() {
8.     memset(prime, 0, sizeof(prime));
9.     for (int i = 2; i <= maxn; i++) {
10.         if (!prime[i]) prime[++prime[0]] = i;
11.         for (int j = 1; j <= prime[0] && prime[j] * i <= maxn; j++) {
12.             prime[prime[j] * i] = 1;
13.             if (i % prime[j] == 0) break;
14.         }
15.     }
16. }
17. int main() {
18.     getPrime(); //先进行素数筛选预处理
19.     int a, b;
20.     while (scanf("%d%d", &a, &b) != EOF) {
21.         if (a > b) swap(a, b); //a 有可能比 b 大
22.         int ans = 0;
23.         for (int i = 1; i <= prime[0]; i++) {
24.             if (prime[i] >= a) ans++; //素数大于 a 答案加一
25.             if (prime[i] > b) {
26.                 ans--; //大于 b 要减回来
27.             break;
28.         }
29.     }
30. }
```

```
28.         }
29.     }
30.     printf("%d\n", ans);
31. }
32. return 0;
33. }
```

### 练习题目

DreamJudge 1375 素数



## 3.7 分解素因数

我们知道, 对于任意一个大于 1 的整数, 它都可以分解成多个质因子相乘的形式。

### 质因数个数

#### 题目描述:

求正整数  $N (N > 1)$  的质因数的个数。 相同的质因数需要重复计算。如  $120 = 2 * 2 * 2 * 3 * 5$ , 共有 5 个质因数。

#### 输入描述:

可能有多组测试数据, 每组测试数据的输入是一个正整数  $N$ , ( $1 < N < 10^9$ )。

#### 输出描述:

对于每组数据, 输出  $N$  的质因数的个数。

#### 输入样例#:

120

#### 输出样例#:

5

#### 题目来源:

DreamJudge 1156

**题目解析:** 我们可以通过上一节学会的素数筛选, 先将所有的素数筛选出来。然后再不断的去分解素数, 直到将数分解到 1 为止。由于我们的素数筛选只能到 1000000, 对于更大的素因子我们可以不继续分解, 因为不会存在两个大于 1000000 的素因子, 如果存在, 那么这个数的范围一定大于题目所给的范围  $10^9$ 。

## 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. // 线性素数筛选 prime[0]存的是素数的个数
5. const int maxn = 1000000 + 5;
6. int prime[maxn];
7. void getPrime() {
8.     memset(prime, 0, sizeof(prime));
9.     for (int i = 2; i <= maxn; i++) {
10.         if (!prime[i]) prime[++prime[0]] = i;
11.         for (int j = 1; j <= prime[0] && prime[j] * i <= maxn; j++) {
12.             prime[prime[j] * i] = 1;
13.             if (i % prime[j] == 0) break;
14.         }
15.     }
16. }
17. int main() {
18.     getPrime(); //先进行素数筛选预处理
19.     int n;
20.     while (scanf("%d", &n) != EOF) {
21.         int ans = 0;
22.         for (int i = 1; i <= prime[0]; i++) {
23.             while (n % prime[i] == 0) {
24.                 n /= prime[i];
25.                 ans++;
26.             }
27.         }
28.         if (n > 1) ans++;
29.         printf("%d\n", ans);
30.     }
31.     return 0;
32. }
```

## 练习题目

DreamJudge 1284 整除问题

DreamJudge 1464 最大素因子

### 3.8 二分快速幂

有一类题目是这样的

求  $(x^y) \% p$

当  $y$  很大的时候, 我们不能直接用 for 去一个一个的乘, 因为这样的方法复杂度是  $O(N)$  的。  
那么有没有更高效的方法呢?

遇到这类问题的时候, 我们就可以使用二分快速幂的方法来求解。

举个例子

$$3^7 = 3^4 * 3^2 * 3^1$$

首先我们要知道, 对于任意一个数  $s$ , 它的二进制代表了它可以由 2 的次幂的累加和来表示。

比如 7 的二进制是 111

$$\text{那么它就是说 } 7 = 2^2 + 2^1 + 2^0$$

再比如一个数 12 的二进制是 1101

$$13 = 2^3 + 2^2 + 2^0$$

所以对于任意一个  $x^y$ , 我们都可以将  $y$  分解成 2 的幂次的形式。

$$\text{例如 } 5^{13} = 5^8 * 5^4 * 5^1$$

这样做有什么好处呢?

$$1..2..4..8..16..32..64..128.....1024....$$

你发现其中的奥秘了吗?

1、每一个数都是它前一个数的 2 倍

2、它的迭代速度超级快

#### 幂次方

题目描述:

对任意正整数  $N$ , 求  $X^N \% 233333$  的值。

要求运算的时间复杂度为  $O(\log N)$ 。

例如

$$X^{30} = X^{15} * X^{15}$$

$$X^{15} = X^7 * X^7 * X$$

$$X^7 = X^3 * X^3 * X$$

$$X^3 = X * X * X$$

共 7 次乘法运算完毕。

**输入描述:**

输入两个整数 X 和 N, 用空格隔开, 其中 X, N<=10^9。

**输出描述:**

输出  $X^N$  对 233333 取模的结果。

**输入样例#:**

2 5

**输出样例#:**

32

**题目来源:**

DreamJudge 1017

**题目解析:** 运用上面讲过的二分快速幂思想和同余模定理即可。

**参考代码**

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. typedef long long ll;
5. ll mod_pow(ll x, ll y, ll mod) {
6.     ll res = 1;
7.     while (y > 0) {
8.         //如果二进制最低位为 1、则乘上 x^(2^i)
9.         if (y & 1) res = res * x % mod;
10.        x = x * x % mod; // 将 x 平方
11.        y >>= 1;
12.    }
13.    return res;
14. }
15.
16. int main() {
17.     ll x, n;//注意 x*x 会超出 int 范围
18.     scanf("%lld%lld", &x, &n);
19.     printf("%lld\n", mod_pow(x, n, 233333));
20.     return 0;
21. }
```

## 3.9 常见数学公式总结

### 错排公式

问题: 十本不同的书放在书架上。现重新摆放, 使每本书都不在原来放的位置。有几种摆法?

递推公式为:  $D(n) = (n - 1) * [D(n - 1) + D(n - 2)]$

### 海伦公式

$$S = \sqrt{p * (p - a) * (p - b) * (p - c)}$$

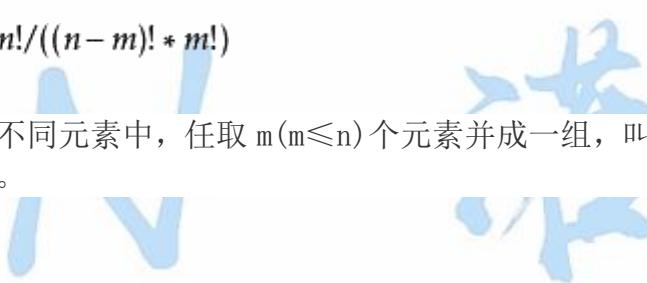
公式描述: 公式中  $a, b, c$  分别为三角形三边长,  $p$  为半周长,  $S$  为三角形的面积。

### 组合数公式

$$c(n, m) = p(n, m)/m! = n!/((n - m)! * m!)$$

公式描述:

组合数公式是指从  $n$  个不同元素中, 任取  $m$  ( $m \leq n$ ) 个元素并成一组, 叫做从  $n$  个不同元素中取出  $m$  个元素的一个组合。



### 两点之间的距离公式

$$|AB| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

公式描述: 公式中  $(x_1, y_1), (x_2, y_2)$  分别为  $A, B$  两个点的坐标。

扇形面积:  $S = 1/2 \times \text{弧长} \times \text{半径}$ ,  $S_{\text{扇}} = (n/360) \pi R^2$

## 卡特兰数

原理:

令  $h(0)=1, h(1)=1$ , catalan 数满足递归式:

$$h(n) = h(0)*h(n-1) + h(1)*h(n-2) + \dots + h(n-1)*h(0) \quad (\text{其中 } n \geq 2)$$

另类递推公式:

$$h(n) = h(n-1) * (4 * n - 2) / (n + 1)$$

该递推关系的解为:

$$h(n) = C(2n, n) / (n + 1) \quad (n = 1, 2, 3, \dots)$$

卡特兰数的应用实质上都是递归等式的应用

前几项为: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, ...

所有的奇卡塔兰数  $C_n$  都满足  $n = 2^k - 1$ 、所有其他的卡塔兰数都是偶数。

## 应用

1、12 个高矮不同的人, 排成两排, 每排必须是从矮到高排列, 而且第二排比对应的第一排的人高, 问排列方式有多少种?

2、括号化问题。矩阵链乘:  $P = A_1 \times A_2 \times A_3 \times \dots \times A_n$ , 依据乘法结合律, 不改变其顺序, 只用括号表示成对的乘积, 试问有几种括号化的方案?

3、将多边形划分为三角形问题。将一个凸多边形区域分成三角形区域(划分线不交叉)的方法数?

4、出栈次序问题。一个栈(无穷大)的进栈序列为 1、2、3、...、n, 有多少个不同的出栈序列?

5、通过连结顶点而将  $n + 2$  边的凸多边形分成三角形的方法个数。

6、所有在  $n \times n$  格点中不越过对角线的单调路径的个数。

7、有  $n+1$  个叶子的二叉树的个数。

更多的变化不胜枚举, 遇到这类规律题建议使用下一节讲的 OEIS 来解决问题。

## 3.10 规律神器 OEIS

下面给大家隆重介绍一个神器, 所有的考生, 看这里: OEIS。

对, 就是它, 考试必备神器, 你值得拥有。有了它, 你可以变得更自信, 你可以满怀信心的走进考场, 你可以飞快解出那些所谓推公式的规律难题。

它的网址: <http://oeis.org/>

它究竟有多变态, 只要你输入前几项, 它就可以给你找出规律来, 并且自动给你推出公式。所以对于找规律的题目, 你只需要手动计算出前几项的值, 或者暴力打表求出前几项的值, 然后输入 OEIS, 他就可以告诉你公式是什么。

怎么样, 超级有用吧! 妈妈再也不用担心我的数学了^\_^

What! 你说它考试的时候没用? 难道如何优雅的上厕所也需要教?

所有的考生, 一定要学会, 用它! 用它! 一定要用它!

只用花一分钟你就学会用这个东西了, 如果你不会用, 而你的竞争对手会用, 那你就惨了, 说不定你推到死都推不出来的公式, 别人几秒钟就搞定了。反之, 你学会了用这个东西, 而你的竞争对手不会, 到时候你会感觉爽翻天(\*^▽^\*)

快来用这个神器做一下下面这个题练练手吧

### 01 字符串

题目描述:

给你一串长度为  $n$  的全为 0 的字符串, 你可以进行一个压缩操作, 将两个相邻的 0 压缩成一个 1。请问最多会有多少种组合出现?

例如  $n$  为 3 则有下面 3 种组合:

000

10

01

**输入描述:**

输入一个正整数 n (1<=n<=10000) 。

**输出描述:**

输出最多有多少种组合出现, 由于结果可能过大, 请将答案对 2333333 取模。

**输入样例#:**

3

**输出样例#:**

3

**题目来源:**

DreamJudge 1479

**题目解析:** 对于这类让我们去找有多少种情况的题目, 第一反应就是他是有规律的, 它的规律可能简单, 也可能很难。这时候我们先推出它的前几项值, n 为 1 时答案等于 1, n 为 2 时答案等于 2, n 为 3 时答案等于 3, n 为 4 时答案等于 5。然后我们将这前几项值 1, 2, 3, 5 输入到 OEIS 中, 它立马告诉我们这个规律是斐波那契数列,  $f[i] = f[i-1] + f[i-2]$ 。

**参考代码**

```
1. #include <stdio.h>
2.
3. long long f[10005] = {0};
4. int main(){
5.     int n;
6.     scanf("%d", &n);
7.     f[0] = 1; f[1] = 1;
8.     for (int i = 2; i <= n; i++) {
9.         f[i] = f[i-1] + f[i-2];
10.        f[i] %= 2333333;
11.    }
12.    printf("%lld\n", f[n]);
13.    return 0;
14. }
```

上面这个题目规律很简单, 可能你不借助 OEIS 都能推出来, 但是很多时候规律的公式很复杂, 涉及到组合数之类的, 手推是非常难推出来的。既然能几秒钟解决的问题, 为什么要花大量时间去手推公式呢? 多留点时间做其他题不好吗?

## 第四章 高精度问题

本章我们重点讲解一些常见的高精度题型, 包括 python 解法、java 解法、C/C++解法等内容。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高精度问题。



本书配套视频精讲: <https://www.bilibili.com/video/av81203473>

## 4.1 Python 解法

人生苦短, 我用 python!

如果你的考试院校支持用 python, 那真是太好不过了。Python 真是超级棒的一门语言, 虽然 python 速度慢, 但是那及其丰富的库和框架, 再加上简洁的语法, 真是美妙至极。

对于高精度的大数类问题, 对 python 来说简直来说就是小菜一碟。

什么? 你在问我 python 大数应该怎么样用?

Are you kidding me? 在 python 眼中就没有大数这个东西。

我们还是假装给一下两个大数相加的代码。

```
1. while True:
2.     try:
3.         a, b = map(int, input().split())
4.         c = a+b
5.         print(c)
6.     except:
7.         break
```

所以只要你会用 python 就可以了。

### 练习题目

DreamJudge 1474 大整数加法

DreamJudge 1475 大整数乘法

## 4.2 Java 解法

基本上所有的 OJ 都支持 Java, 所以建议大家使用 Java 来解决高精度的题目。

BigDecimal (表示浮点数) 和 BigInteger (表示整数) 加上

```
import java.math.*  
1. valueOf(parament); //将参数转换为指定类型  
2. add(); //大数加法  
3. subtract(); //减法  
4. multiply(); //乘法  
5. divided(); //相除取整  
6. remainder(); //取余  
7. pow(); //a.pow(b) = a ^ b  
8. gcd(); //最大公约数  
9. abs(); //绝对值  
10. negate(); //取反数  
11. mod(); //a.mod(b) = a % b = a.remainder(b)  
12. max(); min();  
13. public int compareTo(); //比较  
14. boolean equals(); //比较是否相等
```

参考代码

noobdream.com

```
1. import java.math.BigInteger;  
2. import java.util.Scanner;  
3.  
4. public class Main {  
5.  
6.     public static void main(String[] args) {  
7.         Scanner sr=new Scanner(System.in);  
8.         while (sr.hasNext()) {  
9.             BigInteger a,b;  
10.            a=sr.nextBigInteger();  
11.            b=sr.nextBigInteger();  
12.            System.out.println(a.add(b));  
13.        }  
14.    }  
15. }
```

## 4.3 C/C++解法

C/C++可以通过模拟的方法解决高精度的问题，但是我们不是特别建议在考试的时候自己手动去模拟大整数的问题，这样很容易出现失误。

当然，如果是很简单的加减法运算，用 C/C++模拟也是挺不错的，毕竟更换 IDE 也挺麻烦的。

下面给出 C/C++大数加法的代码

```
1. #include <iostream>
2. #include <string>
3. #include <algorithm>
4. using namespace std;
5.
6. string Add(string a, string b) {
7.     //a 一直为位数较长的字符串
8.     if (a.length() < b.length()) a.swap(b);
9.
10.    string result(a.length(), 0); //初步设置 result 长度为较长字符长度
11.    b.insert(0, a.length() - b.length(), '0'); //较短的字符串前面补零方便计算
12.    int carry = 0; //进位
13.    for (int i = a.length() - 1; i >= 0; i--) {
14.        int sum = (a[i] - 48) + (b[i] - 48) + carry;
15.        carry = sum / 10;
16.        result[i] = sum % 10 + 48;
17.    }
18.    //若进位不为 0, 还要在前面补上进位
19.    if (carry != 0) {
20.        result.insert(result.begin(), carry + 48);
21.    }
22.    return result;
23. }
24.
25. int main() {
26.     string a, b;
27.     while (cin >> a >> b)
28.         cout << Add(a, b) << endl;
29.     return 0;
30. }
```

## 第五章 数据结构

本章我们重点讲解一些常见的数据结构题型, 包括栈的应用、哈夫曼树、二叉树、二叉排序树、hash 算法、前缀树等内容。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数据结构问题。



本书配套视频精讲: <https://www.bilibili.com/video/av81203473>

## 5.1 栈的应用

栈是一种只能在一端进行插入和删除操作的数据结构, 它满足先进后出的特性。

我们通过 `stack<int> S` 来定义一个全局栈来储存整数的空的 `stack`。当然 `stack` 可以存任何类型的数据, 比如 `stack<string> S` 等等。

```
1. #include <iostream>
2. #include <stack>
3. using namespace std;
4. stack<int> S;
5. int main() {
6.     S.push(1);
7.     S.push(10);
8.     S.push(7);
9.     while (!S.empty()) {
10.         cout << S.top() << endl;
11.         S.pop();
12.     }
13.     return 0;
14. }
```

只能使用 C 语言机试的同学直接用数组模拟栈即可。

```
1. //栈和队列都可以用数组模拟实现
2. #include <stdio.h>
3.
4. int st[1005];//定义一个栈
5. int main() {
6.     int top = 0;//栈顶下标
7.     st[++top] = 1;//将 1 入栈
8.     st[++top] = 10;//将 10 入栈
9.     st[++top] = 7;//将 7 入栈
10.    while (top > 0) {//当栈不为空
11.        printf("%d\n", st[top]);//输出栈顶元素
12.        top--;//将栈顶元素出栈
13.    }
14.    return 0;
15. }
```

## 括号的匹配

### 题目描述:

假设表达式中允许包含两种括号:圆括号和方括号。编写一个算法判断表达式中的括号是否正确配对。

### 输入描述:

由括号构成的字符串, 包含” ( “、” ) “、” [ “和” ] “。

### 输出描述:

如果匹配输出 YES, 否则输出 NO。

### 输入样例#:

[([]()])

### 输出样例#:

YES

### 题目来源:

DreamJudge 1501

**题目解析:** 用栈模拟即可, 和栈顶元素匹配就说明配对成功, 将栈顶元素出栈, 否则配对不成功, 就将当前元素入栈。最后查看栈是否为空, 若为空则是 YES, 否则就是 NO。

### 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int main() {
5.     char s[300];
6.     scanf("%s", &s);
7.     int len = strlen(s);
8.     stack<char> st;
9.     for (int i = 0; i < len; i++) {
10.         if (!st.empty()) {
11.             char now = st.top();
12.             if (s[i] == ')' && now == '(' || s[i] == ']' && now == '[')
```

```
13.         st.pop();
14.         else st.push(s[i]);
15.     }
16.     else st.push(s[i]);
17. }
18. if (!st.empty()) printf("NO\n");
19. else printf("YES\n");
20. return 0;
21. }
```

其他比较常见的应用方式

- 1、计算表达式的值
- 2、带优先级的括号匹配问题

### 练习题目

DreamJudge 1067 括号的匹配

DreamJudge 1296 括号匹配问题

DreamJudge 1838 括号匹配

noobdream.com

## 5.2 哈夫曼树

### 基本概念

给定  $N$  个权值作为  $N$  个叶子结点, 构造一棵二叉树, 若该树的带权路径长度达到最小, 称这样的二叉树为最优二叉树, 也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树, 权值较大的结点离根较近。

### 哈夫曼树题目一般有以下几种考点

- 1、直接要求构造哈夫曼树, 输出 WPL, 即带权路径长度。
- 2、考察概念的理解, 如下面的例题, 合并果子。
- 3、考察哈夫曼编码



#### 题目描述:

在一个果园里, 多多已经将所有的果子打了下来, 而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。每一次合并, 多多可以把两堆果子合并到一起, 消耗的体力等于两堆果子的重量之和。可以看出, 所有的果子经过  $n-1$  次合并之后, 就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。因为还要花大力气把这些果子搬回家, 所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1, 并且已知果子的种类数和每种果子的数目, 你的任务是设计出合并的次序方案, 使多多耗费的体力最少, 并输出这个最小的体力耗费值。例如有 3 种果子, 数目依次为 1, 2, 9。可以先将 1、2 堆合并, 新堆数目为 3, 耗费体力为 3。接着, 将新堆与原先的第三堆合并, 又得到新的堆, 数目为 12, 耗费体力为 12。所以多多总共耗费体力  $= 3 + 12 = 15$ 。可以证明 15 为最小的体力耗费值。

#### 输入描述:

输入包括两行, 第一行是一个整数  $n$  ( $1 \leq n \leq 10000$ ), 表示果子的种类数。第二行包含  $n$  个整数, 用空格分隔, 第  $i$  个整数  $a_i$  ( $1 \leq a_i \leq 20000$ ) 是第  $i$  种果子的数目。

#### 输出描述:

输出包括一行, 这一行只包含一个整数, 也就是最小的体力耗费值。输入数据保证这个值小于

$2^{31}$ 。

输入样例#:

```
3
1 2 9
```

输出样例#:

```
15
```

题目来源:

```
DreamJudge 1544
```

**题目解析:** 同学们读了这个题以后, 大概都能想到, 想要求得最小值必然是每次合并最小两个元素即可。然后一看输入样例, 就认为只要从小到大排个序即可, 然后从前往后依次合并。实际上这个方法是错误的, 随手就能举出一个反例, 5, 6, 7, 8 这四个数, 如果先合并 5 和 6, 然后再和 7 合并, 然后再和 8 合并这样得到的结果就会偏大, 正确的方法是先合并 5 和 6, 然后再合并 7 和 8, 最后 11 和 15 合并在一起, 这样才是最优解。为什么呢? 因为在合并的过程中产生的新的数不一定是最小的, 所以在每一次合并的过程中我们都需要重新排序找出当前最小的两个数。但是如果每一次合并就要重新全部排序, 复杂度太高, 不能接受。于是, 我们就想到了使用优先队列来维护单调性, 问题就解决了。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. struct node {
5.     int x;
6.     node(int a) {x = a;}//构造函数方便赋值
7. };
8.
9. //定义优先队列的比较关系 和 sort 的 cmp 类似
10. bool operator < (const node& a, const node& b ){
11.     return a.x > b.x;
12. }
13.
14. int main()
15. {
```

```
16. priority_queue<node> q;
17. int n, x;
18. cin >> n;
19. for(int i = 0; i < n; i++) {
20.     cin >> x;
21.     q.push(x);
22. }
23. int ans = 0;
24. while (q.size() > 1) {
25.     node num1 = q.top();
26.     q.pop();
27.     node num2 = q.top();
28.     q.pop();
29.     ans += (num1.x + num2.x);
30.     q.push(node{num1.x+num2.x});
31. }
32. cout << ans;
33. return 0;
34. }
```

注：如果是纯 C 语言机试的同学，数据不大的情况下，可以用合并之后重新排序的暴力算法，如果数据量大，建议用后面一节的二叉排序树来实现，如果二叉排序树平衡，新增一个点和删除一个点都是  $\log$  级别的复杂度，可以满足数据量大的需求。

noobdream.com

## 哈夫曼编码

哈夫曼树的应用很广，哈夫曼编码就是其在电讯通信中的应用之一。广泛地用于数据文件压缩的十分有效的编码方法。其压缩率通常在 20%~90% 之间。在电讯通信业务中，通常用二进制编码来表示字母或其他字符，并用这样的编码来表示字符序列。

例：如果需传送的电文为 ‘ABACCDA’，它只用到四种字符，用两位二进制编码便可分辨。假设 A, B, C, D 的编码分别为 00, 01, 10, 11，则上述电文便为 ‘00010010101100’（共 14 位），译码员按两位进行分组译码，便可恢复原来的电文。

能否使编码总长度更短呢?

实际应用中各字符的出现频度不相同, 用短(长)编码表示频率大(小)的字符, 使得编码序列的总长度最小, 使所需总空间量最少

### 数据的最小冗余编码问题

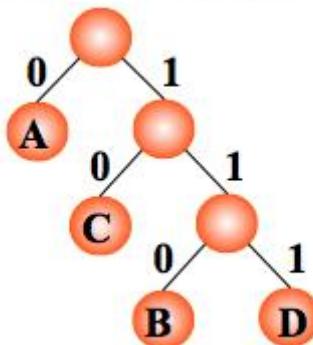
在上例中, 若假设 A, B, C, D 的编码分别为 0, 00, 1, 01, 则电文 ‘ABACCDA’ 便为 ‘0000011010’ (共 9 位), 但此编码存在多义性: 可译为: ‘BBCCDA’、‘ABACCDA’、‘AAAACCACA’ 等。

### 译码的惟一性问题

要求任一字符的编码都不能是另一字符编码的前缀, 这种编码称为前缀编码 (其实是非前缀码)。在编码过程要考虑两个问题, 数据的最小冗余编码问题, 译码的惟一性问题, 利用最优二叉树可以很好地解决上述两个问题

### 用二叉树设计二进制前缀编码

以电文中的字符作为叶子结点构造二叉树。然后将二叉树中结点引向其左孩子的分支标 ‘0’, 引向其右孩子的分支标 ‘1’; 每个字符的编码即为从根到每个叶子的路径上得到的 0, 1 序列。如此得到的即为二进制前缀编码。



编码: A: 0, C: 10, B: 110, D: 111

任意一个叶子结点都不可能在其它叶子结点的路径中。

### 用哈夫曼树设计总长最短的二进制前缀编码

假设各个字符在电文中出现的次数 (或频率) 为  $w_i$ , 其编码长度为  $l_i$ , 电文中只有  $n$  种字

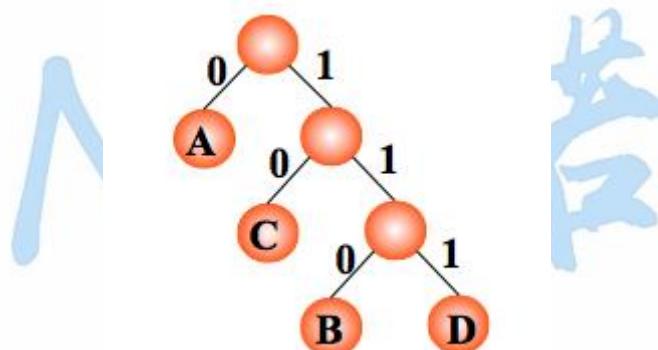
符, 则电文编码总长为:

$$WPL = \sum_{i=1}^n w_i l_i \xrightarrow{\text{从根到叶子的路径长度}} \text{叶子结点的权}$$

设计电文总长最短的编码, 设计哈夫曼树 (以  $n$  种字符出现的频率作权),

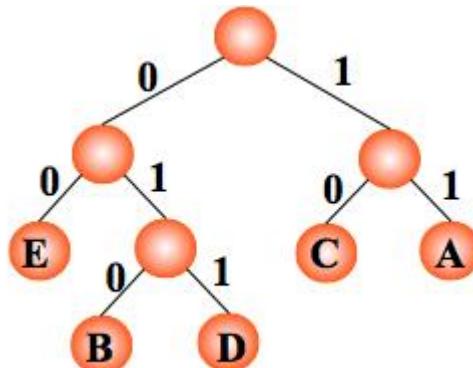
由哈夫曼树得到的二进制前缀编码称为哈夫曼编码

例: 如果需传送的电文为 ‘ABACCDA’ , 即: A, B, C, D 的频率 (即权值) 分别为 0.43, 0.14, 0.29, 0.14, 试构造哈夫曼编码。



编码: A: 0, C: 10, B: 110, D: 111。电文 ‘ABACCDA’ 便为 ‘0110010101110’ (共 13 位)。

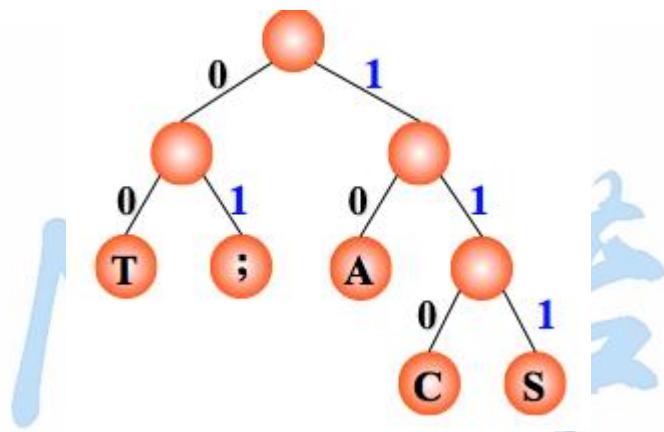
例: 如果需传送的电文为 ‘ABCACCDAEAE’ , 即: A, B, C, D, E 的频率 (即权值) 分别为 0.36, 0.1, 0.27, 0.1, 0.18, 试构造哈夫曼编码。



编码: A: 11, C: 10, E: 00, B: 010, D: 011, 则电文 ‘ABCACCDAAEAE’ 便为 ‘110101011101001111001100’ (共 24 位, 比 33 位短)。

## 译码

从哈夫曼树根开始, 对待译码电文逐位取码。若编码是“0”, 则向左走; 若编码是“1”, 则向右走, 一旦到达叶子结点, 则译出一个字符; 再重新从根出发, 直到电文结束。



电文为 “1101000”, 译文只能是 “CAT”

noobdream.com

## 练习题目

DreamJudge 1382 哈夫曼树

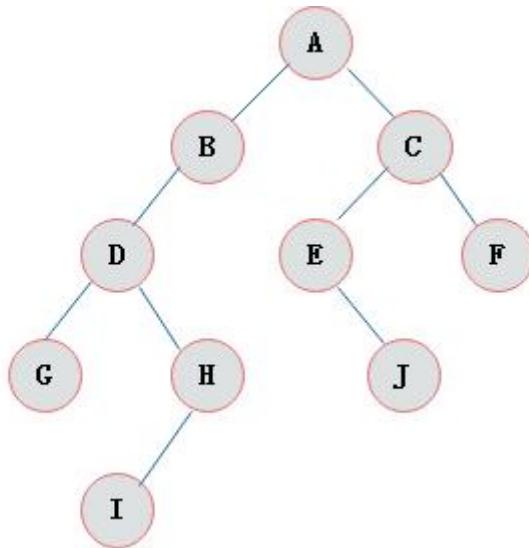
DreamJudge 1562 哈夫曼编码

## 5.3 二叉树

### 定义

二叉树是  $n(n \geq 0)$  个结点的有限集合, 该集合或者为空集 (称为空二叉树), 或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树组成。

下图展示了一棵普通二叉树:



### 二叉树特点

noobdream.com

由二叉树定义以及图示分析得出二叉树有以下特点:

- 1) 每个结点最多有两颗子树, 所以二叉树中不存在度大于 2 的结点。
- 2) 左子树和右子树是有顺序的, 次序不能任意颠倒。
- 3) 即使树中某结点只有一棵子树, 也要区分它是左子树还是右子树。

### 二叉树性质

- 1) 在二叉树的第  $i$  层上最多有  $2^{i-1}$  个节点。 ( $i \geq 1$ )
- 2) 二叉树中如果深度为  $k$ , 那么最多有  $2^k - 1$  个节点。 ( $k \geq 1$ )
- 3)  $n_0 = n_2 + 1$   $n_0$  表示度数为 0 的节点数,  $n_2$  表示度数为 2 的节点数。
- 4) 在完全二叉树中, 具有  $n$  个节点的完全二叉树的深度为  $\lceil \log_2 n \rceil + 1$ , 其中  $\lceil \log_2 n \rceil$  是向下取整。

5) 若对含  $n$  个结点的完全二叉树从上到下且从左至右进行 1 至  $n$  的编号, 则对完全二叉树中任意一个编号为  $i$  的结点有如下特性:

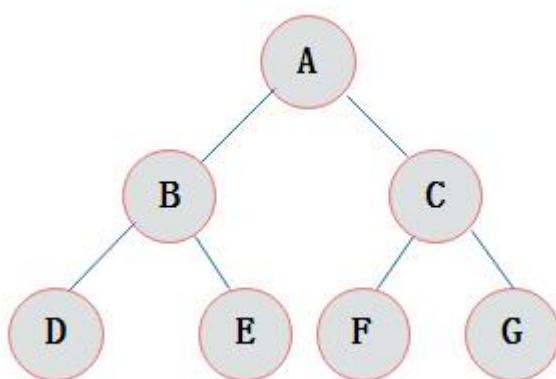
- (1) 若  $i=1$ , 则该结点是二叉树的根, 无双亲, 否则, 编号为  $[i/2]$  的结点为其双亲结点;
- (2) 若  $2i>n$ , 则该结点无左孩子, 否则, 编号为  $2i$  的结点为其左孩子结点;
- (3) 若  $2i+1>n$ , 则该结点无右孩子结点, 否则, 编号为  $2i+1$  的结点为其右孩子结点。

## 满二叉树

满二叉树: 在一棵二叉树中。如果所有分支结点都存在左子树和右子树, 并且所有叶子都在同一层上, 这样的二叉树称为满二叉树。

满二叉树的特点有:

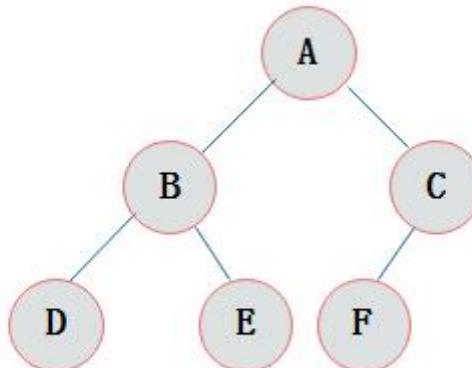
- 1) 叶子只能出现在最下一层。出现在其它层就不可能达成平衡。
- 2) 非叶子结点的度一定是 2。
- 3) 在同样深度的二叉树中, 满二叉树的结点个数最多, 叶子数最多。



## 完全二叉树

完全二叉树: 对一颗具有  $n$  个结点的二叉树按层编号, 如果编号为  $i(1 \leq i \leq n)$  的结点与同样深度的满二叉树中编号为  $i$  的结点在二叉树中位置完全相同, 则这棵二叉树称为完全二叉树。

下图展示一棵完全二叉树



特点：

- 1) 叶子结点只能出现在最下层和次下层。
- 2) 最下层的叶子结点集中在树的左部。
- 3) 倒数第二层若存在叶子结点，一定在右部连续位置。
- 4) 如果结点度为 1，则该结点只有左孩子，即没有右子树。
- 5) 同样结点数目的二叉树，完全二叉树深度最小。

注：满二叉树一定是完全二叉树，但反过来不一定成立。

noobdream.com

## 二叉树遍历

二叉树的遍历一个重点考查的知识点。

二叉树的遍历是指从二叉树的根结点出发，按照某种次序依次访问二叉树中的所有结点，使得每个结点被访问一次，且仅被访问一次。

二叉树的访问次序可以分为四种：

前序遍历

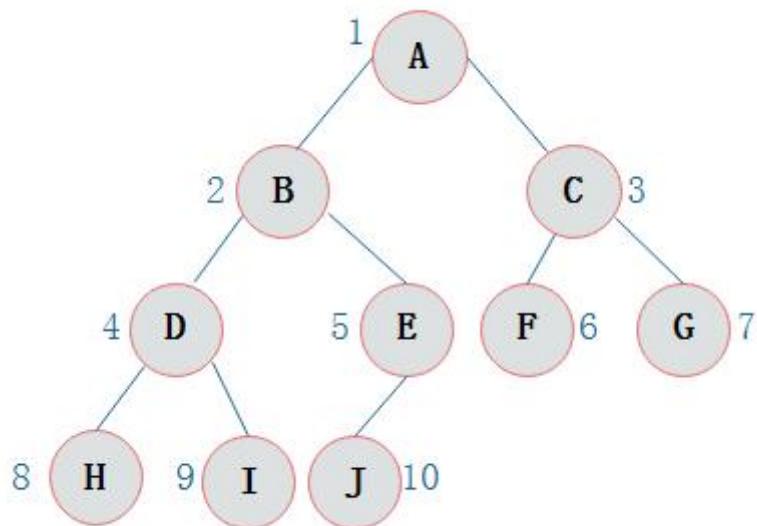
中序遍历

后序遍历

层序遍历

## 前序遍历

前序遍历通俗的说就是从二叉树的根结点出发, 当第一次到达结点时就输出结点数据, 按照先向左在向右的方向访问。



上图所示二叉树访问如下:

从根结点出发, 则第一次到达结点 A, 故输出 A;

继续向左访问, 第一次访问结点 B, 故输出 B;

按照同样规则, 输出 D, 输出 H;

当到达叶子结点 H, 返回到 D, 此时已经是第二次到达 D, 故不在输出 D, 进而向 D 右子树访问,

D 右子树不为空, 则访问至 I, 第一次到达 I, 则输出 I;

I 为叶子结点, 则返回到 D, D 左右子树已经访问完毕, 则返回到 B, 进而到 B 右子树, 第一次到达 E, 故输出 E;

向 E 左子树, 故输出 J;

按照同样的访问规则, 继续输出 C、F、G;

则上图所示二叉树的前序遍历输出为:

ABDHIEJCFG

## 中序遍历

中序遍历就是从二叉树的根结点出发, 当第二次到达结点时就输出结点数据, 按照先向左在向右的方向访问。

上图所示二叉树中序访问如下:

从根结点出发, 则第一次到达结点 A, 不输出 A, 继续向左访问, 第一次访问结点 B, 不输出 B; 继续到达 D, H;

到达 H, H 左子树为空, 则返回到 H, 此时第二次访问 H, 故输出 H;

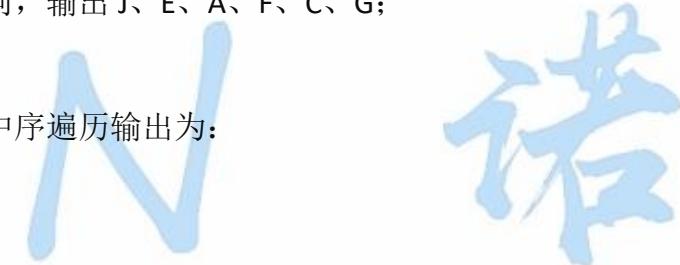
H 右子树为空, 则返回至 D, 此时第二次到达 D, 故输出 D;

由 D 返回至 B, 第二次到达 B, 故输出 B;

按照同样规则继续访问, 输出 J、E、A、F、C、G;

则上图所示二叉树的中序遍历输出为:

HDIBJEAFCG



## 后序遍历

noobdream.com

后序遍历就是从二叉树的根结点出发, 当第三次到达结点时就输出结点数据, 按照先向左在向右的方向访问。

上图所示二叉树后序访问如下:

从根结点出发, 则第一次到达结点 A, 不输出 A, 继续向左访问, 第一次访问结点 B, 不输出 B; 继续到达 D, H;

到达 H, H 左子树为空, 则返回到 H, 此时第二次访问 H, 不输出 H;

H 右子树为空, 则返回至 H, 此时第三次到达 H, 故输出 H;

由 H 返回至 D, 第二次到达 D, 不输出 D;

继续访问至 I, I 左右子树均为空, 故第三次访问 I 时, 输出 I;

返回至 D, 此时第三次到达 D, 故输出 D;

按照同样规则继续访问, 输出 J、E、B、F、G、C、A;

则上图所示二叉树的后序遍历输出为:

HIDJEBFGCA

## 层次遍历

层次遍历就是按照树的层次自上而下的遍历二叉树。

针对上图所示二叉树的层次遍历结果为:

ABCDEFGHIJ

## 遍历常考考点

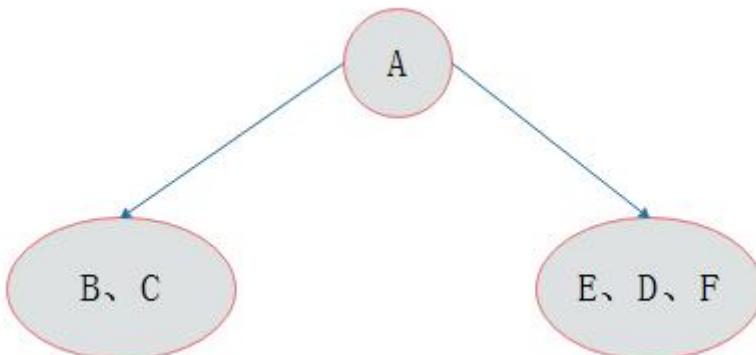
对于二叉树的遍历有一类典型题型。

1) 已知前序遍历序列和中序遍历序列, 确定一棵二叉树。

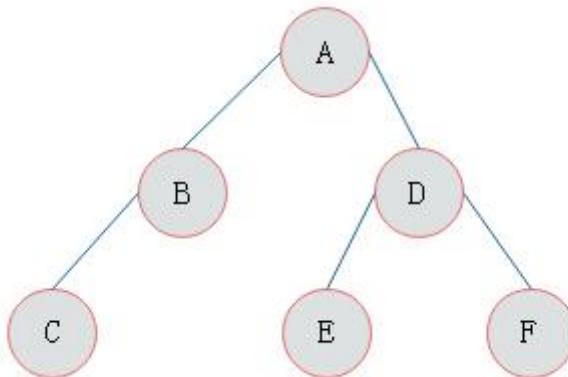
例题: 若一棵二叉树的前序遍历为 ABCDEF, 中序遍历为 CBAEDF, 请画出这棵二叉树。

分析: 前序遍历第一个输出结点为根结点, 故 A 为根结点。早中序遍历中根结点处于左右子树结点中间, 故结点 A 的左子树中结点有 CB, 右子树中结点有 EDF。

如图所示:



按照同样的分析方法, 对 A 的左右子树进行划分, 最后得出二叉树的形态如图所示:



2) 已知后序遍历序列和中序遍历序列, 确定一棵二叉树。

后序遍历中最后访问的为根结点, 因此可以按照上述同样的方法, 找到根结点后分成两棵子树, 进而继续找到子树的根结点, 一步步确定二叉树的形态。

注: 已知前序遍历序列和后序遍历序列, 不可以唯一确定一棵二叉树。

## 二叉树的建立和遍历

### 题目描述:

建立以二叉链作为存储结构的二叉树, 实现 1) 先序遍历; 2) 中序遍历; 3) 后序遍历; 4) 层序遍历; 5) 编程计算二叉树的叶子结点个数。

### 输入描述:

按照先序遍历序列输入二叉树中数据元素的值, 没有的输入 0 表示。

### 输出描述:

第一行输出先序遍历序列 第二行输出中序遍历序列 第三行输出后序遍历序列 第四行输出叶子结点的个数。

### 输入样例#:

A B C 0 0 0 D E 0 F 0 0 G 0 0

### 输出样例#:

A B C D E F G  
C B A E F D G  
C B F E G D A  
3

### 题目来源:

DreamJudge 1109

题目解析: 这是一道综合性的题目, 就是考察学生对二叉树的几种遍历方式的掌握情况。

参考代码 (完整模板)

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. typedef struct node{ //注意 typedef 不能省略
5.     char data;
6.     struct node *lchild,*rchild;
7. }*BitTree;
8. //先序遍历的方式创建二叉树
9. void CreatBitTree(BitTree &T) {
10.     char c;
11.     cin >> c;
12.     if (c == '0') T = NULL;
13.     else {
14.         T = new node;
15.         T->data = c;
16.         CreatBitTree(T->lchild);
17.         CreatBitTree(T->rchild);
18.     }
19. }
20. //将二叉树按照先序输出
21. void PreOrderTraverse(BitTree T) {
22.     if (T != NULL) {
23.         cout << T->data << ' ';
24.         PreOrderTraverse(T->lchild);
25.         PreOrderTraverse(T->rchild);
26.     }
27. }
28. //将二叉树按照中序输出
29. void InOrderTraverse(BitTree T) {
30.     if (T != NULL) {
31.         InOrderTraverse(T->lchild);
32.         cout << T->data << ' ';
33.         InOrderTraverse(T->rchild);
34.     }
}
```

```
35. }
36. //将二叉树按照后序输出
37. void PostOrderTraverse(BitTree T) {
38.     if (T != NULL) {
39.         PostOrderTraverse(T->lchild);
40.         PostOrderTraverse(T->rchild);
41.         cout << T->data << ' ';
42.     }
43. }
44. //二叉树的叶子节点个数
45. int Leaf(BitTree T) {
46.     if (T == NULL) return 0;
47.     if (T->lchild == NULL && T->rchild == NULL) return 1;
48.     return Leaf(T->lchild) + Leaf(T->rchild);
49. }
50. //二叉树的深度
51. int Depth(BitTree T) {
52.     if (T == NULL) return 0;
53.     int x = Depth(T->lchild);
54.     int y = Depth(T->rchild);
55.     return max(x,y) + 1;
56. }
57. int main(){
58.     BitTree T;
59.     CreatBitTree(T);
60.     PreOrderTraverse(T); cout << endl;
61.     InOrderTraverse(T); cout << endl;
62.     PostOrderTraverse(T); cout << endl;
63.     cout << Leaf(T) << endl;
64.     return 0;
65. }
```

## 练习题目

DreamJudge 1161 二叉树遍历

DreamJudge 1233 二叉树

DreamJudge 1264 二叉树 2

DreamJudge 1401 二叉树遍历

DreamJudge 1551 判断二叉树是否对称

DreamJudge 1561 二叉树 (北京邮电大学)

## 5.4 二叉排序树

### 二叉排序树有两种考法

- 1、考察定义的理解, 根据定义建立二叉排序树, 然后输出其先序、中序、后序。
- 2、考察二叉排序树的应用, 例如多次查找, 建立二叉排序树之后将查找的复杂度从线性降低到  $\log$  级别。对于可以使用 C++ 的同学而言, 直接用前面讲过的 map 即可, 对于只能使用 C 语言的同学而言, 掌握二叉排序可以解决大量的查找类问题。

二叉排序树 (Binary Sort Tree), 又称二叉查找树 (Binary Search Tree), 亦称二叉搜索树。

### 定义

一棵空树, 或者是具有下列性质的二叉树:

- (1) 若左子树不空, 则左子树上所有结点的值均小于它的根结点的值;
- (2) 若右子树不空, 则右子树上所有结点的值均大于它的根结点的值;
- (3) 左、右子树也分别为二叉排序树;
- (4) 没有键值相等的结点。

### 二叉排序树 2

#### 题目描述:

输入一系列整数, 建立二叉排序树, 并进行前序, 中序, 后序遍历。

#### 输入描述:

输入第一行包括一个整数  $n$  ( $1 \leq n \leq 100$ )。

接下来的一行包括  $n$  个整数。

#### 输出描述:

可能有多组测试数据, 对于每组数据, 将题目所给数据建立一个二叉排序树, 并对二叉排序树进行前序、中序和后序遍历。

每种遍历结果输出一行。每行最后一个数据之后有一个空格。

输入中可能有重复元素, 但是输出的二叉树遍历序列中重复元素不用输出。

#### 输入样例#:

5

1 6 5 9 8

输出样例#:

1 6 5 9 8

1 5 6 8 9

5 8 9 6 1

题目来源:

DreamJudge 1411

题目解析: 根据二叉排序的定义将所有元素插入到二叉排序树中, 然后分别输出这颗二叉排序树的先序遍历、中序遍历、后序遍历。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. typedef struct node{
5.     int data;
6.     struct node *lchild,*rchild;
7. }*BitTree;
8. //将元素插入二叉排序树中
9. void InsertBitTree(BitTree &T, int x) {
10.     if (T == NULL) {
11.         T = new node;
12.         T->data = x;
13.         T->lchild = NULL;
14.         T->rchild = NULL;
15.         return;
16.     }
17.     if (x == T->data) return;
18.     else if (x < T->data) InsertBitTree(T->lchild, x);
19.     else InsertBitTree(T->rchild, x);
20. }
21. //将二叉树按照先序输出
22. void PreOrderTraverse(BitTree T) {
```

```
23.     if (T != NULL) {
24.         cout << T->data << ' ';
25.         PreOrderTraverse(T->lchild);
26.         PreOrderTraverse(T->rchild);
27.     }
28. }
29. //将二叉树按照中序输出
30. void InOrderTraverse(BitTree T) {
31.     if (T != NULL) {
32.         InOrderTraverse(T->lchild);
33.         cout << T->data << ' ';
34.         InOrderTraverse(T->rchild);
35.     }
36. }
37. //将二叉树按照后序输出
38. void PostOrderTraverse(BitTree T) {
39.     if (T != NULL) {
40.         PostOrderTraverse(T->lchild);
41.         PostOrderTraverse(T->rchild);
42.         cout << T->data << ' ';
43.     }
44. }
45. int main(){
46.     int n, x;
47.     while (cin >> n) {
48.         BitTree T = NULL;
49.         for (int i = 1; i <= n; i++) {
50.             cin >> x;
51.             InsertBitTree(T, x);
52.         }
53.         PreOrderTraverse(T); cout << endl;
54.         InOrderTraverse(T); cout << endl;
55.         PostOrderTraverse(T); cout << endl;
56.     }
57.     return 0;
58. }
```

## 练习题目

DreamJudge 1317 二叉搜索树

DreamJudge 1396 二叉排序树 - 华科

## 5.5 hash 算法

哈希算法在考研机试中的题目几乎都可以用 `map` 来解决。

哈希的几种考法如下

1、输入  $N$  个数, 统计某个数出现的次数。

解: 使用辅助数组进行标记

2、输入  $N$  个数, 进行排序或求前  $K$  小的数, 其中数值的区间范围小。

解: 使用辅助数组进行标记, 如果值为负数或很大, 将区间进行平移即可

3、有多段区间进行覆盖, 问其中某个点被覆盖到的次数。

解: 使用辅助数组进行标记, 直接遍历每一段区间累加上去。

上面这几种勉强可列为哈希算法的范围, 其实都是对数组的一种应用技巧, 使用数组下标对应存储的值, 数组存的值是出现的次数。

4、给一个等式  $a+b+c+d=0$ , 其中  $a,b,c,d$  的范围是  $[-50,50]$ , 求  $a,b,c,d$  使得等式成立的值。

解: 如果直接暴力枚举, 那么复杂度是  $O(n^4)$ , 我们可以将等式移项变成  $a+b=-(c+d)$ , 我们分别枚举  $a+b$  的值存起来,  $-(c+d)$  的值存起来, 复杂度是  $O(n^2)$ , 那么问题就转化成了, 比较两个数组中相同的值的个数。可以用二分查找的方法, 也可以用 `map`, 还可以自己构造哈希函数, 最终的时间复杂度是  $O(n^2 \log(n^2))$ 。

5、输入  $n$  个字符串, 问相同的字符串的个数

解: 很典型的字符串哈希, 建议用 `map`, 基本都能解决。

### 练习题目

DreamJudge 1329 统计同成绩的学生人数

DreamJudge 1225 谁是你潜在朋友

DreamJudge 1175 剩下的树

DreamJudge 1209 刷出一道墙

## 5.6 前缀树

### 定义

又称单词查找树, Trie 树, 是一种树形结构, 是一种哈希树的变种。典型应用是用于统计, 排序和保存大量的字符串(但不仅限于字符串), 所以经常被搜索引擎系统用于文本词频统计。它的优点是: 利用字符串的公共前缀来减少查询时间, 最大限度地减少无谓的字符串比较, 查询效率比哈希树高。

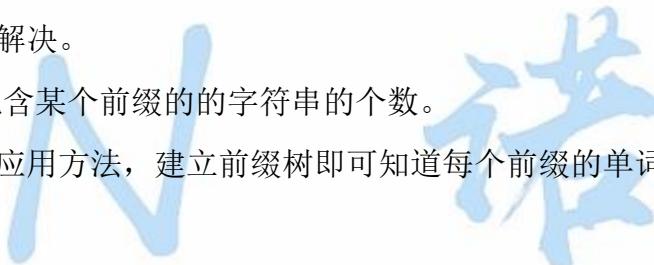
### 考点分析

1、 $n$  个字符串中找到某个字符串是否存在或出现了几次。

解法: 数据小直接顺序查找比较, 数据大的时候可以直接用 map 来查找, 如果只能用 C 语言, 那么就需要用前缀树来解决。

2、 $n$  个字符串中查找包含某个前缀的的字符串的个数。

解法: 这是非常典型的应用方法, 建立前缀树即可知道每个前缀的单词个数。



### 前缀字符串

#### 题目描述:

如果一个字符串  $s_1$  是由另一个字符串  $s_2$  的前面部分连续字符组成的, 那么我们就说  $s_1$  就是  $s_2$  的前缀。比如“ac”是“acm”的前缀, “abcd”是“abcdfasf”的前缀, 特别的“kdfa”是“kdfa”的前缀。现在给你一些字符串, 你的任务就是从这些字符串中找出一些字符串放到一个集合中, 使得这个集合中任意一个字符串不是其他字符串的前缀, 并且要使集合里的字符串尽可能的多。输出这个集合中字符串的个数。

#### 输入描述:

有多组测试数据。每组测试数据以一个整数  $n$  开头, 随后有  $n$  个字符串。当  $n=0$  时表示输入结束。

$0 < n < 100$ , 字符串长度不大于 20。

#### 输出描述:

每组测试数据输出一个整数, 即所求的最大值。每组数据占一行。

输入样例#:

```
6
acm
yuou
yuoufsdaf
acmmmdf
acmm
fdhf
0
```

输出样例#:

```
3
```

题目来源:

DreamJudge 1098

**题目解析:** 这道题的数据很小, 所以各种办法都可以解决, 如果题目的数据大一些, 比如有 10W 个字符串的时候, 这个时候就需要用前缀树来解决了, 其实这个题就是求前缀树的叶子结点的个数, 因为只有是叶子结点才不会是其他字符串的前缀。

noobdream.com

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 26; //26 个字母, 如果包含大小写的话改成 52
5. typedef struct TrieNode {
6.     int nCount; //结点出现次数
7.     struct TrieNode *next[maxn];
8. }Trie;
9. Trie root;
10. //初始化
11. void InitTrie() {
12.     for (int i = 0; i < maxn; i++)
```

```
13.         root->next[i] = NULL;
14.     }
15. //创建 Trie 树
16. void CreateTrie(char *str) {
17.     int len = strlen(str);
18.     Trie *p = &root, *q;
19.     for (int i = 0; i < len; i++) {
20.         int k = str[i] - 'a';
21.         if (p->next[k] == NULL) {
22.             q = (Trie *)malloc(sizeof(root));
23.             q->nCount = 1;
24.             for (int j = 0; j < maxn; j++)
25.                 q->next[j] = NULL;
26.             p->next[k] = q;
27.             p = p->next[k];
28.         }
29.         else {
30.             p->next[k]->nCount++;
31.             p = p->next[k];
32.         }
33.     }
34. }
35. //统计 Trie 树中叶子结点的个数
36. int sum;
37. void CountTrie() {
38.     queue<Trie*> q;
39.     q.push(&root);
40.     while (!q.empty()) {
41.         Trie *p = q.front();
42.         q.pop();
43.         int son = 0;
44.         for (int i = 0; i < maxn; i++) {
45.             int k = i;
46.             if (p->next[k] == NULL) son++;
47.             else {
48.                 q.push(p->next[k]);
49.             }
50.         }
51.         if (son == maxn) sum += 1;
52.     }
53. }
54.
55. int main() {
```

```
56.     int n;
57.     while (cin >> n) {
58.         if (n == 0) break;
59.         char str[20];
60.         sum = 0;
61.         InitTrie();
62.         for (int i = 0; i < n; i++) {
63.             scanf("%s", str);
64.             CreateTrie(str);
65.         }
66.         CountTrie();
67.         printf("%d\n", sum);
68.     }
69.     return 0;
70. }
```

下面给出两个通用模板, 同学们做题的时候可以直接套上去使用, 一个是链式存储的方法, 另一个是静态数组的方式实现, 同学们根据自己的喜好选择使用即可。

## 链式存储

noobdream.com

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 26;
5. typedef struct TrieNode {
6.     int nCount;
7.     struct TrieNode *next[maxn];
8. }Trie;
9. Trie root;
10. void InitTrie() {
11.     for (int i = 0; i < maxn; i++)
12.         root.next[i] = NULL;
13. }
14. void CreateTrie(char *str) {
15.     int len = strlen(str);
16.     Trie *p = &root, *q;
```

```
17.     for (int i = 0; i < len; i++) {
18.         int k = str[i] - 'a';
19.         if (p->next[k] == NULL) {
20.             q = (Trie *)malloc(sizeof(root));
21.             q->nCount = 1;
22.             for (int j = 0; j < maxn; j++)
23.                 q->next[j] = NULL;
24.             p->next[k] = q;
25.             p = p->next[k];
26.         }
27.     else {
28.         p->next[k]->nCount++;
29.         p = p->next[k];
30.     }
31. }
32. }
33. int FindTrie(char *str) {
34.     int len = strlen(str);
35.     Trie *p = &root;
36.     for (int i = 0; i < len; i++) {
37.         int k = str[i] - 'a';
38.         if (p->next[k] == NULL) return 0;
39.         p = p->next[k];
40.     }
41.     return p->nCount;
42. }
43. int main() {
44.     char str[15];
45.     InitTrie();
46.     while (gets(str) && str[0]) CreateTrie(str);
47.     while (gets(str)) printf("%d\n", FindTrie(str));
48.     return 0;
49. }
```

## 静态数组

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 1000000 + 10;
```

```
5. int trie[maxn][26] = {0}; // 存储下一个字符的位置
6. int num[maxn] = {0};
7. int pos = 1;
8. void InsertTrie(char word[]) {
9.     int c = 0;
10.    for (int i = 0; word[i]; i++) {
11.        int k = word[i] - 'a';
12.        if (trie[c][k] == 0) trie[c][k] = pos++;
13.        c = trie[c][k];
14.        num[c]++;
15.    }
16. }
17. int FindTrie(char word[]) {
18.     int c = 0;
19.     for (int i = 0; word[i]; i++) {
20.         int k = word[i] - 'a';
21.         if (trie[c][k] == 0) return 0;
22.         c = trie[c][k];
23.     }
24.     return num[c];
25. }
26. int main() {
27.     char str[15] = {0};
28.     while (gets(str) && str[0]) InsertTrie(str);
29.     while (gets(str)) printf("%d\n", FindTrie(str));
30.     return 0;
31. }
```

## 练习题目

DreamJudge 1492 三叉树

DreamJudge 1654 二叉树

DreamJudge 1827 有向树形态

DreamJudge 1841 树的高度

## 第六章 搜索

本章我们重点讲解一些常见的搜索题型, 包括暴力枚举、广度优先搜索 (BFS) 、递归及其应用、深度优先搜索 (DFS) 、搜索剪枝技巧、终极骗分技巧等内容。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搜索问题。



本书配套视频精讲: <https://www.bilibili.com/video/av81203473>

## 6.1 暴力枚举

### 定义

枚举算法是在分析问题时, 逐个列举出所有可能情况, 然后根据条件判断此答案是否合适, 合适就保留, 不合适就丢弃, 最后得出一个结论。主要利用计算机运算速度快、精确度高的特点, 对要解决问题的所有可能情况, 一个不漏地进行检验, 从中找出符合要求的答案, 因此枚举法是通过牺牲时间来换取答案的全面性。

### 举例说明

有一个整数  $ABCD$ , 一定是四位数,  $A$  不能为 0, 其中  $ABCD \times 4 = DCBA$ 。

其中  $A$ 、 $B$ 、 $C$ 、 $D$  都是一个数字, 求  $ABCD$  是多少?

如何解决这个问题呢?

我们可以直接枚举  $ABCD$  四个数的值



代码如下

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int main() {
5.     for (int A = 0; A <= 9; A++) {
6.         for (int B = 0; B <= 9; B++) {
7.             for (int C = 0; C <= 9; C++) {
8.                 for (int D = 0; D <= 9; D++) {
9.                     if (A == 0) continue;
10.                    int s1 = A * 1000 + B * 100 + C * 10 + D;
11.                    int s2 = D * 1000 + C * 100 + B * 10 + A;
12.                    if (s1 * 4 == s2) printf("%d\n", s1);
13.                }
14.            }
15.        }
16.    }
17.    return 0;
18. }
```

经过计算, 答案为: 2178

这种方法就是枚举, 优点是直接了当, 缺点是复杂度高。

我们可以做一个最简单优化, 看最高位可知,  $D >= A * 4$ , 因此  $A$  只能取值 1 和 2。

### 练习题目

DreamJudge 1348 百鸡问题

DreamJudge 1165 abc

DreamJudge 1274 Old Bill



## 6.2 广度优先搜索 (BFS)

广度优先搜索其实简单的理解就是从起点开始一层一层的扩散出去, 要实现这个一层一层的扩散就要用到队列的先进先出的思想, 所以一般我们都用队列来实现广度优先搜索算法。

### 迷宫

#### 题目描述:

小 A 同学现在被困在了一个迷宫里面, 他很想从迷宫中走出来, 他可以 向上、向下、向左、向右移动、每移动一格都需要花费 1 秒的时间, 不能够走到 边界之外。假设小 A 现在的位置在 S, 迷宫的出口在 E, 迷宫可能有多个出口。问小 A 想要走到迷宫出口最少需要花费多少秒?

#### 输入描述:

有多组测试数据。

第一行输入两个正整数 H ( $0 < H \leq 100$ ) 和 W ( $0 < W \leq 100$ ), 分别表示迷宫的高和宽。

接下来 H 行, 每行 W 个字符 (其中 '\*' 表示路, '#' 表示墙, 'S' 表示小 A 的位置, 'E' 表示迷宫出口)。

当 H 与 W 都等于 0 时程序结束。

#### 输出描述:

输出小 A 走到迷宫出口最少需要花费多少秒, 如果永远无法走到出口则输出 “-1”。

#### 输入样例#:

```
3 3
S**#
**#
#*E
0 0
```

#### 输出样例#:

```
4
```

题目来源:

DreamJudge 1563

题目解析: 直接使用广度优先搜索模板即可。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 100 + 5;
5. char mpt[maxn][maxn];
6. int vis[maxn][maxn];
7. int dir[4][2] = {0, 1, 1, 0, 0, -1, -1, 0};
8. struct node {
9.     int x, y;
10.    int step;
11. };
12. //使用广度优先搜索求解
13. int bfs(int sx, int sy) {
14.     memset(vis, 0, sizeof(vis));
15.     queue<node> q; //使用队列来维护一层层发散的优先级
16.     q.push(node{sx, sy, 0});
17.     vis[sx][sy] = 1;
18.     int ans = -1;
19.     while(!q.empty()) {
20.         node now = q.front();
21.         q.pop();
22.         if (mpt[now.x][now.y] == 'E') { //找到终点
23.             ans = now.step;
24.             break;
25.         }
26.         for (int i = 0; i < 4; i++) { //上下左右四个方向
27.             int nx = now.x + dir[i][0];
28.             int ny = now.y + dir[i][1];
29.             if ((mpt[nx][ny] == '*' || mpt[nx][ny] == 'E') && vis[nx][ny] == 0) {
30.                 q.push(node{nx, ny, now.step + 1});
31.                 vis[nx][ny] = 1;
32.             }
33.         }
34.     }
35. }
```

```
34.     }
35.     return ans;
36. }
37. int main() {
38.     int h, w;
39.     while (scanf("%d%d", &h, &w) != EOF) {
40.         if (h == 0 && w == 0) break;
41.         int sx = 0, sy = 0;
42.         memset(mpt, 0, sizeof(mpt));
43.         for (int i = 1; i <= h; i++) {
44.             scanf("%s", mpt[i] + 1);
45.             for (int j = 1; j <= w; j++) {
46.                 if (mpt[i][j] == 'S') {
47.                     sx = i, sy = j;//记录起点坐标
48.                 }
49.             }
50.         }
51.         int ans = bfs(sx, sy);
52.         printf("%d\n", ans);
53.     }
54.     return 0;
55. }
```

noobdream.com

## 练习题目

DreamJudge 1308 迷宫逃离 2

DreamJudge 1124 生化武器 2

DreamJudge 1126 生化武器

## 6.3 递归及其应用

### 定义

递归算法（英语：recursion algorithm）在计算机科学中是指一种通过重复将问题分解为同类的子问题而解决问题的方法。递归式方法可以被用于解决很多的计算机科学问题，因此它是计算机科学中十分重要的一个概念。绝大多数编程语言支持函数的自调用，在这些语言中函数可以通过调用自身来进行递归。计算理论可以证明递归的作用可以完全取代循环，因此在很多函数编程语言（如 Scheme）中习惯用递归来实现循环。

例：求 N！

递归实现代码如下

```
1. #include <stdio.h>
2.
3. int fac(int x) {
4.     if (x == 0) return 1;
5.     return x * fac(x - 1);
6. }
7. int main() {
8.     int n;
9.     scanf("%d", &n);
10.    printf("%d\n", fac(n));
11.    return 0;
12. }
```



### Hanoi 塔问题

#### 题目描述：

(n 阶 Hanoi 塔问题) 假设有三个分别命名为 A、B、C 的塔座，在塔座 A 上插有 n ( $n < 20$ ) 个直径大小各不相同、依小到大编号为 1, 2, ..., n 的圆盘。现要求将 A 轴上的 n 个圆盘移至塔座 C 上并仍按同样顺序叠排，圆盘移动时必须遵循下列规则： 1) 每次只能移动一个圆盘； 2) 圆盘可以插在 A、B、C 中的任一塔座上； 3) 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。 请通过编程来打印出移动的步骤。

### 输入描述:

只有一组输入数据. 输入数据  $N$  (; 表示在开始时 A 塔座上的盘子数), 当输入 0 时程序结束.

### 输出描述:

输出移动的步骤. 如"A-->C", "A-->B"等. 每两的步骤之间有三个空格隔开, 每输出 5 个步骤就换行. 详细的见 Sample Output.

### 输入样例#:

```
5
2
0
```

### 输出样例#:

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

### 题目来源:

DreamJudge 1082

**题目解析:** 汉诺塔问题是一个很经典的问题了, 找到移动过程中的共同点, 然后通过递归的方式进行求解。

### 参考代码

```
1. #include<iostream>
2. using namespace std;
3.
4. int step;//移动步数
```

```
5. void Hanoi(int n, char a, char b, char c) {
6.     if(n == 1) {
7.         if((step+1) % 5 == 0)
8.             cout << a << "-->" << c << endl;
9.         else cout << a << "-->" << c << "    ";
10.        step++;
11.        return;
12.    }
13.    Hanoi(n-1, a, c, b);
14.    Hanoi(1, a, b, c);
15.    Hanoi(n-1, b, a, c);
16. }
17. int main() {
18.     int n;
19.     while(cin>>n)
20.     {
21.         if(n == 0) break;
22.         step = 0;
23.         Hanoi(n, 'A', 'B', 'C'); cout << endl;
24.     }
25.     return 0;
26. }
```

noobdream.com

### 练习题目

DreamJudge 1185 全排列

## 6.4 深度优先搜索 (DFS)

深度优先搜索的实现形式就是上一节的递归, 如果把递归的流程画出来, 那么我们可以得到一颗递归树, 其实就是一个多叉树。

简单来说, **BFS** 和 **DFS** 到底有什么区别呢?

对于一棵二叉树,

BFS 就是二叉树的层次遍历, 一层一层的扩展下去。

DFS 就是二叉树的中序遍历, 一条路走到底, 然后回溯走第二条, 直到所有路都走完。

大家需要注意的是, DFS 一般情况下效率不如 BFS, 比如求 DFS 中的迷宫的最短路径使用 DFS 就会超时。

### 迷宫

#### 题目描述:

小 A 同学现在被困在了一个迷宫里面, 他很想从迷宫中走出来, 他可以 向上、向下、向左、向右移动、每移动一格都需要花费 1 秒的时间, 不能够走到 边界之外。假设小 A 现在的位置在 S, 迷宫的出口在 E, 迷宫可能有多个出口。问小 A 想要走到迷宫出口最少需要花费多少秒?

#### 输入描述:

有多组测试数据。

第一行输入两个正整数 H ( $0 < H \leq 100$ ) 和 W ( $0 < W \leq 100$ ), 分别表示迷宫的高和宽。

接下来 H 行, 每行 W 个字符 (其中 '\*' 表示路, '#' 表示墙, 'S' 表示小 A 的位置, 'E' 表示迷宫出口)。

当 H 与 W 都等于 0 时程序结束。

#### 输出描述:

输出小 A 走到迷宫出口最少需要花费多少秒, 如果永远无法走到出口则输出 “-1”。

#### 输入样例#:

3 3

S\*#

\*\*#

#\*E

0 0

输出样例#:

4

题目来源:

DreamJudge 1563

题目解析: 直接使用深度优先搜索 (DFS) 模板。

超时代码 (直接用 DFS)

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 100 + 5;
5. char mpt[maxn][maxn];
6. int vis[maxn][maxn];
7. int dir[4][2] = {1, 0, 0, -1, -1, 0, 0, 1};
8. int ans;
9. //使用深度优先搜索求解
10. void dfs(int x, int y, int step) {
11.     if (step >= ans) return; //一个小剪枝, 当步数超过答案就不用继续
12.     if (mpt[x][y] == 'E') { //找到终点
13.         ans = min(ans, step);
14.         return;
15.     }
16.     for (int i = 0; i < 4; i++) { //上下左右四个方向
17.         int nx = x + dir[i][0];
18.         int ny = y + dir[i][1];
19.         if ((mpt[nx][ny] == '*' || mpt[nx][ny] == 'E') && vis[nx][ny] == 0) {
20.             vis[nx][ny] = 1;
21.             dfs(nx, ny, step+1);
22.             vis[nx][ny] = 0;
23.         }
24.     }
25. }
26. int main() {
```

```
27.     int h, w;
28.     while (scanf("%d%d", &h, &w) != EOF) {
29.         if (h == 0 && w == 0) break;
30.         int sx = 0, sy = 0;
31.         memset(mpt, 0, sizeof(mpt));
32.         memset(vis, 0, sizeof(vis));
33.         for (int i = 1; i <= h; i++) {
34.             scanf("%s", mpt[i] + 1);
35.             for (int j = 1; j <= w; j++) {
36.                 if (mpt[i][j] == 'S') {
37.                     sx = i, sy = j;//记录起点坐标
38.                 }
39.             }
40.         }
41.         ans = 99999999;
42.         dfs(sx, sy, 0);
43.         printf("%d\n", ans);
44.     }
45.     return 0;
46. }
```

注: 当题目数据范围小的时候可以直接这样 DFS 搜索, 但是数据大的时候容易超出时间限制。

## 石油储藏

### 题目描述:

有一个 GeoSurvComp 地质勘探公司正在负责探测地底下的石油块。这个公司在一个时刻调查一个矩形区域, 并且创建成一个个的格子用来表示众多正方形块。它然后使用测定设备单个的分析每块区域, 决定是否有石油。一块有石油小区域被称为一个 pocket, 假设两个 pocket 是相邻的, 然后他们就是相同石油块的一部分, 石油块可能非常的大并且包涵很多的 pocket。你的任务就是在一个网格中存在多少个石油块。输入首先给出图的大小, 然后给出这个图。\*代表没有石油, @代表存在石油。输出每种情况下石油块的个数。

### 输入描述:

输入包含一个或多个网格。 每个网格都以包含 m 和 n 的行开始, 网格中的行和列数为 m 和 n, 以单个空格分隔。 如果 m = 0, 则表示输入结束。 否则为  $1 \leq m \leq 100$  和  $1 \leq n \leq 100$ 。 这之后是 m 行, 每行 n 个字符 (不计算行末字符)。 每个字符对应一个情节, 要么是代表没

有油的“\*”，要么是代表油囊的“@”。

#### 输出描述:

在水平, 垂直或对角线上都算作相邻, 输出每种情况下石油块的个数。

#### 输入样例#:

```
5 5
*****@
*@@*@
*@@*@
@@@*@
@@*@
0 0
```

#### 输出样例#:

```
2
```

#### 题目来源:

DreamJudge 1564

题目解析: 其实就是搜索有多少个@块, 有8个方向, 我们直接使用DFS搜索即可。

参考代码

noobdream.com

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 100 + 5;
5. char mpt[maxn][maxn];
6. int vis[maxn][maxn];
7. int dir[8][2] = {1,0,0,-1,-1,0,0,1,1,1,-1,-1,1,-1,-1};
8. //使用深度优先搜索求解
9. int dfs(int x, int y) {
10.     vis[x][y] = 1;
11.     for (int i = 0; i < 8; i++) { //8个方向
12.         int nx = x + dir[i][0];
13.         int ny = y + dir[i][1];
14.         if (mpt[nx][ny] == '@' && vis[nx][ny] == 0) {
15.             dfs(nx, ny);
```

```
16.     }
17. }
18. }
19. int main() {
20.     int h, w;
21.     while (scanf("%d%d", &h, &w) != EOF) {
22.         if (h == 0 && w == 0) break;
23.         int sx = 0, sy = 0;
24.         memset(mpt, 0, sizeof(mpt));
25.         memset(vis, 0, sizeof(vis));
26.         for (int i = 1; i <= h; i++) {
27.             scanf("%s", mpt[i] + 1);
28.         }
29.         int ans = 0;
30.         for (int i = 1; i <= h; i++) {
31.             for (int j = 1; j <= w; j++) {
32.                 if (vis[i][j] == 0 && mpt[i][j] == '@') {
33.                     ans++;
34.                     dfs(i, j);
35.                 }
36.             }
37.         }
38.         printf("%d\n", ans);
39.     }
40.     return 0;
41. }
```



## 6.5 搜索剪枝技巧

### 什么是剪枝?

常用的搜索有 DFS 和 BFS。

BFS 的剪枝通常就是判重, 因为一般 BFS 寻找的是步数最少, 重复的话必定不会在之前的情况下产生最优解。

深搜, 它的进程近似一颗树(通常叫 DFS 树)。

而剪枝就是一种生动的比喻: 把不会产生答案的, 或不必要的枝条“剪掉”。

剪枝的关键就在于剪枝的判断: 什么枝该剪, 什么枝不该剪, 在什么地方剪。

### 剪枝的原则?

正确性, 准确性, 高效性。

常用的剪枝有: 可行性剪枝、最优性剪枝、记忆化搜索、搜索顺序剪枝。

#### 1. 可行性剪枝

如果当前条件不合法就不再继续搜索, 直接 `return`。这是非常好理解的剪枝, 搜索初学者都能轻松地掌握, 而且也很好想。一般的搜索都会加上。

一般格式:

```
1. dfs(int x)
2. {
3.     if(x>n) return;
4.     if(!check1(x)) return;
5.     ....
6.     return;
7. }
```

#### 2. 最优性剪枝

如果当前条件所创造出的答案必定比之前的答案大, 那么剩下的搜索就毫无必要, 甚至可以剪

掉。

我们利用某个函数估计出此时条件下答案的‘下界’，将它与已经推出的答案相比，如果不比当前答案小，就可以剪掉。

一般格式：

```
1. long long ans=98747447743448711;
2. ... Dfs(int x,...)
3. {
4.     if(x... && ...){ans=....;return ...;}
5.     if(check2(x)>=ans) return ...; //最优性剪枝
6.     for(int i=1;...;++i)
7.     {
8.         vis[...]=1;
9.         dfs(...);
10.        vis[...]=0;
11.    }
12. }
13. //一般实现：在搜索取和最大值时，如果后面的全部取最大仍然不比当前答案大就可以返回。
14. //在搜和最小时同理，可以预处理后缀最大/最小和进行快速查询。
```

### 3. 记忆化搜索

记忆化搜索其实很像动态规划（DP）。

它的关键是：如果对于相同情况下必定答案相同，就可以把这个情况的答案值存储下来，以后再次搜索到这种情况时就可以直接调用。

还有就是不能搜出环来，不能互相依赖。

一般格式：

```
1. long long ans=98747447743448711;
2. ... Dfs(int x,...)
3. {
4.     if(x... && ...){ans=....;return ...;}
5.     if(vis[x]!=0) return f[x];vis[x]=1;
6.     for(int i=1;...;++i)
7.     {
8.         vis[...]=1;
9.         dfs(...);
```

```
10.     vis[...]=0;
11.     f[x]=...;
12. }
13. }
```

#### 4. 搜索顺序剪枝

在一些迷宫题, 网格题, 或者其他搜索中可以贪心的题, 搜索顺序显得十分重要。我们经常听见有人说(我自己也说过): “从左边搜会 T, 从右边搜就 A 了” 之类的语句。

其实在迷宫、网格类的题目中, 以左上->右下为例, 右下左上就明显比左上右下优秀。

在一些推断搜索题中, 从已知信息最多的地方开始搜索显然更加优秀。

在一些题中, 先搜某个值大的, 再搜某个值小的(比如树的度数, 产生答案的预计(A\*), 速度明显会比乱搜更快。

搜索的复杂度明显讲不清, 这种剪枝自然是能加就加。



## 6.6 终极骗分技巧

有的时候题目的正解并不是搜索, 由于数据范围大等原因, 我们搜索不管怎么剪枝都会超时怎么办? 坐以待毙吗?

答案当然是不!

当你想不到正解只会搜索的时候, 千万不要气馁, 这一章我们教会大家用科学解析那些玄学骗分技巧, 那句话怎么说来着, 没有挖不走的墙角, 只有.....

### 1、利用预处理的思想

这是一种中规中矩的处理方式, 先将搜索过程中可能会用到的值预先处理出来, 本质是用空间换时间。例如 A\* (启发式搜索) 就是用的这个思想, 这样可以大大节约搜索时的时间开销, 也能保证答案的正确性。



### 2、暴力剪枝

这种方法就好像一棵树上结了一个果子, 你不再是每一个树枝都去寻找, 而是直接砍掉一些树枝不要, 然后再在剩下的树上寻找, 很明显这种方法不能保证一定正确, 但是却可以大大减少时间的消耗, 也可以变成递归到多少层就不继续往下递归了, 不管是否后面有没有果子。

### 3、利用贪心的思想

贪心是一种很简单的思想, 虽然贪心不一定是正确, 但是却有可能正确。

### 4、利用概率论的思想

我们都知道现在图像识别已经进入了大家的生活, 但是识别不是都能 100% 的准确, 现在看各个厂家的产品都是说识别率能达到百分之多少, 而不是 100%。

那么我们可以根据对应的题目去解析几个特征值, 然后设置一下概率来触发搜索的条件。

后面三种的使用效果要看各人的天赋, 毕竟是绝招嘛, 一般情况下不会使用的。

本质上都是利用了考研的机试题目一般都不会特别强的弱点。

## 第七章 图论

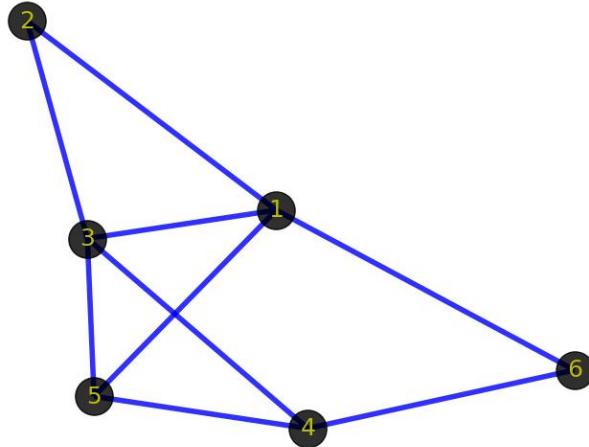
本章我们重点讲解一些常见的图论题型, 包括图的理论基础、图的存储、并查集、最小生成树问题、最短路径问题、拓扑排序等内容。希望能帮助读者更好的掌握计算机考研机试中所涉及到的图论问题。



本书配套视频精讲: <https://www.bilibili.com/video/av81203473>

## 7.1 理论基础

对于大部分图论问题, 直接套算法模板即可解决。



### 一、顶点 (vertex)

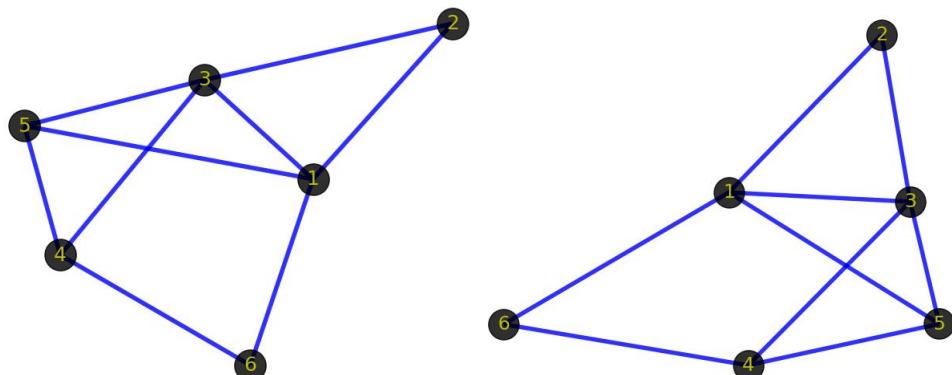
上图中黑色的带数字的点就是顶点, 表示某个事物或对象。由于图的术语没有标准化, 因此, 称顶点为点、节点、结点、端点等都是可以的。叫什么无所谓, 理解是什么才是关键。

### 二、边 (edge)

上图中顶点之间蓝色的线条就是边, 表示事物与事物之间的关系。需要注意的是边表示的是顶点之间的逻辑关系, 粗细长短都无所谓。包括上面的顶点也一样, 表示逻辑事物或对象, 画的时候大小形状都无所谓。

### 三、同构 (Isomorphism )

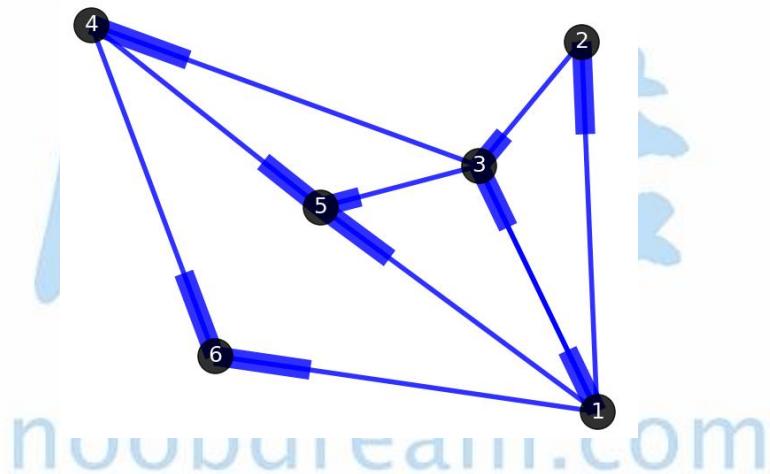
先看看下面 2 张图:



首先你的感觉是这 2 个图肯定不一样。但从图 (graph) 的角度出发, 这 2 个图是一样的, 即它们是同构的。前面提到顶点和边指的是事物和事物的逻辑关系, 不管顶点的位置在哪, 边的粗细长短如何, 只要不改变顶点代表的事物本身, 不改变顶点之间的逻辑关系, 那么就代表这些图拥有相同的信息, 是同一个图。同构的图区别仅在于画法不同。

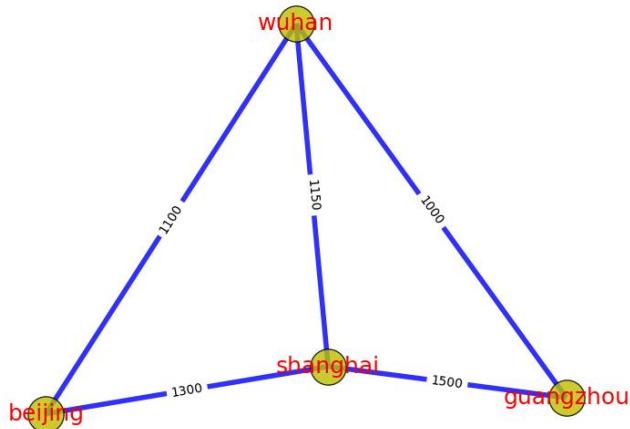
#### 四、有向/无向图 (Directed Graph/ Undirected Graph)

最基本的图通常被定义为“无向图”, 与之对应的则被称为“有向图”。两者唯一的区别在于, 有向图中的边是有方向性的。下图即是一个有向图, 边的方向分别是: (1->2), (1->3), (3->1), (1->5), (2->3), (3->4), (3->5), (4->5), (1->6), (4->6)。要注意, 图中的边(1->3)和(3->1)是不同的。有向图和无向图的许多原理和算法是相通的。



#### 五、权重 (weight)

边的权重 (或者称为权值、开销、长度等), 也是一个非常核心的概念, 即每条边都有与之对应的值。例如当顶点代表某些物理地点时, 两个顶点间边的权重可以设置为路网中的开车距离。下图中顶点为 4 个城市:Beijing, Shanghai, Wuhan, Guangzhou, 边的权重设置为 2 城市之间的开车距离。有时候为了应对特殊情况, 边的权重可以是零或者负数, 也别忘了“图”是用来记录关联的东西, 并不是真正的地图。



## 六、路径/最短路径 (path/shortest path)

在图上任取两顶点, 分别作为起点 (start vertex) 和终点 (end vertex), 我们可以规划许多条由起点到终点的路线。不会来来回回绕圈子、不会重复经过同一个点和同一条边的路线, 就是一条“路径”。两点之间存在路径, 则称这 2 个顶点是连通的 (connected)。

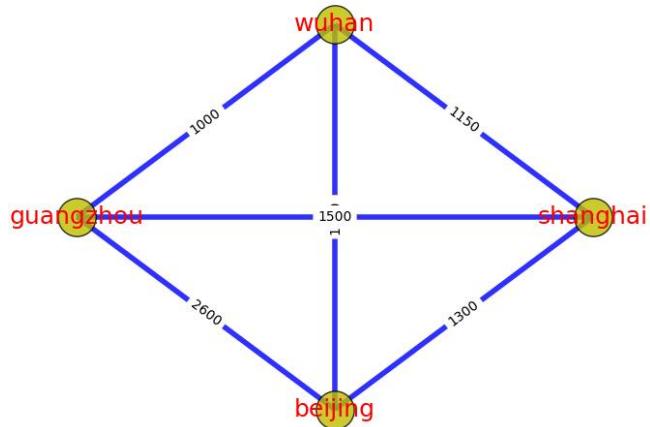
还是上图的例子, 北京->上海->广州, 是一条路径, 北京->武汉->广州, 是另一条路径, 北京->武汉->上海->广州, 也是一条路径。而北京->武汉->广州这条路径最短, 称为最短路径。

路径也有权重。路径经过的每一条边, 沿路加权重, 权重总和就是路径的权重 (通常只加边的权重, 而不考虑顶点的权重)。在路网中, 路径的权重, 可以想象成路径的总长度。在有向图中, 路径还必须跟随边的方向。

值得注意的是, 一条路径包含了顶点和边, 因此路径本身也构成了图结构, 只不过是一种特殊的图结构。

## 七、环 (loop)

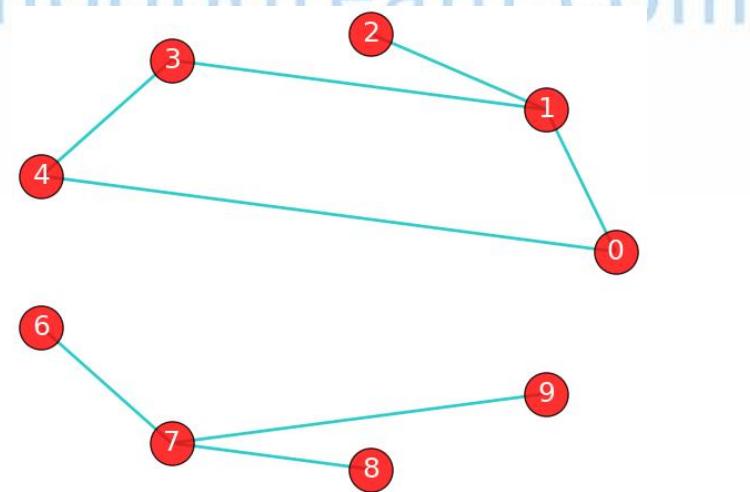
环, 也成为环路, 是一个与路径相似的概念。在路径的终点添加一条指向起点的边, 就构成一条环路。通俗点说就是绕圈。



上图中, 北京->上海->武汉->广州->北京, 就是一个环路。北京->武汉->上海->北京, 也是一个环路。与路径一样, 有向图中的环路也必须跟随边的方向。环本身也是一种特殊的图结构。

## 八、连通图/连通分量 (connected graph/connected component)

如果在图 G 中, 任意 2 个顶点之间都存在路径, 那么称 G 为连通图 (注意是任意 2 顶点)。上面那张城市之间的图, 每个城市之间都有路径, 因此是连通图。而下面这张图中, 顶点 8 和顶点 2 之间就不存在路径, 因此下图不是一个连通图, 当然该图中还有很多顶点之间不存在路径。



上图虽然不是一个连通图, 但它有多个连通子图: 0,1,2 顶点构成一个连通子图, 0,1,2,3,4 顶点构成的子图是连通图, 6,7,8,9 顶点构成的子图也是连通图, 当然还有很多子图。我们把一个图

的最大连通子图称为它的连通分量。 $0,1,2,3,4$  顶点构成的子图就是该图的最大连通子图, 也就是连通分量。连通分量有如下特点:

- 1) 是子图;
- 2) 子图是连通的;
- 3) 子图含有最大顶点数。

注意: “最大连通子图”指的是无法再扩展了, 不能包含更多顶点和边的子图。 $0,1,2,3,4$  顶点构成的子图已经无法再扩展了。

显然, 对于连通图来说, 它的最大连通子图就是其本身, 连通分量也是其本身。

## 九、有向图/无向图的度数

对于无向图

每个顶点的度数就是它连接边的数量

对于有向图

入度就是: 有向图的某个顶点作为终点的次数和。

出度就是: 有向图的某个顶点作为起点的次数和。

noobdream.com

## 7.2 图的存储

图有两种存储方式, 邻接矩阵和邻接表。

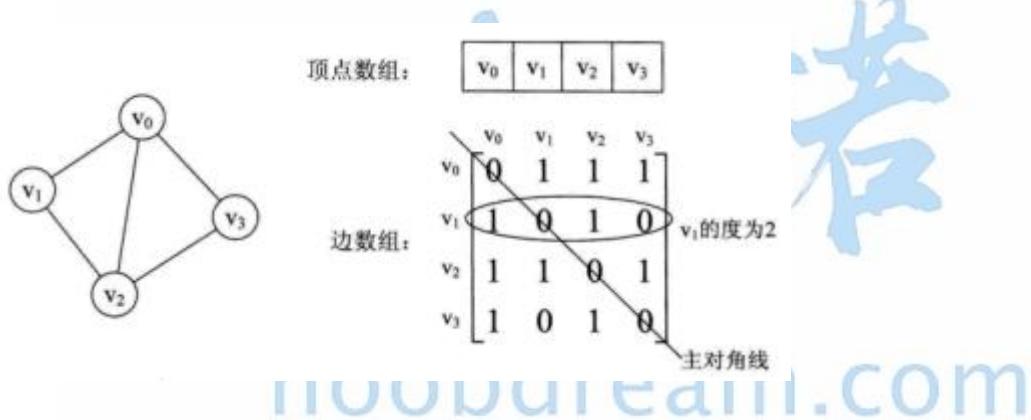
### 邻接矩阵

图的邻接矩阵存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息, 一个二维数组 (邻接矩阵) 存储图中的边或弧的信息。

设图  $G$  有  $n$  个顶点, 则邻接矩阵是一个  $n \times n$  的方阵, 定义为:

$$arc[i][j] = \begin{cases} 1, & \text{若 } (v_i, v_j) \in E \text{ 或 } \langle v_i, v_j \rangle \in E \\ 0, & \text{反之} \end{cases}$$

看一个实例, 下图左就是一个无向图。



从上面可以看出, 无向图的边数组是一个对称矩阵。所谓对称矩阵就是  $n$  阶矩阵的元满足  $a_{ij} = a_{ji}$ 。即从矩阵的左上角到右下角的主对角线为轴, 右上角的元和左下角相对应的元全都是相等的。

从这个矩阵中, 很容易知道图中的信息。

- (1) 要判断任意两顶点是否有边无边就很容易了;
- (2) 要知道某个顶点的度, 其实就是这个顶点  $v_i$  在邻接矩阵中第  $i$  行或 (第  $i$  列) 的元素之和;
- (3) 求顶点  $v_i$  的所有邻接点就是将矩阵中第  $i$  行元素扫描一遍,  $arc[i][j]$  为 1 就是邻接点; 而有向图讲究入度和出度, 顶点  $v_i$  的入度为 1, 正好是第  $i$  列各数之和。顶点  $v_i$  的出度为 2, 即第  $i$  行的各数之和。

若图 G 是网图, 有 n 个顶点, 则邻接矩阵是一个 n\*n 的方阵, 定义为

$$arc[i][j] = \begin{cases} W_{ij}, & \text{若 } (v_i, v_j) \in E \text{ 或 } \langle v_i, v_j \rangle \in E \\ 0, & \text{若 } i = j \\ \infty, & \text{反之} \end{cases}$$

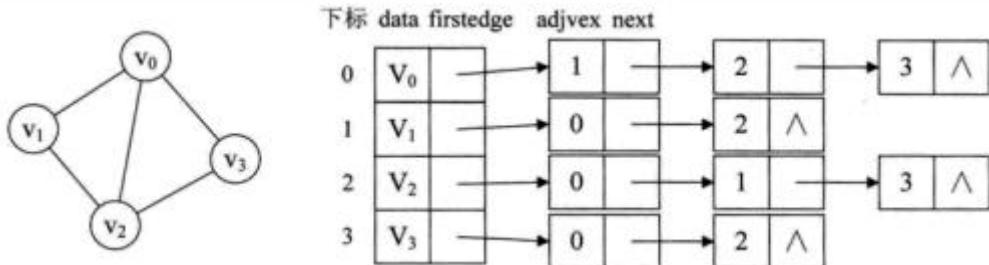
## 邻接表

邻接矩阵是不错的一种图存储结构, 但是, 对于边数相对顶点较少的图, 这种结构存在对存储空间的极大浪费。因此, 找到一种数组与链表相结合的存储方法称为邻接表。

邻接表的处理方法是这样的:

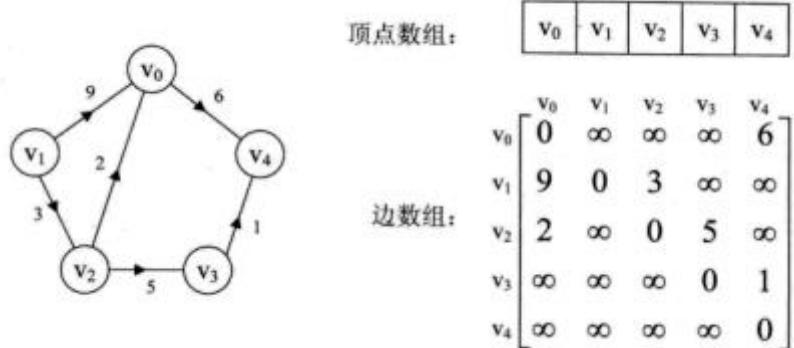
- (1) 图中顶点用一个一维数组存储, 当然, 顶点也可以用单链表来存储, 不过, 数组可以较容易的读取顶点的信息, 更加方便。
- (2) 图中每个顶点  $v_i$  的所有邻接点构成一个线性表, 由于邻接点的个数不定, 所以, 用单链表存储, 无向图称为顶点  $v_i$  的边表, 有向图则称为顶点  $v_i$  作为弧尾的出边表。

例如, 下图就是一个无向图的邻接表的结构。



从图中可以看出, 顶点表的各个结点由 data 和 firstedge 两个域表示, data 是数据域, 存储顶点的信息, firstedge 是指针域, 指向边表的第一个结点, 即此顶点的第一个邻接点。边表结点由 adjvex 和 next 两个域组成。adjvex 是邻接点域, 存储某顶点的邻接点在顶点表中的下标, next 则存储指向边表中下一个结点的指针。

对于带权值的网图, 可以在边表结点定义中再增加一个 weight 的数据域, 存储权值信息即可。如下图所示。



## 两者区别

对于一个具有  $n$  个顶点  $e$  条边的无向图

它的邻接表表示有  $n$  个顶点表结点  $2e$  个边表结点

对于一个具有  $n$  个顶点  $e$  条边的有向图

它的邻接表表示有  $n$  个顶点表结点  $e$  个边表结点

如果图中边的数目远远小于  $n^2$  称作稀疏图, 这时用邻接表表示比用邻接矩阵表示节省空间;

如果图中边的数目接近于  $n^2$ , 对于无向图接近于  $n*(n-1)$  称作稠密图, 考虑到邻接表中要附加链域, 采用邻接矩阵表示法为宜。

noobdream.com

## 7.3 并查集

并查集是解决集合类问题的, 比如朋友关系, 比如道路连通关系等等。

并查集本质是利用树形结构来加快区分集合的算法, 这么一看, 用 map 来区分也是可以的。

但是并查集在树形结构的特点上加入了路径压缩的思想, 使得算法效率远高于 map。

### 畅通工程 2

#### 题目描述:

某省调查城镇交通状况, 得到现有城镇道路统计表, 表中列出了每条道路直接连通的城镇。省政府“畅通工程”的目标是使全省任何两个城镇间都可以实现交通(但不一定有直接的道路相连, 只要互相间接通过道路可达即可)。问最少还需要建设多少条道路?

#### 输入描述:

测试输入包含若干测试用例。每个测试用例的第 1 行给出两个正整数, 分别是城镇数目  $N$  ( $< 1000$ ) 和道路数目  $M$ ; 随后的  $M$  行对应  $M$  条道路, 每行给出一对正整数, 分别是该条道路直接连通的两个城镇的编号。为简单起见, 城镇从 1 到  $N$  编号。

注意: 两个城市之间可以有多条道路相通, 也就是说

3 3

1 2

1 2

2 1

这种输入也是合法的

当  $N$  为 0 时, 输入结束, 该用例不被处理。

#### 输出描述:

对每个测试用例, 在 1 行里输出最少还需要建设的道路数目。

#### 输入样例#:

4 2

1 3

4 3

3 3

```
1 2
1 3
2 3
5 2
1 2
3 5
999 0
0
```

输出样例#:

```
1
0
2
998
```

题目来源:

DreamJudge 1319

题目解析: 直接用并查集算法模板即可。

noobdream.com

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 1005;
5. int fa[maxn];
6. //并查集模板
7. int find(int x) {
8.     if (x == fa[x]) return x;
9.     fa[x] = find(fa[x]); //路径压缩
10.    return fa[x];
11. }
12. int main(){
13.     int N, M;
```

```
14.     while(scanf("%d", &N) != EOF){  
15.         if (N == 0) break;  
16.         scanf("%d", &M);  
17.         for (int i = 1; i <= N; i++) fa[i] = i;  
18.         int sum = 0;  
19.         for (int i = 0; i < M; i++) {  
20.             int x, y;  
21.             scanf("%d%d", &x, &y);  
22.             int fx = find(x);  
23.             int fy = find(y);  
24.             if (fx != fy) {  
25.                 fa[fx] = fy;  
26.                 sum++;  
27.             }  
28.         }  
29.         printf("%d\n", N - sum - 1);  
30.     }  
31.     return 0;  
32. }
```



noobdream.com

## 7.4 最小生成树问题

几乎 99% 的最小生成树问题都可以用 kruskal 算法解决

下面给出 kruskal 和 prim 两种算法的通用模板, 方便同学们套用

### 畅通工程

#### 题目描述:

省政府“畅通工程”的目标是使全省任何两个村庄间都可以实现公路交通（但不一定有直接的公路相连，只要能间接通过公路可达即可）。经过调查评估，得到的统计表中列出了有可能建设公路的若干条道路的成本。现请你编写程序，计算出全省畅通需要的最低成本。

#### 输入描述:

测试输入包含若干测试用例。每个测试用例的第 1 行给出评估的道路条数 N、村庄数目 M (N, M < =100)；随后的 N 行对应村庄间道路的成本，每行给出一对正整数，分别是两个村庄的编号，以及此两村庄间道路的成本（也是正整数）。为简单起见，村庄从 1 到 M 编号。当 N 为 0 时，全部输入结束，相应的结果不要输出。

#### 输出描述:

对每个测试用例，在 1 行里输出全省畅通需要的最低成本。若统计数据不足以保证畅通，则输出“?”。

#### 输入样例#:

```
3 3
1 2 1
1 3 2
2 3 4
1 3
2 3 2
0 100
```

#### 输出样例#:

3

?

题目来源:

DreamJudge 1312

题目解析: 直接套用 kruskal 算法模板或 prim 算法模板皆可。

参考代码 (kruskal 模板)

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 105;
5. struct node {
6.     int u, v, w;
7. }edge[maxn * maxn];
8. int cmp(node A, node B) {
9.     return A.w < B.w;
10. }
11. int fa[maxn];
12. int find(int x) {
13.     if (x == fa[x]) return x;
14.     fa[x] = find(fa[x]);
15.     return fa[x];
16. }
17. int main(){
18.     int N , M;
19.     while(scanf("%d%d", &M,&N) != EOF){
20.         if (M == 0) break;
21.         for (int i = 0; i < M; i++) {
22.             scanf("%d%d%d", &edge[i].u, &edge[i].v, &edge[i].w);
23.         }
24.         for (int i = 1; i <= N; i++) fa[i] = i;
25.         sort(edge, edge + M, cmp);
26.         int sum = 0;
27.         int total = 0;
28.         for (int i = 0; i < M; i++) {
29.             int fx = find(edge[i].u);
```

```
30.         int fy = find(edge[i].v);
31.         if (fx != fy) {
32.             fa[fx] = fy;
33.             sum += edge[i].w;
34.             total++;//统计加入边数量
35.         }
36.     }
37.     if (total < N - 1)//不能生成树
38.         printf("?\\n");
39.     else printf("%d\\n", sum);
40. }
41. return 0;
42. }
```

参考代码 (prim 模板)

```
1. #include <bits/stdc++.h>
2. using namespace std;
3. #define INF 0x3f3f3f3f
4. const int maxn = 105;
5. int mpt[maxn][maxn];//邻接矩阵存储图
6. int dist[maxn];
7. int main(){
8.     int N , M;
9.     while(scanf("%d%d", &M,&N) != EOF) {
10.         if (M == 0) break;
11.         for (int i = 1; i <= N; i++) {
12.             dist[i] = INF;//初始化为无穷大
13.             for (int j = 1; j <= N; j++) {
14.                 if (i == j) mpt[i][j] = 0;
15.                 else mpt[i][j] = INF;
16.             }
17.         }
18.         for(int i = 0;i < M;i++) {
19.             int u, v, w;
20.             scanf("%d%d%d", &u, &v, &w);
21.             mpt[u][v] = min(mpt[u][v], w);//防止重边
22.             mpt[v][u] = min(mpt[v][u], w);//重边用最小的
23.         }
24.         int sum = 0;
25.         int flag = 0;
```

```
26.     for (int i = 1; i <= N; i++) dist[i] = mpt[1][i];
27.     for (int i = 1; i < N; i++) {
28.         int min_len = INF;
29.         int min_p = -1;
30.         for (int j = 1; j <= N; j++) {
31.             if (min_len > dist[j] && dist[j] != 0) {
32.                 min_len = dist[j];
33.                 min_p = j;
34.             }
35.         }
36.         if (min_p == -1) {
37.             flag = 1; break; //判断是否能生成树
38.         }
39.         sum += min_len;
40.         for (int j = 1; j <= N; j++) {
41.             if (dist[j] > mpt[min_p][j] && dist[j] != 0)
42.                 dist[j] = mpt[min_p][j];
43.         }
44.     }
45.     if (flag) printf("?\n"); //不能生成树
46.     else printf("%d\n", sum);
47. }
48. return 0;
49. }
```

noobdream.com

## 练习题目

DreamJudge 1311 继续畅通工程

DreamJudge 1341 还是畅通工程

DreamJudge 1183 Freckles

DreamJudge 1234 Jungle Roads

## 7.5 最短路径问题

### Floyd 算法特点

- 1、适合求多源最短路径
- 2、可以求最小环
- 3、可以有负边权, 不能有负环
- 4、可以求有向图的传递闭包
- 5、时间复杂度  $O(n^3)$

### 单源最短路径算法的选择?

Dijkstra + 堆优化? 无法处理负边权的问题

SPFA + 堆优化? 期望复杂度很优秀, 但是复杂度不稳定

基于本书面向的是计算机考研机试, 几乎不可能出现故意构造大量数据使得 SPFA 超时的情况发生, 就如同不可能在机试中故意构造数据使得 sort 函数的复杂度退化到  $O(n^2)$  一样。当然, 也为了防止本书爆红之后被出题老师故意针对, 后面我们也附上 Dijkstra 的通用算法模板以备不时之需。

noobdream.com

### 最短路

#### 题目描述:

在每年的校赛里, 所有进入决赛的同学都会获得一件很漂亮的 t-shirt。但是每当我们的工作人员把上百件的衣服从商店运回到赛场的时候, 却是非常累的! 所以现在他们想要寻找最短的从商店到赛场的路线, 你可以帮助他们吗?

#### 输入描述:

输入包括多组数据。每组数据第一行是两个整数  $N, M$  ( $N \leq 100, M \leq 10000$ ) ,  $N$  表示成都的大街上有几个路口, 标号为 1 的路口是商店所在地, 标号为  $N$  的路口是赛场所在地,  $M$  则表示在成都有几条路。 $N=M=0$  表示输入结束。接下来  $M$  行, 每行包括 3 个整数  $A, B, C$  ( $1 \leq A, B \leq N, 1 \leq C \leq 1000$ ) , 表示在路口  $A$  与路口  $B$  之间有一条路, 我们的工作人员需要  $C$  分钟的时间走过这条路。

输入保证至少存在 1 条商店到赛场的路线。

**输出描述:**

对于每组输入, 输出一行, 表示工作人员从商店走到赛场的最短时间

**输入样例#:**

```
2 1
1 2 3
3 3
1 2 5
2 3 5
3 1 2
0 0
```

**输出样例#:**

```
3
2
```

**题目来源:**

DreamJudge 1565

题目解析: 最短路径的模板题, 直接使用最短路径模板算法即可。

**参考代码 (通用模板)**

```
1. /*
2. spfa + vector
3. SPFA 可以有负边权、可以用一点个入队是否超过 N 次判断是否存在负环
4. */
5. #include <bits/stdc++.h>
6. using namespace std;
7.
8. #define INF 0x3f3f3f3f
9. const int maxn = 105;
10. int n, m;
11.
```

```
12. struct Edge{  
13.     int u, v, w;  
14.     Edge(int u, int v, int w):u(u),v(v),w(w) {}  
15. };  
16.  
17. vector<Edge> edges;  
18. vector<int> G[maxn];  
19. int dist[maxn]; // 存放起点到 i 点的最短距离  
20. int vis[maxn]; // 标记是否访问过  
21. int p[maxn]; // 存放路径  
22.  
23. void spfa(int s) {  
24.     queue<int> q; // 如果这个 spfa 超时的时候可以把队列改为和 dijkstra 一样的优先队列  
25.     for (int i = 0; i <= n; i++) dist[i] = INF;  
26.     dist[s] = 0;  
27.     memset(vis, 0, sizeof(vis));  
28.     q.push(s);  
29.     while (!q.empty()) {  
30.         int u = q.front(); q.pop();  
31.         vis[u] = 0;  
32.         for (int i = 0; i < G[u].size(); i++) {  
33.             Edge& e = edges[G[u][i]];  
34.             if (dist[e.v] > dist[u] + e.w) { // 松弛过程  
35.                 dist[e.v] = dist[u] + e.w;  
36.                 p[e.v] = u; // 松弛过程 记录路径  
37.                 if (!vis[e.v]) {  
38.                     vis[e.v] = 1;  
39.                     q.push(e.v);  
40.                 }  
41.             }  
42.         }  
43.     }  
44. }  
45.  
46. void addedge(int u, int v, int w) {  
47.     edges.push_back(Edge(u, v, w));  
48.     int sz = edges.size();  
49.     G[u].push_back(sz - 1);  
50. }  
51.  
52. void init() {  
53.     for(int i = 0; i <= n; i++) G[i].clear();  
54.     edges.clear();
```

```
55. }
56.
57. int main() {
58.     while (scanf("%d%d", &n, &m) != EOF) {
59.         if (n + m == 0) break;
60.         init();
61.         for (int i = 0; i < m; i++) {
62.             int a, b, c;
63.             scanf("%d%d%d", &a, &b, &c);
64.             addedge(a, b, c);
65.             addedge(b, a, c);
66.         }
67.         spfa(1);
68.         printf("%d\n", dist[n]);
69.     }
70.     return 0;
71. }
```

参考代码 (floyd 模板)



```
1. /*
2. floyd 算法可以求多源最短路径
3. */
4. #include <bits/stdc++.h>
5. using namespace std;
6.
7. #define INF 0x3f3f3f3f
8. const int maxn = 105;
9. int mpt[maxn][maxn];
10. int n, m;
11.
12. void floyd() {
13.     for (int k = 1; k <= n; k++) {
14.         for (int i = 1; i <= n; i++) {
15.             for (int j = 1; j <= n; j++) {
16.                 mpt[i][j] = min(mpt[i][k] + mpt[k][j], mpt[i][j]);
17.             }
18.         }
19.     }
20. }
```

```
21.
22. int main() {
23.     while (scanf("%d%d", &n, &m) != EOF) {
24.         if (n + m == 0) break;
25.         for (int i = 1; i <= n; i++) {
26.             for (int j = 1; j <= n; j++) {
27.                 if (i == j) mpt[i][j] = 0;
28.                 else mpt[i][j] = INF;
29.             }
30.         }
31.         for (int i = 1; i <= m; i++) {
32.             int a, b, c;
33.             scanf("%d%d%d", &a, &b, &c);
34.             if (c < mpt[a][b]) { //注意重边
35.                 mpt[a][b] = c;
36.                 mpt[b][a] = c;
37.             }
38.         }
39.         floyd();
40.         printf("%d\n", mpt[1][n]);
41.     }
42.     return 0;
43. }
```

参考代码 (dijkstra 模板)

```
1. // dijkstra + 堆优化
2. #include <bits/stdc++.h>
3. using namespace std;
4.
5. #define INF 0x3f3f3f3f
6. const int maxn = 105;
7. int n, m;
8.
9. struct Edge{
10.     int u, v, w;
11.     Edge(int u, int v, int w):u(u),v(v),w(w) {}
12. };
13.
14. struct node {
15.     int d, u;
16.     node(int d, int u):d(d),u(u) {}
17.     friend bool operator < (node a, node b) {
```

```
18.         return a.d > b.d;
19.     }
20. }
21.
22. vector<Edge> edges;
23. vector<int> G[maxn];
24. int dist[maxn]; // 存放起点到 i 点的最短距离
25. int vis[maxn]; // 标记是否访问过
26. int p[maxn]; // 存放路径
27.
28. void dijkstra(int s) {
29.     priority_queue<node> q;
30.     for (int i = 0; i <= n; i++) dist[i] = INF;
31.     dist[s] = 0;
32.     memset(vis, 0, sizeof(vis));
33.     q.push(node(0, s));
34.     int cnt = 0; //统计松弛次数
35.     while (!q.empty()) {
36.         node now = q.top(); q.pop();
37.         int u = now.u;
38.         if (vis[u]) continue;
39.         vis[u] = 1;
40.         cnt++;
41.         if (cnt >= n) break; // 小优化
42.         for (int i = 0; i < G[u].size(); i++) { // // Sum -> O(E)
43.             Edge& e = edges[G[u][i]];
44.             if (dist[e.v] > dist[u] + e.w) { // O(lgV)
45.                 dist[e.v] = dist[u] + e.w;
46.                 p[e.v] = G[u][i];
47.                 q.push(node(dist[e.v], e.v));
48.             }
49.         }
50.     }
51. }
52.
53. void addedge(int u, int v, int w) {
54.     edges.push_back(Edge(u, v, w));
55.     int sz = edges.size();
56.     G[u].push_back(sz - 1);
57. }
58.
59. void init() {
60.     for(int i = 0; i <= n; i++) G[i].clear();
```

```
61.     edges.clear();
62. }
63.
64. int main() {
65.     while (scanf("%d%d", &n, &m) != EOF) {
66.         if (n + m == 0) break;
67.         init();
68.         for (int i = 0; i < m; i++) {
69.             int a, b, c;
70.             scanf("%d%d%d", &a, &b, &c);
71.             addedge(a, b, c);
72.             addedge(b, a, c);
73.         }
74.         dijkstra(1);
75.         printf("%d\n", dist[n]);
76.     }
77.     return 0;
78. }
```



由于部分同学只能使用 C 语言进行机试, 下面给出 C 语言实现的最短路算法模板。

SPFA 算法模板 (C 语言实现)

```
1. #include<stdio.h>
2. #include<string.h>
3.
4. #define INF 0xffffffff
5. struct node {
6.     int to, next, val;
7. }edge[100005];//边的数量 100005
8. int head[1005];//点的数量 1005
9. int dist[1005];
10. int vis[1005];
11. int Q[100005];//定义一个数组模拟的队列
12. void SPFA(int s, int n) {//SPFA 算法模板
13.     int l = 0;//定义队首
14.     int r = 0;//定义队尾
```

```
15.     memset(vis,0,sizeof(vis));
16.     for(int i = 1; i <= n; i++)
17.         dist[i] = INF;
18.     dist[s] = 0;
19.     vis[s] = 1;
20.     Q[r++] = s;//将起点入队
21.     while (l < r) {//当队列不为空
22.         int now = Q[l];//取队首元素
23.         l++; //出队
24.         vis[now] = 0;
25.         for(int i = head[now];i != -1;i = edge[i].next) {
26.             int to = edge[i].to;
27.             if(dist[to] > dist[now] + edge[i].val) {
28.                 dist[to] = dist[now] + edge[i].val;
29.                 if(vis[to] == 0) {
30.                     vis[to] = 1;
31.                     Q[r++] = to; //将点 to 入队
32.                 }
33.             }
34.         }
35.     }
36. }
37. int k;
38. void init() {//初始化
39.     k = 0;
40.     memset(head, -1, sizeof(head));
41. }
42. void add(int x, int y, int val) {//加边操作
43.     edge[k].to = y;
44.     edge[k].val = val;
45.     edge[k].next = head[x];
46.     head[x] = k++;
47. }
48. int main() {
49.     int n, m;
50.     while(scanf("%d%d", &n, &m) != EOF){
51.         if(n == 0 && m == 0)break;
52.         init();
53.         int a, b, c;
54.         for(int i = 1; i <= m; i++) {
55.             scanf("%d%d%d", &a, &b, &c);
56.             add(a, b, c);//加入有向边 a->b 权值为 c
57.             add(b, a, c);
58.     }
59. }
```

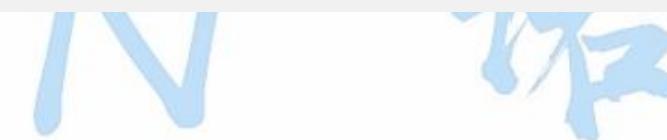
```
58.     }
59.     SPFA(1, n); //传入起点和点的数量
60.     printf("%d\n", dist[n]);
61. }
62. return 0;
63. }
```

## 练习题目

DreamJudge 1344 最短路径问题

DreamJudge 1286 最短路径

DreamJudge 1224 I Wanna Go Home



noobdream.com

## 7.6 拓扑排序

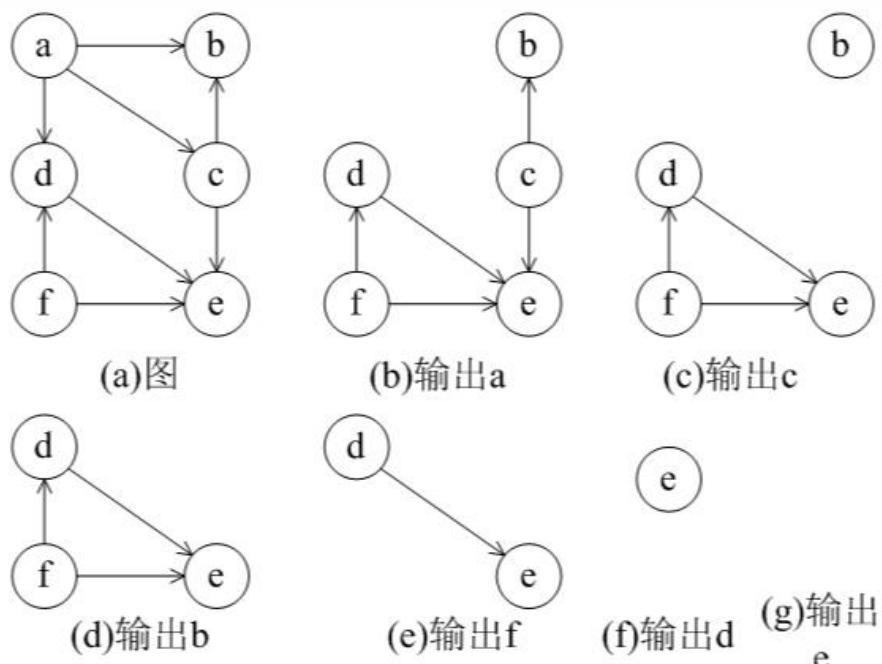
在一个有向图中, 对所有的节点进行排序, 要求没有一个节点指向它前面的节点。

先统计所有节点的入度, 对于入度为 0 的节点就可以分离出来, 然后把这个节点指向的节点的入度减一。

一直做改操作, 直到所有的节点都被分离出来。

如果最后不存在入度为 0 的节点, 那就说明有环, 不存在拓扑排序, 也就是很多题目的无解的情况。

下面是算法的演示过程。



拓扑排序序列: acbfde

### 确定比赛名次

题目描述:

有  $N$  个比赛队 ( $1 \leq N \leq 500$ ), 编号依次为 1, 2, 3, . . . . . ,  $N$  进行比赛, 比赛结束后, 裁判委员会要将所有参赛队伍从前往后依次排名, 但现在裁判委员会不能直接获得每个队的比赛成绩, 只知道每场比赛的结果, 即  $P_1$  赢  $P_2$ , 用  $P_1$ ,  $P_2$  表示, 排名时  $P_1$  在  $P_2$  之前。现在请你编程序确定排名。

输入描述:

输入有若干组, 每组中的第一行为二个数  $N$  ( $1 \leq N \leq 500$ ) ,  $M$ ; 其中  $N$  表示队伍的个数,  $M$  表示接着有  $M$  行的输入数据。接下来的  $M$  行数据中, 每行也有两个整数  $P_1$ ,  $P_2$  表示即  $P_1$  队赢了  $P_2$  队。

#### 输出描述:

给出一个符合要求的排名。输出时队伍号之间有空格, 最后一名后面没有空格。

其他说明: 符合条件的排名可能不是唯一的, 此时要求输出时编号小的队伍在前; 输入数据保证是正确的, 即输入数据确保一定能有一个符合要求的排名。

#### 输入样例#:

```
4 3
1 2
2 3
4 3
```

#### 输出样例#:

```
1 2 4 3
```

#### 题目来源:

DreamJudge 1566

noobdream.com

题目解析: 使用拓扑排序算法模板即可解决。

#### 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. const int maxn = 505;
5. bool mpt[maxn][maxn];
6. int lev[maxn];
7. vector<int> v[maxn];
8. priority_queue<int, vector<int>, greater<int> > q;
9. //拓扑排序
10. void topo(int n) {
11.     for (int i = 1; i <= n; i++) {
```

```
12.         if (!lev[i]) q.push(i); //队列里面存的是入度为 0 的点
13.     }
14.     int flag = 0; //统计出队的元素个数
15.     while(!q.empty()) {
16.         int now = q.top();
17.         q.pop();
18.         if (flag) printf(" %d", now);
19.         else printf("%d", now);
20.         flag++;
21.         for (int i = 0; i < v[now].size(); i++) {
22.             int next = v[now][i];
23.             lev[next]--;
24.             if (!lev[next]) q.push(next);
25.         }
26.     }
27.     if (flag != n) {
28.         printf("这个图有环、并没有拓扑排序\n");
29.     }
30. }
31.
32. int main() {
33.     int n, m;
34.     while(scanf("%d%d", &n, &m) != EOF) {
35.         memset(mpt, 0, sizeof(mpt));
36.         for (int i = 1; i <= m; i++) {
37.             int a, b;
38.             scanf("%d%d", &a, &b);
39.             mpt[a][b] = 1;
40.         }
41.         for (int i = 1; i <= n; i++) {
42.             v[i].clear();
43.             for (int j = 1; j <= n; j++) {
44.                 if (mpt[i][j]) {
45.                     v[i].push_back(j);
46.                     lev[j]++;
47.                 }
48.             }
49.         }
50.         topo(n);
51.         printf("\n");
52.     }
53.     return 0;
54. }
```

## 第八章 动态规划

本章我们重点讲解一些常见的动态规划题型, 包括递推求解、最大子段和、最长上升子序列 (LIS) 、最长公共子序列 (LCS) 、背包类问题、记忆化搜索、字符串相关的动态规划等内容。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动态规划问题。



本书配套视频精讲: <https://www.bilibili.com/video/av81203473>

## 8.1 递推求解

首先, 什么是动态规划?

动态规划是通过拆分问题, 定义问题状态和状态之间的关系, 使得问题能够以递推(或者说分治)的方式去解决。其实就是分解问题, 分而治之。可能这样说大家都不太理解, 其实这个有点类似于数学中的递推公式。来举一个简单的例子, 看下边这个题:

**N 阶楼梯上楼问题:** 一次可以走两阶或一阶, 问有多少种上楼方式。

这就是动态规划最简单的一个例子。拿到这个题, 大家是不是都有点迷, 这个到底怎么做? 是不是没有思路, 那么按照动态规划思想, 我们可以先分析下问题, 每次都有两种跳法, 分别是一阶或者两阶, 那么如果当前是第  $n$  个台阶, 那么跳法是不是  $(n-1)$  台阶的跳法数加上  $(n-2)$  台阶的跳法数? 如果划算成公式是  $F(n) = F(n-1) + F(n-2)$ 。

$F(n)$  代表第  $n$  阶台阶的跳法的数量。

这不就相当于找到了一个递推公式, 然后来进行计算。

### N 阶楼梯上楼问题

**题目描述:**

N 阶楼梯上楼问题: 一次可以走两阶或一阶, 问有多少种上楼方式。(要求采用非递归)

**输入描述:**

输入包括一个整数  $N$ ,  $(1 \leq N \leq 90)$ 。

**输出描述:**

可能有多组测试数据, 对于每组数据,

输出当楼梯阶数是  $N$  时的上楼方式个数。

**输入样例#:**

4

**输出样例#:**

5

**题目来源:**

DreamJudge 1413

题目解析: 根据上面的分析得出公式  $F(n) = F(n-1) + F(n-2)$ , 然后递推即可。

## 参考代码

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i,N;
5.     long long a[90];
6.     while(scanf("%d",&N) != EOF) {
7.         a[1]=1;
8.         a[2]=2;
9.         for(i=3;i<=N;i++)
10.             a[i]=a[i-1]+a[i-2];
11.         printf("%lld\n",a[N]);
12.     }
13.     return 0;
14. }
```



noobdream.com

## 练习题目

DreamJudge 1197 吃糖果

DreamJudge 1033 细菌的繁殖

DreamJudge 1726 不连续 1 的子串数量

## 8.2 最大子段和

最大子段和的考察方式有以下几种

- 1、直接求最大子段和的值
- 2、要求记录最大子段值的起始坐标和终止坐标或者起始值和终止值
- 3、首尾可以连接成环的最大子段和的值

### 最大序列和

**题目描述:**

给出一个整数序列  $S$ , 其中有  $N$  个数, 定义其中一个非空连续子序列  $T$  中所有数的和为  $T$  的“序列和”。对于  $S$  的所有非空连续子序列  $T$ , 求最大的序列和。变量条件:  $N$  为正整数,  $N \leq 1000000$ , 结果序列和在范围  $(-2^{63}, 2^{63}-1)$  以内。

**输入描述:**

第一行为一个正整数  $N$ , 第二行为  $N$  个整数, 表示序列中的数。

**输出描述:**

输入可能包括多组数据, 对于每一组输入数据,

仅输出一个数, 表示最大序列和。

**输入样例#:**

```
5
1 5 -3 2 4
6
1 -2 3 4 -10 6
4
-3 -1 -2 -5
```

**输出样例#:**

```
9
7
-1
```

题目来源:

DreamJudge 1172

题目解析: 由于  $N$  很大, 所以我们不能暴力的枚举起点和终点, 使用动态规划的思想, 我们可以发现一个特点, 即我们只需要从一个正数开始不断的累加, 然后更新其中的最大值即可。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int dp[1000010];
5. int a[1000010];
6. long long maxx;
7. int main() {
8.     int n ;
9.     while(cin >> n){
10.         for(int i = 0; i < n; i++)
11.             cin >> a[i];
12.         dp[0] = a[0];
13.         maxx = a[0];//最小的情况就是不选那么答案就是 0
14.         for(int i = 1; i < n; i++){
15.             dp[i] = max(dp[i-1] + a[i], a[i]);
16.             if(maxx < dp[i]) {//如果累加到更大的值则更新
17.                 maxx = dp[i];
18.             }
19.         }
20.         cout << maxx << endl;
21.     }
22.     return 0;
23. }
```

接下来我来看一道关于最大子段和的进阶应用方法

### 字符串区间翻转

#### 题目描述:

小诺有一个由 0 和 1 组成的字符串

现在小诺有一次机会, 可以选择一个任意的区间  $[L, R]$ , 将该区间内的所有字符串进行翻转(即  $0 \rightarrow 1, 1 \rightarrow 0$ )。

请问小诺经过一次翻转之后字符串中最多会有多少个 1?

#### 输入描述:

第一行输入一个正整数  $n$ , 表示字符串长度,  $n \leq 10^7$ 。

接下来一行输入一个 01 字符串。

#### 输出描述:

输出题目要求的答案。

#### 输入样例#:

```
4
1001
```

#### 输出样例#:

```
4
```

#### 题目来源:

DreamJudge 1642

**题目解析:** 首先, 看到题目的数据范围, 第一反应就是这个题只能用  $O(n)$  时间复杂度的算法。那么可供我们选择的就不多了。一种方式是深挖题目, 可以发现其实题目就是让我找出一段 0 比 1 的个数多的最多的区间, 我们可以用满分篇中讲到的毛毛虫算法来进行不断蠕动解决。那么有没有另一种解决方法呢? 和本节讲到的最大子段和有半毛钱关系吗? 很多同学横看竖看怎么都想不到最大子段和怎么去解决这道题。这个时候就要教同学们一招很有用的东西, 叫拨开云雾见青天(俗称给题目脱衣服), 很多时候题目就不会特别直接的让你一眼看出这个题能用什么方法解决, 一眼能看出解决方案的题叫做裸题, 而很多题目穿上了衣服, 大部分同学就懵了。

我们把 01 字符串的 0 变成 1, 1 变成-1, 然后构成一个 1 和-1 的字符串。

对这样一个字符串去求它的最大子段和即可, 最后再把 1 的个数加上就是最终的答案。

## 参考代码

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int dp[10000005];
5. int a[10000005];
6. char s[10000005];
7. int _max(int x, int y) {
8.     return x > y ? x : y;
9. }
10. int main() {
11.     int n;
12.     while (scanf("%d", &n) != EOF) {
13.         scanf("%s", s);
14.         for (int i = 0; i < n; i++) {
15.             if (s[i] == '0') a[i] = 1;
16.             else a[i] = -1;
17.         }
18.         memset(dp, 0, sizeof(dp));
19.         dp[0] = a[0];
20.         int maxx = 0;//可以为空
21.         for (int i = 0; i < n; i++) {
22.             dp[i] = _max(dp[i - 1] + a[i], a[i]);
23.             if (maxx < dp[i]) maxx = dp[i];
24.         }
25.         int ans = 0;//统计 1 的个数
26.         for (int i = 0; i < n; i++)
27.             if (s[i] == '1') ans++;
28.         printf("%d\n", maxx + ans);
29.     }
30.     return 0;
31. }
```

## 练习题目

DreamJudge 1334 最大连续子序列

DreamJudge 1703 最大子串和

## 8.3 最长上升子序列 (LIS)

最长上升子序列(Longest Increasing Subsequence, 简称 LIS)是 dp 中比较经典的一个算法模型, 它有一种朴素的算法  $O(n^2)$  和一种优化版的算法  $O(n \log n)$  实现, 通过它, 我们可以进一步了解 dp 的思想。

接下来我们实现  $O(n^2)$  的算法 LIS\_nn()

首先确定状态转移方程  $dp[i]$  代表以第  $i$  项为结尾的 LIS 的长度

$dp[i] = \max(dp[i], \max(dp[j]) + 1) \quad \text{if } j < i \text{ and } a[j] < a[i]$

根据上面的状态转移方程可以写出下面的代码

```
1. int LIS_nn() {
2.     int ans = 0;
3.     for (int i = 1; i <= n; ++i) {
4.         dp[i] = 1;
5.         for (int j = 1; j < i; ++j) {
6.             if (a[j] < a[i]) { //要满足上升的条件
7.                 dp[i] = max(dp[i], dp[j] + 1);
8.             }
9.         }
10.        ans = max(ans, dp[i]);
11.    }
12.    return ans;
13. }
```

我们继续思考一下刚才的算法是否还有优化的空间呢?

在刚才的内层 for 我们从前往后找一个最大的 LIS 值, 仔细想一下是否可以发现这个值一定是单调递增的呢?

由于这个值是单调递增的, 所以我们就没有必要使用从前往后遍历的方法, 可以使用二分查找来优化这个寻找的过程。

于是可以实现  $O(n \log n)$  算法的 LIS\_nlogn() 函数

```
1. int LIS_nlogn() {
2.     int len = 1;
```

```
3.     dp[1] = a[1];
4.
5.     for (int i = 2; i <= n; ++i) {
6.         if (a[i] > dp[len]) {
7.             dp[++len] = a[i];
8.         } else {
9.             int pos = lower_bound(dp, dp + len, a[i]) - dp;
10.            dp[pos] = a[i];
11.        }
12.    }
13.    return len;
14. }
```

上面的代码是求最长上升子序列的长度

也可以求最长上升子序列的累加值

## 最大上升子序列和

### 题目描述:

一个数的序列  $b_i$ , 当  $b_1 < b_2 < \dots < b_S$  的时候, 我们称这个序列是上升的。对于给定的一个序列  $(a_1, a_2, \dots, a_N)$ , 我们可以得到一些上升的子序列  $(a_{i1}, a_{i2}, \dots, a_{iK})$ , 这里  $1 \leq i_1 < i_2 < \dots < i_K \leq N$ 。比如, 对于序列  $(1, 7, 3, 5, 9, 4, 8)$ , 有它的一些上升子序列, 如  $(1, 7)$ ,  $(3, 4, 8)$  等等。这些子序列中序列和最大为 18, 为子序列  $(1, 3, 5, 9)$  的和。你的任务, 就是对于给定的序列, 求出最大上升子序列和。注意, 最长的上升子序列的和不一定是最大的, 比如序列  $(100, 1, 2, 3)$  的最大上升子序列和为 100, 而最长上升子序列为  $(1, 2, 3)$ 。

### 输入描述:

输入包含多组测试数据。

每组测试数据由两行组成。第一行是序列的长度  $N$  ( $1 \leq N \leq 1000$ )。第二行给出序列中的  $N$  个整数, 这些整数的取值范围都在 0 到 10000 (可能重复)。

### 输出描述:

对于每组测试数据, 输出其最大上升子序列和。

### 输入样例#:

1 7 3 5 9 4 8

输出样例#:

18

题目来源:

DreamJudge 1257

题目解析: 将上面的求长度的 LIS 模板稍作修改即可得到求累加和的 LIS。

参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int dp[1001], a[1001], n;
5.
6. int LIS_nn() {
7.     int ans = 0;
8.     for (int i = 1; i <= n; ++i) {
9.         dp[i] = a[i];
10.        for (int j = 1; j < i; ++j) {
11.            if (a[j] < a[i]) {
12.                dp[i] = max(dp[i], dp[j] + a[i]);
13.            }
14.        }
15.        ans = max(ans, dp[i]);
16.    }
17.    return ans;
18. }
19.
20. int main() {
21.     while (cin >> n) {
22.         for (int i = 1; i <= n; ++i) {
23.             cin >> a[i];
24.         }
25.         cout << LIS_nn() << endl;
26.     }
27.     return 0;
28. }
```

### 练习题目

DreamJudge 1256 拦截导弹

DreamJudge 1253 合唱队形

DreamJudge 1836 最长递减子序列



## 8.4 最长公共子序列 (LCS)

### 定义

最长公共子序列 (LCS) 是一个在一个序列集合中 (通常为两个序列) 用来查找所有序列中最长子序列的问题。一个数列, 如果分别是两个或多个已知数列的子序列, 且是所有符合此条件序列中最长的, 则称为已知序列的最长公共子序列。

接下来我们分析算法的状态转移方程

$dp[i, j]$  代表  $a$  字符串前  $i$  个字符组成的子串和  $b$  字符串前  $j$  个字符组成的子串的 LCS。

那么

$$\begin{aligned} dp[i, j] &= 0 && \text{if } i = 0 \text{ or } j = 0 \\ dp[i, j] &= dp[i - 1, j - 1] + 1 && \text{if } i, j > 0 \text{ and } a_i = b_j \\ dp[i, j] &= \max\{dp[i, j - 1], dp[i - 1, j]\} && \text{if } i, j > 0 \text{ and } a_i \neq b_j \end{aligned}$$

根据上面的状态转移方程可以写出一下代码

```
1. for(int i = 1; i <= lena; ++i) {  
2.     for (int j = 1; j <= lenb; ++j) {  
3.         if(a[i - 1] == b[j - 1]) {  
4.             dp[i][j] = dp[i - 1][j - 1] + 1;  
5.         } else {  
6.             dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);  
7.         }  
8.     }  
9. }
```

### Coincidence

题目描述:

Find a longest common subsequence of two strings.

输入描述:

First and second line of each input case contain two strings of lowercase character a…z. There are no spaces before, inside or after the strings. Lengths of strings do not exceed 100.

### 输出描述:

For each case, output  $k$  — the length of a longest common subsequence in one line.

### 输入样例#:

abcd

cxybydz

### 输出样例#:

2

### 题目来源:

DreamJudge 1293

题目解析: 最长公共子序列 (LCS) 模板题。

### 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int dp[101][101];
5. int main() {
6.     string a, b;
7.     memset(dp, 0, sizeof(dp));
8.     cin >> a >> b;
9.     int lena = a.size();
10.    int lenb = b.size();
11.    for(int i = 1; i <= lena; ++i) {
12.        for (int j = 1; j <= lenb; ++j) {
13.            if(a[i - 1] == b[j - 1]) {
14.                dp[i][j] = dp[i - 1][j - 1] + 1;
15.            } else {
16.                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
17.            }
18.        }
19.    }
20.    cout << dp[lena][lenb] << endl;
21.    return 0;
22. }
```

### 练习题目

DreamJudge 1731 最长公共子序列

DreamJudge 1730 最长连续公共子序列

DreamJudge 1737 骑车路线

DreamJudge 1664 最大连续子序列

DreamJudge 1571 最长连号



## 8.5 背包类问题

### 01 背包定义

01 背包问题是一个经典的问题, 给定  $N$  个物品和一个背包。物品  $i$  的重量是  $W_i$ , 其体积为  $C_i$ , 背包的容量为  $C$ 。问应该如何选择装入背包的物品, 使得装入背包的物品的总重量为最大。

通常, 背包类问题有以下三种考点

- 1、简单背包问题, 即没有重量属性, 只是判断能否刚好装满。
- 2、01 背包, 即要在能装下的前提下重量尽量大。
- 3、要求输出装入物品的方案或数量

```
1. // 01 背包模板
2. #include <iostream>
3. #include <string.h>
4. using namespace std;
5.
6. int dp[21][1010];
7. int w[21], c[21];
8.
9. int main() {
10.     int N, V;
11.     cin >> N >> V; // 输入物品数量 N 背包体积 V
12.     for (int i = 1; i <= N; ++i) {
13.         cin >> w[i] >> c[i]; // 每个物品的重量 wi 体积 ci
14.     }
15.     // 对于一个动态规划来说, 最重要的是找到状态转移方程。
16.     // 在 01 背包问题中, 一个物品要么装要么不装, 那么我们可以得出下面的式子
17.     // f[i,j] 代表前 i 个物品背包容量最大为 j 最多能装的物品总重量
18.     // f[i,j] = Max{ f[i-1,j-Ci]+Wi( j >= Ci ), f[i-1,j] }
19.     // 根据上面的状态转移方程可以写出下面的代码
20.     for (int i = 1; i <= N; ++i) {
21.         for (int j = 0; j <= V; ++j) {
22.             if(j >= c[i]) {
23.                 dp[i][j] = max(dp[i - 1][j - c[i]] + w[i], dp[i - 1][j]);
24.             }
25.             else {
26.                 dp[i][j] = dp[i-1][j];
27.             }
28.         }
29.     }
30. }
```

```
28.     }
29. }
30. //dp[i][j]表示前 i 个物品装在 j 体积的背包中最大的重量
31. cout << dp[N][V] << endl;
32. return 0;
33. }
```

## 简单背包问题

### 题目描述:

设有一个背包可以放入的物品重量为  $S$ ，现有  $n$  件物品，重量分别是  $w_1, w_2, w_3, \dots, w_n$ 。

问能否从这  $n$  件物品中选择若干件放入背包中，使得放入的重量之和正好为  $S$ 。

如果有满足条件的选择，则此背包有解，否则此背包问题无解。

### 输入描述:

输入数据有多行，包括放入的物品重量为  $s$ ，物品的件数  $n$ ，以及每件物品的重量（输入数据均为正整数）

多组测试数据。

### 输出描述:

对于每个测试实例，若满足条件则输出“YES”，若不满足则输出“NO”

### 输入样例#:

20 5

1 3 5 7 9

### 输出样例#:

YES

### 题目来源:

DreamJudge 1035

**题目解析:** 很多同学看了前面的 01 背包代码感觉自己理解了这类问题，然后做这道题的时候又一脸懵，为什么呢？因为没有深刻的理解 01 背包中的动态规划的思想。这个题其实是 01 背包的简化版，只有一个属性，但是要求刚好装满，只需要在 01 背包的基础上稍作改变，具

体请看下面的参考代码。

## 参考代码

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     int dp[1005][1005] = {0}; // 只有 0 和 1 0 表示不可以 1 表示可以
6.     int w[1005];
7.     int s, n;
8.     while (scanf("%d%d", &s, &n) != EOF) {
9.         int i, j;
10.        for (i = 1; i <= n; i++) {
11.            scanf("%d", &w[i]);
12.        }
13.        memset(dp, 0, sizeof(dp));
14.        dp[0][0] = 1; // 前 0 件物品中能拼凑出 0 重量的方案, 所以为 1
15.        for (i = 1; i <= n; i++) {
16.            for (j = s; j >= 0; j--) {
17.                if (dp[i - 1][j] == 1) dp[i][j] = 1;
18.                if (j - w[i] >= 0 && dp[i - 1][j - w[i]] == 1) dp[i][j] = 1;
19.            }
20.        }
21.        if (dp[n][s] == 1) printf("YES\n");
22.        else printf("NO\n");
23.    }
24.    return 0;
25. }
```

## 练习题目

DreamJudge 1123 小偷的背包

DreamJudge 1567 Buyer

DreamJudge 1086 采药

## 8.6 记忆化搜索

### 简述

记忆化搜索实际上是递归来实现的，但是递归的过程中有许多的结果是被反复计算的，这样会大大降低算法的执行效率。

而记忆化搜索是在递归的过程中，将已经计算出来的结果保存起来，当之后的计算用到的时候直接取出结果，避免重复运算，因此极大的提高了算法的效率。

记忆化搜索，是最容易写，也是效率较高的一种做法。

虽然本质上是 DFS 这种搜索的思路，但其对搜索过的状态进行记录，从而完成对未知状态的推导，实际上也是一种 DP 的思想。

### 滑雪

#### 题目描述：

Michael 喜欢滑雪这并不奇怪，因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael 想知道在一个区域中最长底滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子

```
1  2  3  4  5
16 17 18 19  6
15 24 25 20  7
14 23 22 21  8
13 12 11 10  9
```

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为 24-17-16-1。当然 25-24-23-...-3-2-1 更长。事实上，这是最长的一条。

#### 输入描述：

多组测试数据。

输入的第一行表示区域的行数 R 和列数 C ( $1 \leq R, C \leq 100$ )。下面是 R 行，每行有 C 个整数，代表高度 h,  $0 \leq h \leq 10000$ 。

### 输出描述:

输出最长区域的长度。

### 输入样例#:

```
5 5
1 2 3 4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

### 输出样例#:

```
25
```

### 题目来源:

DreamJudge 1568

**题目解析:**在搜索的过程中使用记忆化的方式进行剪枝, 即将所有处理过的点的答案记录下来, 等下次到达这个点的时候可以直接返回答案, 而不需要再重复往下递归, 因为上一次已经走过这条路了。

noobdream.com

### 参考代码

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int n,m;
5. int a[105][105];
6. int dir[4][2] = {0, 1, 0, -1, 1, 0, -1, 0};
7. int dp[105][105];
8. // 记忆化搜索保证每个点只会计算一次
9. int dfs(int x,int y){
10.     if(dp[x][y]) return dp[x][y];//如果访问过则直接返回结果
11.     int maxx = 1;
12.     for(int i = 0; i < 4; i++){
13.         int tx = x + dir[i][0];
```

```
14.     int ty = y + dir[i][1];
15.     if(tx>=1&&tx<=n&&ty>=1&&ty<=m&&a[tx][ty]>a[x][y]){
16.         maxx = max(maxx, dfs(tx, ty) + 1); //自底向上的
17.     }
18. }
19. dp[x][y] = maxx; //记忆化
20. return maxx;
21. }
22.
23. int main() {
24.     while (cin >> n >> m) {
25.         for(int i = 1; i <= n; i++)
26.             for(int j = 1; j <= m; j++)
27.                 cin >> a[i][j];
28.         int ans = 0;
29.         memset(dp, 0, sizeof(dp));
30.         for(int i = 1; i <= n; i++) {
31.             for(int j = 1; j <= m; j++){
32.                 dp[i][j] = dfs(i,j);
33.                 ans = max(ans, dp[i][j]);
34.             }
35.         }
36.         cout << ans << endl;
37.     }
38.     return 0;
39. }
```



记忆化搜索与 dp 的不同点：

dp 需要对每个状态进行遍历，而记忆化搜索则可以排除无用状态。更重要的是，记忆化搜索还可以剪枝，这样一来，就大大降低了时间复杂度。

## 8.7 字符串相关的动态规划

字符串相关的动态题目非常多, 我们选取其中最具代表性也是最常考的三个知识点

### 1、最长公共子串

问题描述: 给出两个字符串, 找到最长公共子串, 并返回其长度。

输入:  $s = "ABCD"$ ,  $t = "EABDF"$

输出: 2

解释:  $s$  和  $t$  的最长公共子串为 “AB”



题目解析:

假定  $s$  的长度为  $n$ ,  $t$  的长度为  $m$ 。

首先考虑暴力破解,  $s$  有  $n^2$  (实际是  $n(n+1)/2 + 1$ , 此处近似) 个子串,  $t$  有  $m^2$  个子串, 复杂度为  $O(n^2m^2)$  但是暴力破解完全割裂了各个子问题的相互联系性。

倘若记  $s$  中以下标  $i - 1$  结尾的子串, 与  $t$  中以下标  $j - 1$  结尾的子串最长公共子串的长度为  $dp[i][j]$ , 则状态转移方程为

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & s[i-1] = t[j-1] \\ 0, & \text{otherwise} \end{cases}$$

我们定义的  $dp[i][j]$  很关键, 其 “定死” 了  $s$  必须取到  $s[i - 1]$  字符以及  $t$  必须取到  $t[j - 1]$  字符, 倘若两者不等, 则绝无法匹配; 倘若两者相等, 则可以向上个状态  $dp[i - 1][j - 1]$  转移而来。

## 2、最长公共子序列

问题描述: 给出两个字符串, 找到最长公共子序列 (LCS), 返回 LCS 的长度。

输入:  $s = \text{“ABCD”}$ ,  $t = \text{“EABDF”}$

输出: 3

解释:  $s$  和  $t$  的最长公共子序列为 “ABD”

题目解析:

不同于子串, 子序列的定义更加宽松。对于字符串  $s$ , 子串要求从  $s$  中顺序取出, 并且严格相邻; 而子序列则只要求顺序取出, 而不一定相邻(可以不连续)

子序列的定义等于直接宣告暴力破解的失败, 但是对于动态规划而言, 却无足轻重。可以依旧挪用上题的  $dp$  定义, 记  $s$  中以下标  $i - 1$  结尾的子序列, 与  $t$  中以下标  $j - 1$  结尾的子序列的最长公共子序列的长度为  $dp[i][j]$ 。因为, 子串也是子序列, 因此, 只要拓展状态转移方程即可。

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & s[i-1] = t[j-1] \\ \max(dp[i-1][j], dp[i][j-1]), & \text{otherwise} \end{cases}$$

子序列相较于子串的特殊在于,  $s$  中以下标  $i - 1$  结尾的子序列其本身不一定必须取到  $s[i - 1]$ , 因此转移方程也变得更加宽松。

noobaream.com

### 3、字符串相似度/编辑距离

给定两个单词 word1 和 word2, 计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作:

插入一个字符

删除一个字符

替换一个字符

输入: word1 = “horse”, word2 = “ros”

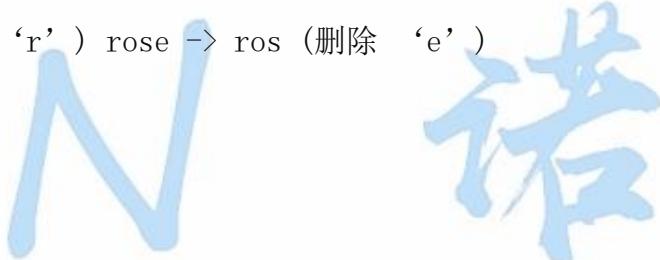
输出: 3

解释:

horse -> rorse (将' h' 替换为 'r' )

rorse -> rose (删除 'r' ) rose -> ros (删除 'e' )

题目解析:



仍旧沿用“结尾”法定义  $dp$ , 记  $s$  中以下标  $i - 1$  结尾的子串, 与  $t$  中以下标  $j - 1$  结尾的子串的最小距离为  $dp[i][j]$ , 则

$$\begin{aligned} t_1 &= 1 + \min(dp[i][j - 1], dp[i - 1][j]) \\ t_2 &= \begin{cases} dp[i - 1][j - 1], & s[i - 1] = t[j - 1] \\ dp[i - 1][j - 1] + 1, & \text{otherwise} \end{cases} \\ dp &= \min(t_1, t_2) \end{aligned}$$

具体而言,  $dp[i][j]$  有三种转移方式

1.  $dp[i - 1][j]$  转移到  $dp[i][j]$ , 即从  $s$  中多取向后一个字符, 此时对  $s$  动用“删除”操作, 或者说对  $t$  动用“插入”操作
2.  $dp[i][j - 1]$  转移到  $dp[i][j]$ , 即从  $t$  中多取向后一个字符, 此时对  $t$  动用“删除”操作, 或者说对  $s$  动用“插入”操作
3.  $dp[i - 1][j - 1]$  转移到  $dp[i][j]$ , 即分别从  $s, t$  中多向后取一个字符。若取出的两个字符相等, 无需操作; 反之, 动用“替换”操作。

### 练习题目

DreamJudge 1642 字符串区间翻转

DreamJudge 1730 最长连续公共子序列



## 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 - 高分篇的全部内容, 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](http://www.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 语言考研复习攻略》

《数据结构考研复习攻略》

《操作系统考研复习攻略》

《计算机网络考研复习攻略》

《计算机组成原理考研复习攻略》

《数据库考研复习攻略》

《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同学)
- 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](http://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~