

图灵程序设计丛书



挑战 (第2版) 程序设计竞赛

【日】秋叶拓哉 岩田阳一 北川宜稔 著

巫泽俊 庄俊元 李津羽 译

陈 越 翁 恺 王 灿 审



人民邮电出版社

北 京

图书在版编目 (C I P) 数据

挑战程序设计竞赛 : 第2版 / (日) 秋叶拓哉, (日) 岩田阳一, (日) 北川宜稔著 ; 巫泽俊, 庄俊元, 李津羽译. -- 北京 : 人民邮电出版社, 2013. 6

(图灵程序设计丛书)

ISBN 978-7-115-32010-0

I. ①挑… II. ①秋… ②岩… ③北… ④巫… ⑤庄… ⑥李… III. ①程序设计 IV. ①TP311.1

中国版本图书馆CIP数据核字 (2013) 第115495号

Programming Contest Challenge Book, The Second edition

Copyright © 2010, 2012 Takuya Akiba, Yoichi Iwata, Masatoshi Kitagawa

Chinese translation rights in simplified characters arranged with Mynavi Corporation through Japan UNI Agency, Inc., Tokyo

本书中文简体字版由 Mynavi Corporation 授权人民邮电出版社独家出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

版权所有, 侵权必究。

内 容 提 要

本书对程序设计竞赛中的基础算法和经典问题进行了汇总, 分为准备篇、初级篇、中级篇与高级篇 4 章。作者结合自己丰富的参赛经验, 对严格筛选的 110 多道各类试题进行了由浅入深、由易及难的细致讲解, 并介绍了许多实用技巧。每章后附有习题, 供读者练习, 巩固所学。

本书适合程序设计人员、程序设计竞赛爱好者以及高校计算机专业师生阅读。

◆ 著 [日] 秋叶拓哉 岩田阳一 北川宜稔
译 巫泽俊 庄俊元 李津羽
责任编辑 乐 馨
执行编辑 徐 骞
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京 印刷

◆ 开本: 800×1000 1/16
印张: 26.5
字数: 626千字 2013年6月第1版
印数: 1-4 000册 2013年6月北京第1次印刷
著作权合同登记号 图字: 01-2012-6696号

定价: 79.00元

读者服务热线: (010)51095186转604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

广告经营许可证: 京崇工商广字第 0021 号

译者序

程序设计竞赛因其涉及的知识面广，比赛形式激烈有趣，吸引了越来越多的学生参与其中。参赛者不但可以从中锻炼算法设计能力，还能够提高代码编写能力。其中的佼佼者也受到了越来越多国际知名公司的重视和欢迎。

本书的几位作者是世界公认的顶尖选手，在竞赛和学术领域都取得了令人瞩目的成就。他们结合自己的专业知识和比赛经验，将自己的心得和技巧集结成书。

全书将不同的算法和例题按专题编排成小节，再将不同的小节由易到难分成四章，这样即便是初出茅庐的新手也不会有太大的阅读障碍。书中涵盖了在程序设计竞赛中会用到的大多数算法和技巧，并在附录中补充了书中未介绍但也比较有用的算法。在题材的安排上，作者取舍得当，主次分明，循序渐进，不以华而不实的奇技淫巧误导读者，又具有一定深度，相信即便是经验丰富的老将同样能从书中有所斩获。本书在结合例题进行讲解时，不是简单地堆砌问题和代码，而是注重引导读者更好地理解 and 运用算法来分析解决问题。对于正在学习数据结构与算法的读者而言，把它作为一本练习和拓展的参考书也是很好的选择。

本书在日本广受好评，还先后在台湾地区和韩国出版。近年来程序设计竞赛在亚洲发展很快，在中国大陆也出版了不少相关书籍，但鲜见高质量的佳作。所以，在读到此书时，我们非常惊喜，迫切希望中国大陆也能引进这样的好书。2012年初，我们通过作者的推特了解到了本书第二版的出版，一些前辈们踊跃翻译计算机专业书籍的经历也鼓舞了我们，让我们萌生了亲自翻译此书的念头并联系了图灵教育。非常幸运的是，图灵教育也正考虑引进此书，于是有了今天呈现在各位读者面前的简体中文版。

在翻译上，我们力求做到既尊重国内选手的习惯，又符合计算机专业的表述。在修正原书中的一些笔误的同时，加入了一些译者注，以方便国内读者理解。但由于译者水平有限，不足之处在所难免，还望读者多多包涵，并不吝提出意见和建议。

在翻译过程中，秋叶拓哉、岩田阳一和北川宜稔三位作者耐心地对我们的一些疑问和笔误给予了一一解答和确认。浙江大学的陈越、王灿和翁恺三位老师不但将我们领进了“快乐”竞赛的大门，还拨冗审阅了译稿并提出了宝贵的意见。网上不少同好也对本书的出版给予了关切和支持。在此谨对他们表示感谢。

巫泽俊 庄俊元 李津羽
2013年5月6日于浙江大学

前 言

如今，形形色色的程序设计竞赛层出不穷，听说过Google Code Jam、TopCoder、ACM-ICPC的读者恐怕不在少数。本书要介绍的正是这类以在规定时间内、又快又准地解决尽可能多的题目为目标的程序设计竞赛。

程序设计竞赛内涵丰富，即便是经验老道的程序员，要想在比赛中取得好成绩也绝非易事。要在程序设计竞赛中取胜，不仅需要运用灵活的想象和丰富的知识得出正确的算法，还需要一气呵成地实现并调试通过。

另一方面，程序设计竞赛对新手而言亦非遥不可及。为了让更多的参赛选手体会到比赛的乐趣，大多数比赛都会准备若干面向初学者的题目。另外，即便未能在比赛中取得好成绩，通过比赛，也能够使自己的能力得到有效的锻炼。最重要的是，大家能够享受到激烈的比赛带来的乐趣。


本书的作者们参加过众多程序设计竞赛，在平时的练习和学习中，也获得了各种各样的知识与技巧，本书将这些知识技巧总结成册，主要介绍算法及其在相关问题中的应用。本书依照由易及难的顺序对问题进行讲解，章节的编排也参考了主题的难易程度及其相互的联系，内容较多的主题则按难易程度划分为多个子主题分别介绍。各个主题由算法介绍和例题讲解穿插而成。

只要是具有编程基础知识的读者，均适合阅读本书。书中的源代码均用C++实现，不过只用到了其基本功能，所以即便读者不熟悉C++也不影响阅读。

【关于再版】

令人惊喜的是，本书的第1版受到了广大读者的高度评价，在此表示感谢。特别是一些并不热衷于程序设计竞赛的读者也购买了本书。这是因为通过本书不仅可以学到算法，更能学到其设计和运用的思想。这正是本书划时代的亮点。

本书第2版追加了计算几何、搜索减枝、分治法和字符串相关算法4个主题。此外还追加了方便读者加深理解的练习题，并为学有余力的读者列出了书中未涉及的拓展主题，进一步丰富了本书内容。



第 1 章

蓄势待发——准备篇

1.1 何谓程序设计竞赛

首先，让我们来说明一下程序设计竞赛到底是什么。

顾名思义，程序设计竞赛就是以程序设计为主题举办的竞赛。世界上有解题竞赛、性能竞赛、创意竞赛等各种各样的程序设计竞赛。本书主要介绍解题竞赛。

解题竞赛在开始时会告知选手题目的数量，选手的目标是解决其中尽可能多的题目。程序设计竞赛中题目的形式如下。

抽 签

你的朋友提议玩一个游戏：将写有数字的 n 个纸片放入口袋中，你可以从口袋中抽取 4 次纸片，每次记下纸片上的数字后都将其放回口袋中。如果这 4 个数字的和是 m ，就是你赢，否则就是你的朋友赢。你挑战了好几回，结果一次也没赢过，于是怒而撕破口袋，取出所有纸片，检查自己是否真的有赢的可能性。请你编写一个程序，判断当纸片上所写的数字是 k_1, k_2, \dots, k_n 时，是否存在抽取 4 次和为 m 的方案。如果存在，输出 *Yes*；否则，输出 *No*。

限制条件

- $1 \leq n \leq 50$
- $1 \leq m \leq 10^8$
- $1 \leq k_i \leq 10^8$

样例 1

输入

```
n = 3
m = 10
k = {1, 3, 5}
```

输出

Yes (例如4次抽取的结果是1、1、3、5，和就是10)

样例 2**输入**

```
n = 3
m = 9
k = {1, 3, 5}
```

输出

```
No (不存在和为9的抽取方案)
```

求解这个问题，可以编写如下程序。

```
#include <stdio>

const int MAX_N = 50;

int main() {
    int n, m, k[MAX_N];

    // 从标准输入读入
    scanf("%d %d", &n, &m);
    for (int i = 0; i < n; i++) {
        scanf("%d", &k[i]);
    }

    // 是否找到和为m的组合的标记
    bool f = false;

    // 通过四重循环枚举所有方案
    for (int a = 0; a < n; a++) {
        for (int b = 0; b < n; b++) {
            for (int c = 0; c < n; c++) {
                for (int d = 0; d < n; d++) {
                    if (k[a] + k[b] + k[c] + k[d] == m) {
                        f = true;
                    }
                }
            }
        }
    }

    // 输出到标准输出
    if (f) puts("Yes");
    else puts("No");

    return 0;
}
```

在许多比赛中，源代码一经提交就会自动编译并运行。预先准备好的输入文件将被重定向作为程

4 第1章 蓄势待发——准备篇

序的标准输入。通过判断程序对应的输出是否正确，来判断解答是否正确。

当然，程序的运行是有时间限制的。在大多数比赛中，运行时间限制在若干秒。一旦程序运行的时间超过了限制，程序就会被强行结束，当做不正确的解答处理。因此，在比赛中还必须考虑高效的解法。

例如，本题中有 $1 \leq n \leq 50$ 这个条件，像上面那样单纯的四重循环的程序，不用1秒就能得出答案。

但是，如果变成 $1 \leq n \leq 1000$ 又会怎样呢？四重循环的程序即便运行很多秒也不会结束，这将被判为不正确。不过，这道题有更为高效的解法，即便是 $1 \leq n \leq 1000$ 的情况，也能够按要求求解（将在1.6节中再讨论）。

由此，可以说程序设计竞赛是综合了以下两个要素的复合竞赛：

- 设计高效且正确的算法
- 正确地实现

并且，为了设计算法，

- 灵活的想象力
- 算法的基础知识

也是必不可少的。

1.2 最负盛名的程序设计竞赛

◆程序设计竞赛有着各种各样的形式，在此，我们来介绍其中最负盛名的几个。

1.2.1 世界规模的大赛——Google Code Jam (GCJ)

它是Google公司几乎每年都会举办的世界规模的程序设计竞赛，参赛者要在2~3小时内解决大约4道题。一旦从在线（Online）进行的几轮预选中胜出，就能够参加现场（Onsite）总决赛。该赛事的特点是，每道题都备有Small和Large两组输入数据。即便是难度系数较大的问题，只要输入规模足够小，依然可以简单地求解，这一形式深受广大参赛者的喜欢。另外，GCJ并不在服务器上自动执行程序，而是要求将源代码和本地执行的结果一同提交。

1.2.2 向高排名看齐！——TopCoder

TopCoder公司是一家策划并举办程序设计竞赛的公司，它举办的比赛涉及多个领域。其中之一就是算法（Algorithm）比赛，该赛事大致每周都以SRM（Single Round Match）的形式举办一场，其具有以下特点。

- (1) 在1小时15分钟的短时间内挑战3道题。
- (2) 提交的结果在比赛结束前是不知道的，整个过程中稍有失误，就会变成0分。
- (3) 在编码阶段（coding phase）结束后，还有一个挑战阶段（challenge phase）。该阶段可以查找别人代码中的漏洞。如果能够提供一组输入数据，使别人的程序返回错误的结果，就能得到额外的分数。

其中第3条是该赛事独一无二的特点^①，也是阅读别人代码的好机会。TopCoder还有一个深受大家喜欢的等级分系统（rating system），它会依据SRM的结果给参赛选手排名。另外，TopCoder还会举办一年一度的TCO（TopCoder Open）公开赛。一旦从在线进行的几轮预选中胜出，就能够参加在拉斯维加斯^②举办的总决赛。

^① 随后提到的Codeforces也参考TopCoder提供了类似但不完全一样的hack功能。——译者注

^② 最初几年，TCO的决赛地点都在拉斯维加斯，不过自2011年起，每年的决赛都选择在美国不同城市举办，如好莱坞、奥兰多和华盛顿。——译者注

1.2.3 历史最悠久的竞赛——ACM-ICPC

ACM-ICPC是由美国计算机协会（ACM）主办的、面向大学生的竞赛，也是历史最悠久的程序设计竞赛。这是一个三人一队的团队比赛，选手要在5个小时内解决大约10道题。因为比赛中三名选手共用一台电脑，题量又比其他赛事多，并且多是一些实现复杂的问题，所以团队配合显得非常重要。想要从日本参加该项赛事，首先要参加在线进行的国内预选赛，胜出后才能参加亚洲区域赛，取得前几名的好成绩后才能够参加世界总决赛。^①

1.2.4 面向中学生的信息学奥林匹克竞赛——JOI-IOI

信息学奥林匹克竞赛是学科奥林匹克竞赛的一种，是以初中生和高中生为参赛对象的设计竞赛。在日本，首先要参加日本信息学奥林匹克竞赛，取得优异成绩后，才能作为日本国家队选手参加国际信息学奥林匹克竞赛。^②其他比赛都需要尽可能快地解决尽可能多的问题，而信息学奥林匹克竞赛只要在规定时间内求解问题即可，成绩与所用时间无关，但是它相对其他比赛而言，求解每道题所花的时间要长得多。虽然是面向中学生的比赛，每年所出问题的难度却是非常高的。

1.2.5 通过网络自动评测——Online Judge（OJ）

在互联网上，有一些被称为Online Judge的系统，它们能够自动评测以往程序设计竞赛中的题目。利用该系统就可以练习了。另外，其中一些Online Judge也会定期举办自己的比赛，不妨去参加一下。在此列举几个有名的Online Judge。

- PKU Online Judge（POJ）——<http://poj.org/>
题库中有大量的题目。
- 会津大学Online Judge（AOJ）——<http://judge.u-aizu.ac.jp/onlinejudge/>
还包含日语题。
- Sphere Online Judge（SPOJ）——<http://www.spoj.pl/>
允许使用各种各样的编程语言。
- SGU Online Contester——<http://acm.sgu.ru/>
具有模拟参加历史比赛的虚拟赛功能。
- UVa Online Judge——<http://uva.onlinejudge.org/>
老字号Online Judge，经常举办比赛。
- Codeforces——<http://codeforces.com/>
与TopCoder一样定期举办比赛，又同其他网站一样不断维护历届题库。

^① 中国大陆的大学生若想晋级世界总决赛，通常也需要参加大陆任意赛区的网络预赛和现场区域赛并获得前几名。当然根据规则也有可能从亚洲其他地区获得出线权，其具体规则比较复杂并可能不断变化，大家可以从网上获得最新的规则。——译者注

^② 中国大陆的中学生首先要闯过全国联赛（NOIP）、全国竞赛（NOI）和国家队选拔赛（CTSC）三关，才能参加国际信息学奥林匹克竞赛（IOI）。——译者注

1.3 本书的使用方法

■在此，就本书所涉及的内容、使用方法及注意点做一下说明。

1.3.1 本书所涉及的内容

本书主要讲解程序设计竞赛中的经典问题和基础算法，并介绍便捷的实用技巧。如果仅仅是死记经典问题和基础算法，遇到难解的应用问题或是需要灵活想象力的问题时，仍然会难以下手。因此，为了加深理解，我们通过选自POJ的经典题和部分原创题来介绍实践中的例子。

另外，每章末尾都备有挑战GCJ中实战题目的小栏目，里面都是精选出来的题目。尽管要找到正确的解法恐怕不太容易，还是建议读者先自己试着多思考一下。在此基础上再阅读题解，能够得到更深刻的理解。

当然，在本书所介绍的解法之外，还会有更简洁或更高效的解法。大家不妨多试着去思考一下别的解法。

1.3.2 所用的编程语言

比赛中可用的编程语言各色各异，而C++在几乎所有比赛中都可用。它的运行速度快，库函数丰富，因而人气很高。本书选择C++作为所用的编程语言，并基本按照g++的规范来编写源代码。

1.3.3 题目描述的处理

在世界规模的大赛中，理所当然用英语来描述题目的。不过，因为题目描述中的英语不那么难，所用的单词往往也非常有限，所以很快就能习惯。当然，这不是英语考试，字典也是允许自由使用的。另外，其中有些比赛会针对日本选手提供日语版的题目描述。英语的阅读理解不是题目的关键，因此本书的题目都与最开始的例子一样用中文概述^①。

1.3.4 程序结构

在许多比赛中，程序都从标准输入按指定格式读入数据。输入并非问题的关键，所以本书的程序

^① 原书为用日语描述。——译者注

8 第1章 蓄势待发——准备篇

都假设输入数据已经由main函数读入并保存在全局变量中，再通过调用solve函数来求解。例如对于最初的例子，程序将变成这样。

```
// 读入输入数据后保存在这里
int n, m, k[MAX_N];

void solve() {
    bool f = false;

    for (int a = 0; a < n; a++) {
        for (int b = 0; b < n; b++) {
            for (int c = 0; c < n; c++) {
                for (int d = 0; d < n; d++) {
                    if (k[a] + k[b] + k[c] + k[d] == m) {
                        f = true;
                    }
                }
            }
        }
    }

    if (f) puts("Yes");
    else puts("No");
}
```

1.3.5 练习题

每章末尾都会介绍与本章所涉及主题相关的题目。请利用它们来加深理解、巩固知识、培养实践能力。各个主题下的题目大致是按照难易程度排列的，其中亦包含非常难的应用问题。

1.3.6 读透本书后更上一层楼的练习方法

独自练习提高时，不妨同时使用Online Judge和TopCoder的Practice Room。特别是TopCoder，既提供了解题教程，又可以阅读别人的代码，当你无论如何都想不到解法时，还可以把它们作为参考，因而在此推荐。



第 2 章

初出茅庐——初级篇

2.1 最基础的“穷竭搜索”

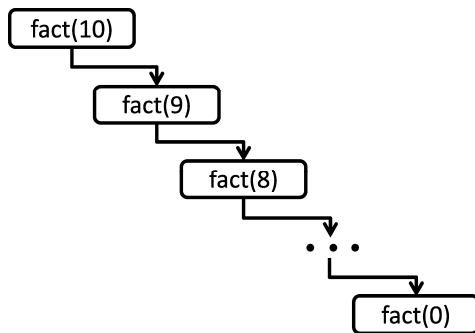
穷竭搜索是将所有的可能性罗列出来，在其中寻找答案的方法。这里我们主要介绍深度优先搜索和广度优先搜索这两种方法。

2.1.1 递归函数

在一个函数中再次调用该函数自身的行为叫做递归，这样的函数被称作递归函数。例如，我们要编写一个计算阶乘的函数`int fact(int n)`，当然，用循环来实现也是可以的。但是根据阶乘的递推式 $n! = n \times (n-1)!$ ，我们可以写成如下形式：

```
int fact(int n) {  
    if (n == 0) return 1;  
    return n * fact(n - 1);  
}
```

在编写一个递归函数时，函数的停止条件是必须存在的。在刚刚的例子中，当 $n=0$ 时`fact`并不是继续调用自身，而是直接返回1。如果没有这一条件存在，函数就会无限地递归下去，程序就会失控崩溃了。

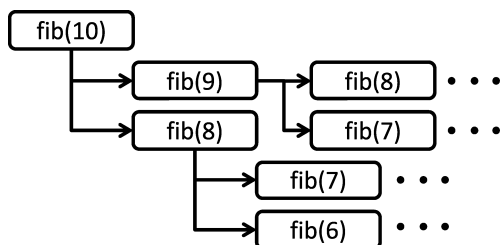


fact递归的过程

我们再来试试编写计算斐波那契数列的函数`int fib(int n)`。斐波那契数列的定义是 $a_0=0$ 、 $a_1=1$ 以及 $a_n=a_{n-1}+a_{n-2}$ ($n>1$)。这里，初项的条件就对应了递归的终止条件。数列的定义直接写成函数就可以了。

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

实际使用这个函数时，即使是求 `fib(40)` 这样的 n 较小时的结果，也要花费相当长的时间。这是因为这个函数在递归时，会像下图一样按照指数级别扩展开来。



fib(10) 递归的过程

在斐波那契数列中，如果 `fib(n)` 的 n 是一定的，无论多少次调用都会得到同样的结果。因此如果计算一次之后，用数列将结果存储起来，便可优化之后的计算。（上图中）`fib(10)` 被调用时同样的 n 被计算了很多次，因此可以获得很大的优化空间。这种方法是出于记忆化搜索或者动态规划的想法，之后我们会介绍。

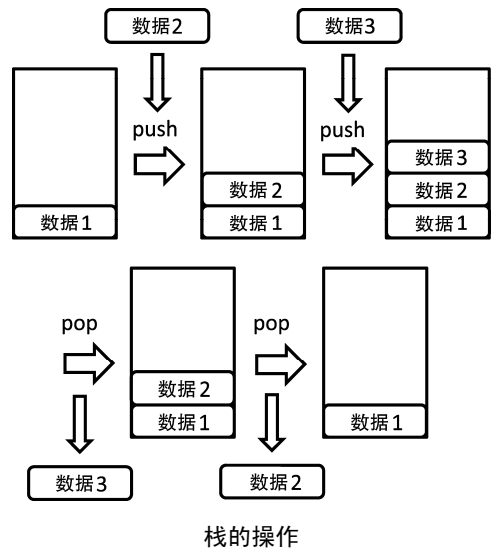
```
int memo[MAX_N + 1];

int fib(int n) {
    if (n <= 1) return n;
    if (memo[n] != 0) return memo[n];
    return memo[n] = fib(n - 1) + fib(n - 2);
}
```

2.1.2 栈

栈（Stack）是支持 `push` 和 `pop` 两种操作的数据结构。`push` 是在栈的顶端放入一组数据的操作。反之，`pop` 是从其顶端取出一组数据的操作。因此，最后进入栈的一组数据可以最先被取出（这种行为被叫做 LIFO: Last In First Out，即后进先出）。

通过使用数组或者列表等结构可以很容易实现栈，不过 C++、Java 等编程语言的标准库已经为我们准备好了这一常用结构，在比赛中需要时不妨使用它们。C++ 的标准库中，`stack::pop` 完成的仅仅是移除最顶端的数据。如果要访问最顶端的数据，需要使用 `stack::top` 函数（这个操作通常也被称为 `peek`）。



函数调用的过程是通过使用栈实现的。因此，递归函数的递归过程也可以改用栈上的操作来实现。现实中需要如此改写的场合并不多，不过作为使用栈的练习试试看也是不错的。以下是使用stack的例子：

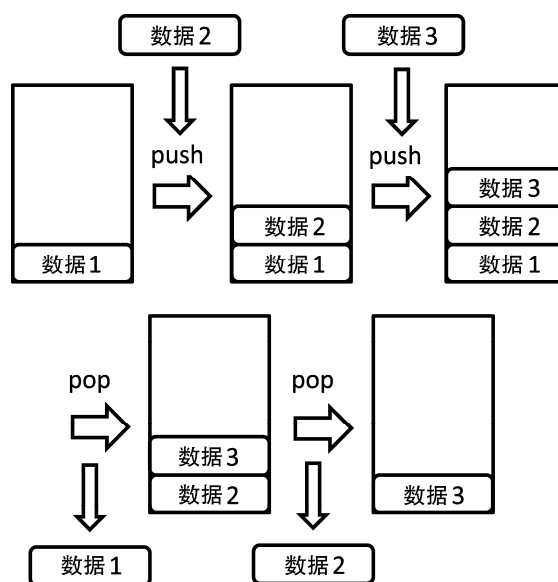
```
#include <stack>
#include <cstdio>

using namespace std;

int main() {
    stack<int> s;           // 声明存储int类型数据的栈
    s.push(1);              // {} → {1}
    s.push(2);              // {1} → {1,2}
    s.push(3);              // {1,2} → {1,2,3}
    printf("%d\n", s.top()); // 3
    s.pop();                // 从栈顶移除 {1,2,3}→{1,2}
    printf("%d\n", s.top()); // 2
    s.pop();                // {1,2} → {1}
    printf("%d\n", s.top()); // 1
    s.pop();                // {1} → {}
    return 0;
}
```

2.1.3 队列

队列（Queue）与栈一样支持push和pop两个操作。但与栈不同的是，pop完成的不是取出最顶端的元素，而是取出最底端的元素。也就是说最初放入的元素能够最先被取出（这种行为被叫做FIFO: First In First Out，即先进先出）。



队列的操作

如同栈一样，C++、Java等的标准库也预置了队列。Java与C++中的函数的名称与用途稍有不同，因此使用时要注意。此外，在C++中`queue::front`是用来访问最底端数据的函数。以下是使用`queue`的例子：

```
#include <queue>
#include <cstdio>

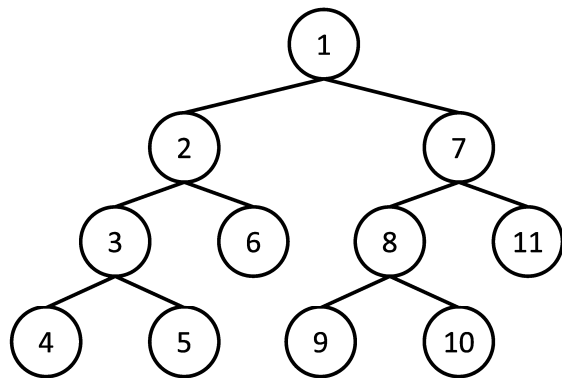
using namespace std;

int main() {
    queue<int> que;           // 声明存储int类型数据的队列
    que.push(1);              // {} → {1}
    que.push(2);              // {1} → {1, 2}
    que.push(3);              // {1, 2} → {1, 2, 3}
    printf("%d\n", que.front()); // 1
    que.pop();                // 从队尾移除 {1, 2, 3} → {2, 3}
    printf("%d\n", que.front()); // 2
    que.pop();                // {2, 3} → {3}
    printf("%d\n", que.front()); // 3
    que.pop();                // {3} → {}
    return 0;
}
```

2.1.4 深度优先搜索

深度优先搜索（DFS，Depth-First Search）是搜索的手段之一。它从某个状态开始，不断地转移

状态直到无法转移，然后回退到前一步的状态，继续转移到其他状态，如此不断重复，直至找到最终的解。例如求解数独，首先在某个格子内填入适当的数字，然后再继续在下一个格子内填入数字，如此继续下去。如果发现某个格子无解了，就放弃前一个格子上选择的数字，改用其他可行的数字。根据深度优先搜索的特点，采用递归函数实现比较简单。



状态转移的顺序

我们来试着解答一下下面的题目：

部分和问题

给定整数 $a_1、a_2、\cdots、a_n$ ，判断是否可以从中选出若干数，使它们的和恰好为 k 。

⚠ 限制条件

- $1 \leq n \leq 20$
- $-10^8 \leq a_i \leq 10^8$
- $-10^8 \leq k \leq 10^8$

样例 1

输入

n=4
a={1,2,4,7}
k=13

输出

Yes (13 = 2 + 4 + 7)

样例 2

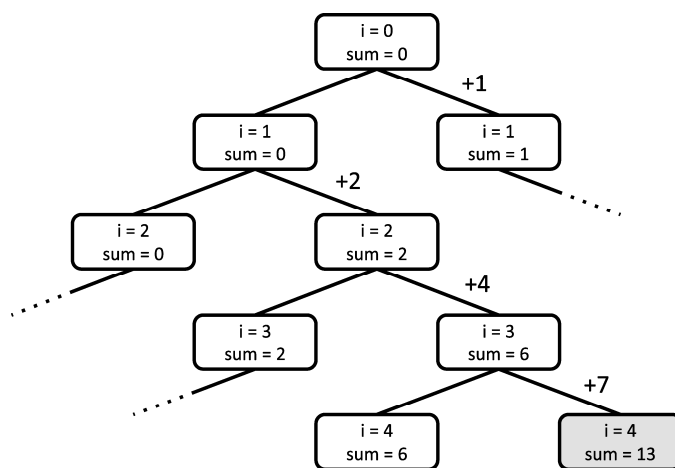
输入

```
n=4
a={1,2,4,7}
k=15
```

输出

No

从 a_1 开始按顺序决定每个数加或不加，在全部 n 个数都决定后再判断它们的和是不是 k 即可。因为状态数是 2^{n+1} ，所以复杂度是 $O(2^n)$ 。如何实现这个搜索，请参见下面的代码。注意 a 的下标与题目描述中的下标偏移了1。在程序中使用的是0起始的下标规则，题目描述中则是1开始的，这一点要注意避免搞混。



状态转移的样子

```
// 输入
int a[MAX_N];
int n, k;

// 已经从前i项得到了和sum，然后对于i项之后的进行分支
bool dfs(int i, int sum) {
    // 如果前n项都计算过了，则返回sum是否与k相等
    if (i == n) return sum == k;

    // 不加上a[i]的情况
    if (dfs(i + 1, sum)) return true;

    // 加上a[i]的情况
```

32 第2章 初出茅庐——初级篇

```
    if (dfs(i + 1, sum + a[i])) return true;

    // 无论是否加上a[i]都不能凑成k就返回false
    return false;
}

void solve() {
    if (dfs(0, 0)) printf("Yes\n");
    else printf("No\n");
}
```

深度优先搜索从最开始的状态出发,遍历所有可以到达的状态。由此可以对所有的状态进行操作,或者列举出所有的状态。

Lake Counting (POJ No.2386)

有一个大小为 $N \times M$ 的园子,雨后积起了水。八连通的积水被认为是连接在一起的。请求出园子里总共有多少水洼? (八连通指的是下图中相对 W 的*的部分)

```
***
*W*
***
```

⚠ 限制条件

- $N, M \leq 100$

样例

输入

```
N=10, M=12
园子如下图 ('W'表示积水, '.'表示没有积水)
W.....WW.
.WWW.....WWW
....WW...WW.
.....WW.
.....W..
..W.....W..
.W.W.....WW.
W.W.W.....W.
.W.W.....W.
..W.....W.
```

输出

3

从任意的w开始，不停地把邻接的部分用'.'代替。1次DFS后与初始的这个w连接的所有w就都被替换成了'.'，因此直到图中不再存在w为止，总共进行DFS的次数就是答案了。8个方向共对应了8种状态转移，每个格子作为DFS的参数至多被调用一次，所以复杂度为 $O(8 \times N \times M) = O(N \times M)$ 。

```
// 输入
int N, M;
char field[MAX_N][MAX_M + 1]; // 园子

// 现在位置(x,y)
void dfs(int x, int y) {
    // 将现在所在位置替换为.
    field[x][y] = '.';

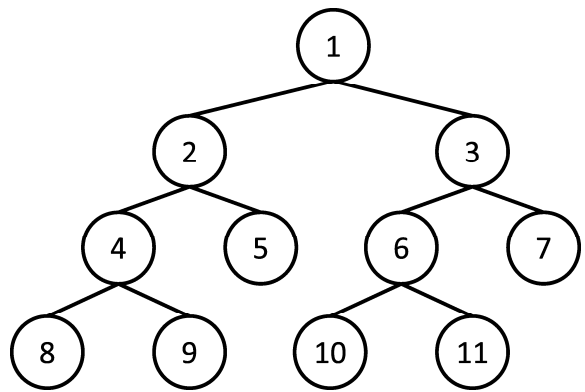
    // 循环遍历移动的8个方向
    for (int dx = -1; dx <= 1; dx++) {
        for (int dy = -1; dy <= 1; dy++) {
            // 向x方向移动dx, 向y方向移动dy, 移动的结果为 (nx,ny)
            int nx = x + dx, ny = y + dy;
            // 判断(nx,ny)是不是在园子内, 以及是否有积水
            if (0 <= nx && nx < N && 0 <= ny && ny < M && field[nx][ny] == 'W') dfs(nx, ny);
        }
    }
    return ;
}

void solve() {
    int res = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (field[i][j] == 'W') {
                // 从有W的地方开始dfs
                dfs(i, j);
                res++;
            }
        }
    }
    printf("%d\n", res);
}
```

2.1.5 宽度优先搜索

宽度优先搜索（BFS, Breadth-First Search）也是搜索的手段之一。它与深度优先搜索类似，从某个状态出发探索所有可以到达的状态。

与深度优先搜索的不同之处在于搜索的顺序，宽度优先搜索总是先搜索距离初始状态近的状态。也就是说，它是按照开始状态→只需1次转移就可以到达的所有状态→只需2次转移就可以到达的所有状态→……这样的顺序进行搜索。对于同一个状态，宽度优先搜索只经过一次，因此复杂度为 $O(\text{状态数} \times \text{转移的方式})$ 。



状态转移的顺序

深度优先搜索（隐式地）利用了栈进行计算，而宽度优先搜索则利用了队列。搜索时首先将初始状态添加到队列里，此后从队列的最前端不断取出状态，把从该状态可以转移到的状态中尚未访问过的部分加入队列，如此往复，直至队列被取空或找到了问题的解。通过观察这个队列，我们可以就知道所有的状态都是按照距初始状态由近及远的顺序被遍历的。

迷宫的最短路径

给定一个大小为 $N \times M$ 的迷宫。迷宫由通道和墙壁组成，每一步可以向邻接的上下左右四格的通道移动。请求出从起点到终点所需的最小步数。请注意，本题假定从起点一定可以移动到终点。

限制条件

- $N, M \leq 100$

样例

输入

$N=10, M=10$ （迷宫如下图所示。'#', '.', 'S', 'G'分别表示墙壁、通道、起点和终点）

```
#S#####.#
.....#..#
.###.###.
.#.....
##.##.####
....#....#
.#####.#
....#.....
.####.###.
....#...G#
```

输出

22

宽度优先搜索按照距开始状态由近及远的顺序进行搜索，因此可以很容易地用来求最短路径、最少操作之类问题的答案。这个问题中，状态仅仅是目前所在位置的坐标，因此可以构造成`pair`或者编码成`int`来表达状态。当状态更加复杂时，就需要封装成一个类来表示状态了。转移的方式为四方向移动，状态数与迷宫的大小是相等的，所以复杂度是 $O(4 \times N \times M) = O(N \times M)$ 。

宽度优先搜索中，只要将已经访问过的状态用标记管理起来，就可以很好地做到由近及远的搜索。这个问题中由于要求最短距离，不妨用`d[N][M]`数组把最短距离保存起来。初始时用充分大的常数`INF`来初始化它，这样尚未到达的位置就是`INF`，也就同时起到了标记的作用。

虽然到达终点时就会停止搜索，可如果继续下去直到队列为空的话，就可以计算出到各个位置的最短距离。此外，如果搜索到最后，`d`依然为`INF`的话，便可得知这个位置就是无法从起点到达的位置。

在今后的程序中，使用像`INF`这样充分大的常数的情况还很多。不把`INF`当作例外，而是直接参与普通运算的情况也很常见。这种情况下，如果`INF`过大就可能带来溢出的危险。

假设 $INF = 2^{31} - 1$ 。例如想用`d[nx][ny] = min(d[nx][ny], d[x][y] + 1)`来更新`d[nx][ny]`，就会发生 $INF + 1 = -2^{31}$ 的情况。这一问题中`d[x][y]`总不等于`INF`，所以没有问题。但是为了防止这样的问题，一般会将`INF`设为放大2~4倍也不会溢出的大小（可参考2.5节Floyd-Warshall算法等）。

因为要向4个不同方向移动，用`dx[4]`和`dy[4]`两个数组来表示四个方向向量。这样通过一个循环就可以实现四方向移动的遍历。

```
const int INF = 100000000;

// 使用pair表示状态时，使用typedef会更加方便一些
typedef pair<int, int> P;

// 输入
char maze[MAX_N][MAX_M + 1]; // 表示迷宫的字符串的数组
int N, M;
int sx, sy; // 起点坐标
int gx, gy; // 终点坐标

int d[MAX_N][MAX_M]; // 到各个位置的最短距离的数组

// 4个方向移动的向量
int dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1};

// 求从(sx, sy)到(gx, gy)的最短距离
// 如果无法到达，则是INF
int bfs() {
```



```

queue<P> que;
// 把所有的位置都初始化为INF
for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++) d[i][j] = INF;
// 将起点加入队列，并把这一地点的距离设置为0
que.push(P(sx, sy));
d[sx][sy] = 0;

// 不断循环直到队列的长度为0
while (que.size()) {
    // 从队列的最前端取出元素
    P p = que.front(); que.pop();
    // 如果取出的状态已经是终点，则结束搜索
    if (p.first == gx && p.second == gy) break;

    // 四个方向的循环
    for (int i = 0; i < 4; i++) {
        // 移动之后的位置记为 (nx, ny)
        int nx = p.first + dx[i], ny = p.second + dy[i];

        // 判断是否可以移动以及是否已经访问过 (d[nx][ny] != INF即为已经访问过)
        if (0 <= nx && nx < N && 0 <= ny && ny < M && maze[nx][ny] != '#' &&
            d[nx][ny] == INF) {
            // 可以移动的话，则加入到队列，并且到该位置的距离确定为到p的距离+1
            que.push(P(nx, ny));
            d[nx][ny] = d[p.first][p.second] + 1;
        }
    }
}
return d[gx][gy];
}

void solve() {
    int res = bfs();
    printf("%d\n", res);
}

```

宽度优先搜索与深度优先搜索一样，都会生成所有能够遍历到的状态，因此需要对所有状态进行处理时使用宽度优先搜索也是可以的。但是递归函数可以很简短地编写，而且状态的管理也更简单，所以大多数情况下还是用深度优先搜索实现。反之，在求取最短路时深度优先搜索需要反复经过同样的状态，所以此时还是使用宽度优先搜索为好。

宽度优先搜索会把状态逐个加入队列，因此通常需要与状态数成正比的内存空间。反之，深度优先搜索是与最大的递归深度成正比的。一般与状态数相比，递归的深度并不会太大，所以可以认为深度优先搜索更加节省内存。

此外，也有采用与宽度优先搜索类似的状态转移顺序，并且注重节约内存占用的迭代加深深度优先搜索（IDDFS，Iterative Deepening Depth-First Search）。IDDFS是一种在最开始将深度优先搜索的递归次数限制在1次，在找到解之前不断增加递归深度的方法。这种方法会在4.5节详细介绍。

2.1.6 特殊状态的枚举

虽然生成可行解空间多数采用深度优先搜索，但在状态空间比较特殊时其实可以很简短地实现。比如，C++的标准库中提供了`next_permutation`这一函数，可以把 n 个元素共 $n!$ 种不同的排列生成出来。又或者，通过使用位运算，可以枚举从 n 个元素中取出 k 个的共 C_n^k 种状态或是某个集合中的全部子集等。3.2节将介绍如何利用位运算枚举状态。

```

bool used[MAX_N];
int perm[MAX_N];

// 生成{0,1,2,3,4,...,n-1}的n!种排列

void permutation1(int pos, int n) {
    if (pos == n) {
        /*
         * 这里编写需要对perm进行的操作
         */
        return ;
    }

    // 针对perm的第pos个位置，究竟使用0~n-1中的哪一个进行循环
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            perm[pos] = i;
            // i已经被使用了，所以把标志设置为true
            used[i] = true;
            permutation1(pos + 1, n);
            // 返回之后把标志复位
            used[i] = false;
        }
    }
    return ;
}

#include <algorithm>

// 即使有重复的元素也会生成所有的排列
// next_permutation是按照字典序来生成下一个排列的
int perm2[MAX_N];

void permutation2(int n) {
    for (int i = 0; i < n; i++) {
        perm2[i] = i;
    }
    do {
        /*
         * 这里编写需要对perm2进行的操作
         */
    } while (next_permutation(perm2, perm2 + n));
    // 所有的排列都生成后，next_permutation会返回false
    return ;
}

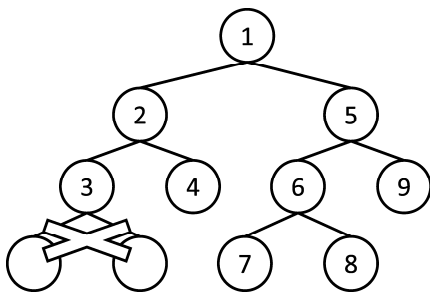
```

2.1.7 剪枝

顾名思义，穷竭搜索会把所有可能的解都检查一遍，当解空间非常大时，复杂度也会相应变大。比如 n 个元素进行排列时状态数总共有 $n!$ 个，复杂度也就成了 $O(n!)$ 。这样的话，即使 $n=15$ 计算也很难较早终止。这里简单介绍一下此类情形要如何进行优化。

深度优先搜索时，有时早已很明确地知道从当前状态无论如何转移都不会存在解。这种情况下，不再继续搜索而是直接跳过，这一方法被称作剪枝。

我们回想一下深度优先搜索的例题“部分和问题”。这个问题中的限制条件如果变为 $0 \leq a_i \leq 10^8$ ，那么在递归中只要sum超过k了，此后无论选择哪些数都不可能让sum等于k，所以此后没有必要继续搜索。



剪枝的情况


关于更多更高级的搜索手段，我们会在4.5节进行详细介绍。

专栏 栈内存和堆内存

调用函数时，主调的函数所拥有的局部变量等信息需要存储在特定的内存区域。这个区域被称作栈内存区。另一方面，利用 `new` 或者 `malloc` 进行分配的内存区域被称为堆内存。

栈内存存在程序启动时被统一分配，此后不能再扩大。由于这一区域有上限，所以函数的递归深度也有上限。虽然与函数中定义的局部变量的数目有关，不过一般情况下 C 和 C++ 中进行上万次的递归是可以的。在 Java 中，在执行程序时可以用参数指定栈的大小。不同的程序设计竞赛所采用的设置各有不同，建议大家预先进行确认。GCJ 的话，程序是在自己的机器上执行的，所以可以自行设置参数。

全局变量被保存在堆内存区。通常不推荐使用全局变量，但是在程序设计竞赛中，由于函数通常不是那么多，并且常常会有多个函数访问同一个数组，因此利用全局变量就很方便。此外，有时必须要申请巨大的数组，与放置在栈内存上相比，将其放置在堆内存上可以减少栈溢出的危险。同时，通常只需定义满足最大需要的数列大小，但如果再额外定义大一些，能很好地避免粗心导致的诸如忘记保留字符串末尾的 `'\0'` 的空间之类的漏洞。



第 4 章

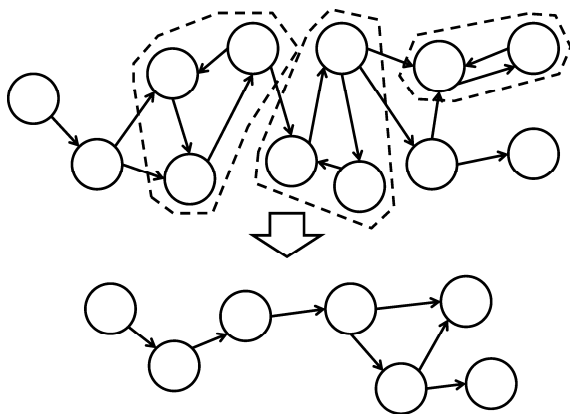
登峰造极——高级篇

4.3 成为图论大师之路

图是非常有用的数据结构，除了在第2章已经介绍的算法外，还有各种各样的相关算法。在此，我们主要讨论强连通分量分解和最近公共祖先等问题。

4.3.1 强连通分量分解

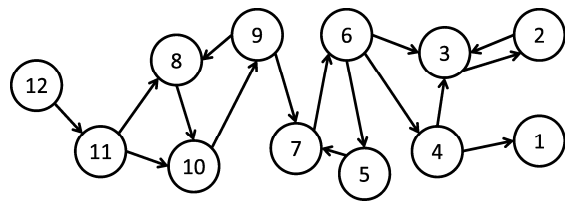
对于一个有向图顶点的子集 S ，如果在 S 内任取两个顶点 u 和 v ，都能找到一条从 u 到 v 的路径，那么就称 S 是强连通的。如果在强连通的顶点集合 S 中加入其他任意顶点集合后，它都不再是强连通的，那么就称 S 是原图的一个强连通分量（SCC: Strongly Connected Component）。任意有向图都可以分解成若干不相交的强连通分量，这就是强连通分量分解。把分解后的强连通分量缩成一个顶点，就得到了一个DAG（有向无环图）。



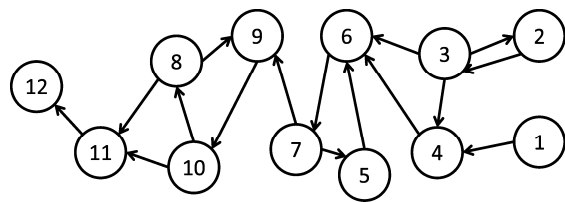
虚线包围的部分构成一个强连通分量

强连通分量分解可以通过两次简单的DFS实现。第一次DFS时，选取任意顶点作为起点，遍历所有尚未访问过的顶点，并在回溯前给顶点标号（post order，后序遍历）。对剩余的未访问过的顶点，不断重复上述过程。

完成标号后，越接近图的尾部（搜索树的叶子），顶点的标号越小。第二次DFS时，先将所有边反向，然后以标号最大的顶点为起点进行DFS。这样DFS所遍历的顶点集合就构成了一个强连通分量。之后，只要还有尚未访问的顶点，就从中选取标号最大的顶点不断重复上述过程。

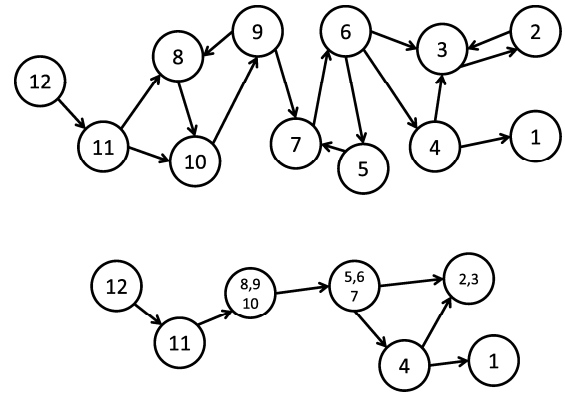


后续遍历的例子。根据搜索顺序的不同，标号结果也可能不同



反向后的图

正如前文所述，我们可以将强连通分量缩点并得到DAG。此时可以发现，标号最大的节点就属于DAG头部（搜索树的根）的强连通分量。因此，将边反向后，就不能沿边访问到这个强连通分量以外的顶点。而对于强连通分量内的其他顶点，其可达性不受边反向的影响，因此在第二次DFS时，我们可以遍历一个强连通分量里的所有顶点。



边反向后，从8、9、10号顶点只能到达其头部方向的顶点11和12

该算法只进行了两次DFS，因而总的复杂度是 $O(|V|+|E|)$ 。

```
int V; // 顶点数
vector<int> G[MAX_V]; // 图的邻接表表示
vector<int> rG[MAX_V]; // 把边反向后的图
vector<int> vs; // 后序遍历顺序的顶点列表
bool used[MAX_V]; // 访问标记
int cmp[MAX_V]; // 所属强连通分量的拓扑序
```

```

void add_edge(int from, int to) {
    G[from].push_back(to);
    rG[to].push_back(from);
}

void dfs(int v) {
    used[v] = true;
    for (int i = 0; i < G[v].size(); i++) {
        if (!used[G[v][i]]) dfs(G[v][i]);
    }
    vs.push_back(v);
}

void rdfs(int v, int k) {
    used[v] = true;
    cmp[v] = k;
    for (int i = 0; i < rG[v].size(); i++) {
        if (!used[rG[v][i]]) rdfs(rG[v][i], k);
    }
}

int scc() {
    memset(used, 0, sizeof(used));
    vs.clear();
    for (int v = 0; v < V; v++) {
        if (!used[v]) dfs(v);
    }
    memset(used, 0, sizeof(used));
    int k = 0;
    for (int i = vs.size() - 1; i >= 0; i--) {
        if (!used[vs[i]]) rdfs(vs[i], k++);
    }
    return k;
}

```

Popular Cows (POJ No.2186)

每头牛都想成为牛群中的红人。给定 N 头牛的牛群和 M 个有序对 (A, B) 。 (A, B) 表示牛 A 认为牛 B 是红人。该关系具有传递性, 所以如果牛 A 认为牛 B 是红人, 牛 B 认为牛 C 是红人, 那么牛 A 也认为牛 C 是红人。不过, 给定的有序对中可能包含 (A, B) 和 (B, C) , 但不包含 (A, C) 。求被其他所有牛认为是红人的牛的总数。

⚠ 限制条件

- $1 \leq N \leq 10000$
- $1 \leq M \leq 50000$
- $1 \leq A, B \leq N$

样例

输入

```
N = 3
M = 3
(A, B) = {(1, 2), (2, 1), (2, 3)}
```

输出

```
1 (3号牛)
```

考虑以牛为顶点的有向图，对每个有序对 (A, B) 连一条从 A 到 B 的有向边。那么，被其他所有牛认为是红人的牛对应的顶点，也就是从其他所有顶点都可达的顶点。虽然这可以通过从每个顶点出发搜索求得，但总的复杂度却是 $O(NM)$ ，是不可行的，必须要考虑更为高效的算法。

假设有两头牛 A 和 B 都被其他所有牛认为是红人。那么显然， A 被 B 认为是红人， B 也被 A 认为是红人，即存在一个包含 A 、 B 两个顶点的圈，或者说， A 、 B 同属于一个强连通分量。反之，如果一头牛被其他所有牛认为是红人，那么其所属的强连通分量内的所有牛都被其他所有牛认为是红人。由此，我们把图进行强连通分量分解后，至多有一个强连通分量满足题目的条件。而按前面介绍的算法进行强连通分量分解时，我们还能够得到各个强连通分量拓扑排序后的顺序，唯一可能成为解的只有拓扑序最后的强连通分量。所以在最后，我们只要检查这个强连通分量是否从所有顶点可达就好了。该算法的复杂度为 $O(N+M)$ ，足以在时限内解决原题。

```
// 输入
int N, M;
int A[MAX_M], B[MAX_M];

void solve() {
    V = N;
    for (int i = 0; i < M; i++) {
        add_edge(A[i] - 1, B[i] - 1);
    }
    int n = scc();

    // 统计备选解的个数
    int u = 0, num = 0;
    for (int v = 0; v < V; v++) {
        if (cmp[v] == n - 1) {
            u = v;
            num++;
        }
    }

    // 检查是否从所有点可达
    memset(used, 0, sizeof(used));
    rdfs(u, 0); // 重用强连通分量分解的代码
```



```

for (int v = 0; v < V; v++) {
    if (!used[v]) {
        // 从该点不可达
        num = 0;
        break;
    }
}

printf("%d\n", num);
}

```

4.3.2 2-SAT

给定一个布尔方程，判断是否存在一组布尔变量的真值指派使整个方程为真的问题，被称为布尔方程的可满足性问题（SAT）。SAT问题是NP完全的，但对于满足一定限制条件的SAT问题，还是能够有效求解的。我们将下面这种布尔方程称为合取范式。

$$(a \vee b \vee \cdots) \wedge (c \vee d \vee \cdots) \wedge \cdots$$

其中 a, b, \cdots 称为文字，它是一个布尔变量或其否定。像 $(a \vee b \vee \cdots)$ 这样用 \vee 连接的部分称为子句。如果合取范式的每个子句中的文字个数都不超过两个，那么对应的SAT问题又称为2-SAT问题。

■ 2-SAT布尔公式的例子

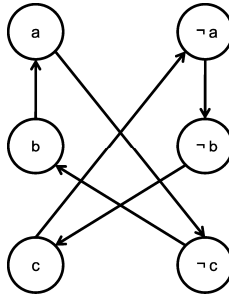
- $(a \vee b) \wedge \neg a$ 令 a 为假而 b 为真，则可以满足
- $(a \vee \neg b) \wedge (b \vee c) \wedge (\neg c \vee \neg a)$ 令 a 和 b 为真而 c 为假，则可以满足
- $(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$ 无法满足

利用强连通分量分解，可以在布尔公式子句数的线性时间内解决2-SAT问题。首先，利用 \Rightarrow （蕴涵）将每个子句 $(a \vee b)$ 改写成等价形式 $(\neg a \Rightarrow b \wedge \neg b \Rightarrow a)$ 。这样原布尔公式就变成了把 $a \Rightarrow b$ 形式的布尔公式用 \wedge 连接起来的形式。对每个布尔变量 x ，构造两个顶点分别代表 x 和 $\neg x$ ，以 \Rightarrow 关系为边建立有向图。此时，如果图上的 a 点能够到达 b 点的话，就表示当 a 为真时 b 也一定为真。因此，该图中同一个强连通分量中所含的所有文字的布尔值均相同。

如果存在某个布尔变量 x ， x 和 $\neg x$ 均在同一个强连通分量中，则显然无法令整个布尔公式的值为真。反之，如果不存在这样的布尔变量，那么对于每个布尔变量 x ，让

x 所在的强连通分量的拓扑序在 $\neg x$ 所在的强连通分量之后 $\Leftrightarrow x$ 为真

就是使得该公式的值为真的一组合适的布尔变量赋值。



$(a \vee \neg b) \wedge (b \vee c) \wedge (\neg c \vee \neg a)$ 所对应的图

```

int main() {
    // 布尔公式为  $(a \vee \neg b) \wedge (b \vee c) \wedge (\neg c \vee \neg a)$  时
    // 构造6个顶点, 分别对应a、b、c、¬a、¬b、¬c。
    V = 6;

    //  $a \vee \neg b$  转成  $\neg a \Rightarrow \neg b \wedge b \Rightarrow a$ 
    add_edge(3, 4); // 从¬a连一条到¬b的边
    add_edge(1, 0); // 从b连一条到a的边
    //  $b \vee c$  转成  $\neg b \Rightarrow c \wedge \neg c \Rightarrow b$ 
    add_edge(4, 2); // 从¬b连一条到c的边
    add_edge(5, 1); // 从¬c连一条到b的边
    //  $\neg c \vee \neg a$  转成  $c \Rightarrow \neg a \wedge a \Rightarrow \neg c$ 
    add_edge(2, 3); // 从c连一条到¬a的边
    add_edge(0, 5); // 从a连一条到¬c的边

    // 进行强连通分量分解
    scc();

    // 判断是否x和¬x在不同的强连通分量中
    for (int i = 0; i < 3; i++) {
        if (cmp[i] == cmp[3 + i]) {
            printf("NO");
            return 0;
        }
    }

    // 如果可满足, 则给出一组解
    printf("YES\n");
    for (int i = 0; i < 3; i++) {
        if (cmp[i] > cmp[3 + i]) {
            printf("true\n");
        } else {
            printf("false\n");
        }
    }

    return 0;
}

```

Priest John's Busiest Day (POJ No.3683)

约翰是街区里唯一的神父。假设有 N 对新人打算在同一天举行结婚仪式。第 i 对新人的结婚仪式的时间为 S_i 到 T_i ，在其仪式开始时或是结束时需要进行一个用时 D_i 的特别仪式（也就是从 S_i 到 S_i+D_i 或是从 T_i-D_i 到 T_i ），该特别仪式需要神父在场。请判断是否可以通过合理安排每个特别仪式在开始还是结束时进行，从而保证神父能够出席所有的特别仪式。如果可能的话，请输出出席各个特别仪式的时间。当然，神父不可能同时出席多个特别仪式。不过神父前往仪式的途中所花费的时间可以忽略不计，神父可以在出席完一个特别仪式后，立刻出席另一个开始时间与其结束时间相等的特别仪式。

⚠ 限制条件

- $1 \leq N \leq 1000$

样例

输入

```
N = 2
(S, T, D) = {(08:00, 09:00, 30), (08:15, 09:00, 20)}
```

输出

```
YES
08:00 08:30
08:40 09:00
```

对于每个结婚仪式 i ，只有在开始或结束时进行特别仪式两种选择。因此可以定义变量 x_i

x_i 为真 \Leftrightarrow 在开始时进行特别仪式

这样，对于结婚仪式 i 和 j ，如果 $S_i \sim S_i+D_i$ 和 $S_j \sim S_j+D_j$ 冲突，就有 $\neg x_i \vee \neg x_j$ 为真。对于开始和结束、结束和开始、结束和结束等三种情况，也可以得到类似的条件。于是，要保证所有特别仪式的时间不冲突，只要考虑将这所有的子句用 \wedge 连接起来所得到的布尔公式就好了。例如，对于输入样例，可以的到布尔公式

$$(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$$

而当 x_1 为真而 x_2 为假时，其值为真。这样，我们就把原问题转为了 2-SAT 问题。接下来只要进行强连通分量分解并判断是否有使得布尔公式值为真的一组布尔变量赋值就好了。

```
// 输入
int N;
int S[MAX_N], T[MAX_N], D[MAX_N]; // S和T是换算成分钟后的时间
```

```

void solve() {
    // 0~N-1: x_i
    // N~2N-1: ¬x_i
    V = N * 2;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < i; j++) {
            if (min(S[i] + D[i], S[j] + D[j]) > max(S[i], S[j])) {
                // x_i⇒¬x_j、x_j⇒¬x_i
                add_edge(i, N + j);
                add_edge(j, N + i);
            }
            if (min(S[i] + D[i], T[j]) > max(S[i], T[j] - D[j])) {
                // x_i⇒x_j、¬x_j⇒¬x_i
                add_edge(i, j);
                add_edge(N + j, N + i);
            }
            if (min(T[i], S[j] + D[j]) > max(T[i] - D[i], S[j])) {
                // ¬x_i⇒¬x_j、x_j⇒x_i
                add_edge(N + i, N + j);
                add_edge(j, i);
            }
            if (min(T[i], T[j]) > max(T[i] - D[i], T[j] - D[j])) {
                // ¬x_i⇒x_j、¬x_j⇒x_i
                add_edge(N + i, j);
                add_edge(N + j, i);
            }
        }
    }
    scc();

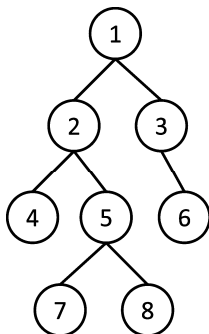
    // 判断是否可满足
    for (int i = 0; i < N; i++) {
        if (cmp[i] == cmp[N + i]) {
            printf("NO\n");
            return;
        }
    }

    // 如果可满足, 则给出一组解
    printf("YES\n");
    for (int i = 0; i < N; i++) {
        if (cmp[i] > cmp[N + i]) {
            // x_i为真, 即在结婚仪式开始时举行
            printf("%02d:%02d %02d:%02d\n", S[i] / 60, S[i] % 60, (S[i] + D[i]) / 60,
                (S[i] + D[i]) % 60);
        } else {
            // x_i为假, 即在结婚仪式结束时举行
            printf("%02d:%02d %02d:%02d\n", (T[i] - D[i]) / 60, (T[i] - D[i]) % 60,
                T[i] / 60, T[i] % 60);
        }
    }
}
}

```

4.3.3 LCA

在有根树中，两个节点 u 和 v 的公共祖先中距离最近的那个被称为最近公共祖先（LCA，Lowest Common Ancestor）。用于高效计算LCA的算法有许多，在此我们介绍其中的两种。在下文中，我们都假设节点数为 n 。



LCA的例子（4和7的LCA为2，8和6的LCA为1，5和8的LCA为5）

1. 基于二分搜索的算法

记节点 v 到根的深度为 $\text{depth}(v)$ 。那么，如果节点 w 是 u 和 v 的公共祖先的话，让 u 向上走 $\text{depth}(u)-\text{depth}(w)$ 步，让 v 向上走 $\text{depth}(v)-\text{depth}(w)$ 步，就都将走到 w 。因此，首先让 u 和 v 中较深的一方向上走 $|\text{depth}(u)-\text{depth}(v)|$ 步，再一起一步步向上走，直到走到同一个节点，就可以在 $O(\text{depth}(u)+\text{depth}(v))$ 时间内求出LCA。

```

// 输入
vector<int> G[MAX_V]; // 图的邻接表表示
int root; // 根节点的编号

int parent[MAX_V]; // 父亲节点（根节点的父亲记为-1）
int depth[MAX_V]; // 节点的深度

void dfs(int v, int p, int d) {
    parent[v] = p;
    depth[v] = d;
    for (int i = 0; i < G[v].size(); i++) {
        if (G[v][i] != p) dfs(G[v][i], v, d + 1);
    }
}

// 预处理
void init() {
    // 预处理出parent和depth
    dfs(root, -1, 0);
}

// 计算u和v的LCA

```

```

int lca(int u, int v) {
    // 让u和v向上走到同一深度
    while (depth[u] > depth[v]) u = parent[u];
    while (depth[v] > depth[u]) v = parent[v];
    // 让u和v向上走到同一节点
    while (u != v) {
        u = parent[u];
        v = parent[v];
    }
    return u;
}

```

节点的最大深度是 $O(n)$ ，所以该算法的复杂度也是 $O(n)$ 。如果只需计算一次LCA的话，这便足够了。但如果要计算多对节点的LCA的话如何是好呢？刚才的算法，通过不断向上走到同一节点来计算 u 和 v 的LCA。这里，到达了同一节点后，不论再怎么向上走，到达的显然还是同一节点。利用这一点，我们能够利用二分搜索求出到达共同祖先所需的最少步数吗？事实上，只要利用如下预处理，就可以实现二分搜索。

首先，对于任意顶点 v ，利用其父亲节点信息，可以通过 $\text{parent2}[v]=\text{parent}[\text{parent}[v]]$ 得到其向上走两步所到的顶点。再利用这一信息，又可以通过 $\text{parent4}[v]=\text{parent2}[\text{parent2}[v]]$ 得到其向上走四步所到的顶点。依此类推，就能够得到其向上走 2^k 步所到的顶点 $\text{parent}[k][v]$ 。有了 $k=\text{floor}(\log n)$ 以内的所有信息后，就可以二分搜索了，每次的复杂度是 $O(\log n)$ 。另外，预处理 $\text{parent}[k][v]$ 的复杂度是 $O(n \log n)$ 。

```

// 输入
vector<int> G[MAX_V]; // 图的邻接表表示
int root; // 根节点的编号

int parent[MAX_LOG_V][MAX_V]; // 向上走 $2^k$ 步所到的节点（超过根时记为-1）
int depth[MAX_V]; // 节点的深度

void dfs(int v, int p, int d) {
    parent[0][v] = p;
    depth[v] = d;
    for (int i = 0; i < G[v].size(); i++) {
        if (G[v][i] != p) dfs(G[v][i], v, d + 1);
    }
}

// 预处理
void init(int V) {
    // 预处理出parent[0]和depth
    dfs(root, -1, 0);
    // 预处理出parent
    for (int k = 0; k + 1 < MAX_LOG_V; k++) {
        for (int v = 0; v < V; v++) {
            if (parent[k][v] < 0) parent[k + 1][v] = -1;
            else parent[k + 1][v] = parent[k][parent[k][v]];
        }
    }
}

```

```
    }  
}  
  
// 计算u和v的LCA  
int lca(int u, int v) {  
    // 让u和v向上走到同一深度  
    if (depth[u] > depth[v]) swap(u, v);  
    for (int k = 0; k < MAX_LOG_V; k++) {  
        if ((depth[v] - depth[u]) >> k & 1) {  
            v = parent[k][v];  
        }  
    }  
    if (u == v) return u;  
    // 利用二分搜索计算LCA  
    for (int k = MAX_LOG_V - 1; k >= 0; k--) {  
        if (parent[k][u] != parent[k][v]) {  
            u = parent[k][u];  
            v = parent[k][v];  
        }  
    }  
    return parent[0][u];  
}
```

像这样，预处理出 2^k 的表的技巧，在计算LCA之外也很有用，相关的问题也经常出现在程序设计竞赛当中。对此，下一节中还会介绍其他例子。

2. 基于RMQ的算法

对于涉及有根树的问题，将树转为从根DFS标号后得到的序列处理的技巧常常十分有效。对于LCA，利用该技巧也能够高效地计算。首先，按从根DFS访问的顺序得到顶点序列 $vs[i]$ 和对应的深度 $depth[i]$ 。对于每个顶点 v ，记其在 vs 中首次出现的下标为 $id[v]$ 。

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vs	1	2	4	2	5	7	5	8	5	2	1	3	6	3	1
depth	0	1	2	1	2	3	2	3	2	1	0	1	2	1	0

i	1	2	3	4	5	6	7	8
id	0	1	11	2	4	12	5	7

样例对应的标号

这些都可以在 $O(n)$ 时间内求得。而 $LCA(u,v)$ 就是访问 u 之后到访问 v 之前所经过顶点中离根最近的那个，假设 $id[u] \leq id[v]$ ，那么有

$$LCA(u,v) = vs[id[u] \leq i \leq id[v] \text{ 中令 } depth(i) \text{ 最小的 } i]$$

而这可以利用RMQ高效地求得。

```

// 输入
vector<int> G[MAX_V];      // 图的邻接表表示
int root;

int vs[MAX_V * 2 - 1];     // DFS访问的顺序
int depth[MAX_V * 2 - 1];  // 节点的深度
int id[MAX_V];             // 各个顶点在vs中首次出现的下标

void dfs(int v, int p, int d, int &k) {
    id[v] = k;
    vs[k] = v;
    depth[k++] = d;
    for (int i = 0; i < G[v].size(); i++) {
        if (G[v][i] != p) {
            dfs(G[v][i], v, d + 1, k);
            vs[k] = v;
            depth[k++] = d;
        }
    }
}

// 预处理
void init(int V) {
    // 预处理出vs、depth和id
    int k = 0;
    dfs(root, -1, 0, k);
    // 预处理出RMQ (返回的不是最小值, 而是最小值对应的下标)
    rmq_init(depth, V * 2 - 1);
}

// 计算u和v的LCA
int lca(int u, int v) {
    return vs[query(min(id[u], id[v]), max(id[u], id[v]) + 1)];
}

```

Housewife Wind (POJ No.2763)

$\times \times$ 村里有 n 个小屋, 小屋之间有双向可达的道路相连, 所构成的图是一棵树。通过连接 a_i 号小屋和 b_i 号小屋的道路 i 需要花费 w_i 的时间。你一开始在 s 号小屋。请处理以下 q 个查询。

A: 输出从当前位置移动到节点 x 所需的时间。 B: 将通过道路 x 所需的时间改为 t 。

限制条件

- $1 \leq n \leq 100000$
- $0 \leq q \leq 100000$
- $1 \leq a_i, b_i \leq n$
- $1 \leq w_i \leq 10000$

样例

输入
<pre>n = 3 q = 3 s = 1 (a, b, w) = {(1, 2, 1), (2, 3, 2)} (查询的类型, x(l, t)) = {(A, 2), (B, 2, 3), (A, 3)}</pre>
输出
<pre>从1移动到2 从2移动到3</pre>

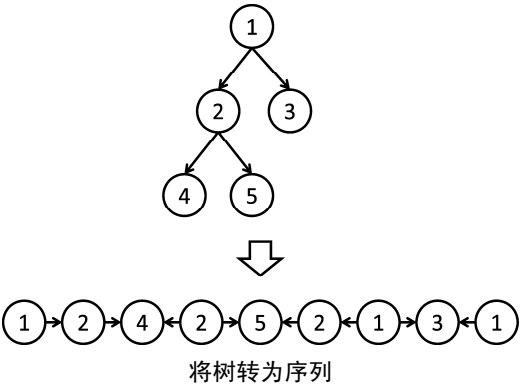
虽然直接DFS也可以求出树上两点之间的距离，但是这对于每个A类型的查询，都要花费 $O(n)$ 的时间，实在太慢了。必须利用树的特性，得到更为高效的算法。为了高效地处理A类型的查询，可以利用二分搜索版LCA算法中介绍的技巧，记录下从每个顶点向上走 2^k 步的总长度。这样一来，在 $O(\log n)$ 地计算LCA的同时，也可以同样 $O(\log n)$ 地求出到LCA的距离，因此处理A类型查询的复杂度为 $O(\log n)$ 。但是，这个方法对于B类型的查询却无法高效地处理。

因此，让我们先考虑一下图是链状时这一简单的情况。假设 i 和 $i+1$ 两点之间的边的长度为 w_i ，则两点 u 和 $v(u < v)$ 之间的距离为

$$\sum_{i=u}^{v-1} w_i$$

只要用BIT，不论是A类型的查询还是B类型的查询，都能够在 $O(\log n)$ 时间内处理。

在回过头来考虑树的情况。因为树中连接两点的路径是唯一的，如果我们对顶点进行合理排列的话，能否像链状时那样，进行类似的处理呢？考虑利用RMQ计算LCA时所用的，按DFS访问的顺序排列的顶点序列。这样， u 和 v 之间的路径，就是在序列中 u 和 v 之间的所有边减去往返重复的部分得到的结果。



于是，只要令边的权重沿叶子方向为正，沿根方向为负，那么往返重复的部分就自然抵消了，于是有

(u, v 之间的距离)=(从 $LCA(u, v)$ 到 u 的边的权重和)+(从 $LCA(u, v)$ 到 v 的边的权重和)

同链状的情况一样，利用BIT的话，计算权重和和更新边权都可以在 $O(\log n)$ 时间内办到，而LCA也能在 $O(\log n)$ 时间内求得。

```

struct edge { int id, to, cost; };

int n, q, s;
int a[MAX_V - 1], b[MAX_V - 1], w[MAX_V - 1];
int type[MAX_Q]; // 0: A类型, 1: B类型
int x[MAX_Q], t[MAX_Q];

vector<edge> G[MAX_V]; // 图的邻接表表示
int root;

int vs[MAX_V * 2 - 1]; // DFS访问的顺序
int depth[MAX_V * 2 - 1]; // 节点的深度
int id[MAX_V]; // 各个顶点在vs中首次出现的下标
int es[(MAX_V - 1) * 2]; // 边的下标 (i*2+(叶子方向:0,根方向:1))

void dfs(int v, int p, int d, int &k) {
    id[v] = k;
    vs[k] = v;
    depth[k++] = d;
    for (int i = 0; i < G[v].size(); i++) {
        edge &e = G[v][i];
        if (e.to != p) {
            add(k, e.cost);
            es[e.id * 2] = k;
            dfs(e.to, v, d + 1, k);
            vs[k] = v;
            depth[k++] = d;
            add(k, -e.cost);
            es[e.id * 2 + 1] = k;
        }
    }
}

int stack_v[MAX_V + 10];
int stack_i[MAX_V + 10];

// 预处理
void init(int V) {
    // 初始化BIT
    bit_n = (V - 1) * 2;
    // 预处理出vs、depth、id和es
    int k = 0;
    dfs(root, -1, 0, k);
    // 预处理出RMQ (返回的不是最小值, 而是最小值对应的下标)
}

```

```

    rmq_init(depth, V * 2 - 1);
}

// 计算u和v的LCA
int lca(int u, int v) {
    return vs[query(min(id[u], id[v]), max(id[u], id[v]) + 1)];
}

void solve() {
    // 预处理
    root = n / 2; // 不论以哪个节点为根都没有问题
    for (int i = 0; i < n - 1; i++) {
        G[a[i] - 1].push_back((edge){i, b[i] - 1, w[i]});
        G[b[i] - 1].push_back((edge){i, a[i] - 1, w[i]});
    }
    init(n);
    // 处理查询
    int v = s - 1; // 当前位置
    for (int i = 0; i < q; i++) {
        if (type[i] == 0) {
            // 从当前位置移动到x[i]
            int u = x[i] - 1;
            int p = lca(v, u);
            // 利用BIT计算p到v和p到u的费用之和, 即区间(id[p], id[v]]和(id[p], id[u]]的权重和
            printf("%d\n", sum(id[v]) + sum(id[u]) - sum(id[p]) * 2);
            v = u;
        } else {
            // 将通过道路x[i]的权重改为t[i]。
            int k = x[i] - 1;
            add(es[k * 2], t[i] - w[k]);
            add(es[k * 2 + 1], w[k] - t[i]);
            w[k] = t[i];
        }
    }
}
}

```
