图灵程序设计丛书 TURING

挑战程序设计竞赛（第2版）

世界顶级程序设计高手的经验总结

ACM-ICPC全球总冠军 巫泽俊 主译

【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
POSTS & TELECOM PRESS

日本 ACM-ICPC参赛者 人手一册

世界顶级程序设计高手联手打造

★秋叶拓哉 昵称iwi

Google Code Jam 2010 第9名
ACM-ICPC World Finals 2012 第11名
TopCoder Open 2012 Algorithm 第4名

★巫泽俊 昵称watashi和rejudge

ACM-ICPC World Finals 2009 第6名
ACM-ICPC World Finals 2011 冠军
Google Code Jam 2012 第7名

★岩田阳一 昵称wata

Google Code Jam 2009 第3名
TopCoder Open 2010 Marathon 冠军
IPSC 2010 个人组 冠军

★庄俊元 昵称navi和navimoe

ACM-ICPC Asia Phuket Regional 2011 冠军
2012年跻身ACM-ICPC World Finals
以及百度Astar总决赛

★北川宜稔 昵称kita_masa

ACM-ICPC World Finals 2010 第16名

★李津羽

浙江大学2011级计算机系博士生
在浙大CAD&CG实验室从事科研工作

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-32010-0

定价：79.00元

最负盛名的程序设计竞赛

★Google Code Jam（GCJ）

Google公司举办的一年一度的程序设计竞赛。

- 影响力最大、参赛面最广的程序设计竞赛。
- 每道题都有Small和Large等不同规模的测试数据。
- 自由选择喜欢的工具在本地运行程序，并提交结果。

★TopCoder

TopCoder公司举办的程序设计竞赛。既有周期举办的SRM，又有一年一度的TCO。

- 在75分钟内挑战难度和分数递增的三道题。
- 每道题的得分随用时递减。
- 独有的挑战阶段，通过寻找他人程序的漏洞赚取额外的分数。
- 系统评测的结果在最后才公布。

★ACM-ICPC

美国计算机协会（ACM）主办的面向大学生的竞赛。

- 历史最悠久、最负盛名的程序设计竞赛。
- 三名选手共用一台电脑的团体比赛。
- 5个小时内挑战10道左右的问题。

图灵程序设计丛书 TURING

挑战程序设计竞赛（第2版）

【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（CIP）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，2013.7
（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
审　陈　越　翁　恺　王　灿
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
北京中新伟业印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：26.5
字数：626千字　2013年 7 月第 1 版
印数：1－4 000册　2013年 7 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和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。

目　　录

第 1 章　蓄势待发——准备篇 ……1

1.1　何谓程序设计竞赛 ……2

1.2　最负盛名的程序设计竞赛 ……5

1.2.1　世界规模的大赛——Google Code Jam（GCJ） ……5

1.2.2　向高排名看齐！——TopCoder ……5

1.2.3　历史最悠久的竞赛——ACM-ICPC ……6

1.2.4　面向中学生的信息学奥林匹克竞赛——JOI-IOI ……6

1.2.5　通过网络自动评测——Online Judge（OJ） ……6

1.3　本书的使用方法 ……7

1.3.1　本书所涉及的内容 ……7

1.3.2　所用的编程语言 ……7

1.3.3　题目描述的处理 ……7

1.3.4　程序结构 ……7

1.3.5　练习题 ……8

1.3.6　读透本书后更上一层楼的练习方法 ……8

1.4　如何提交解答 ……9

1.4.1　POJ 的提交方法 ……9

1.4.2　GCJ 的提交方法 ……11

1.5　以高效的算法为目标 ……15

1.5.1　什么是复杂度 ……15

1.5.2　关于运行时间 ……15

1.6　轻松热身 ……16

1.6.1　先从简单题开始 ……16

1.6.2　POJ 的题目 Ants ……18

1.6.3　难度增加的抽签问题 ……20

第 2 章　初出茅庐——初级篇 ……25

2.1　最基础的“穷竭搜索” ……26

2.1.1　递归函数 ……26

2.1.2　栈 ……27

2.1.3　队列 ……28

2.1.4　深度优先搜索 ……29

2.1.5　宽度优先搜索 ……33

2.1.6　特殊状态的枚举 ……37

2.1.7　剪枝 ……38

2.2　一往直前！贪心法 ……39

2.2.1　硬币问题 ……39

2.2.2　区间问题 ……40

2.2.3　字典序最小问题 ……43

2.2.4　其他例题 ……45

2.3　记录结果再利用的“动态规划” ……51

2.3.1　记忆化搜索与动态规划 ……51

2.3.2　进一步探讨递推关系 ……57

2.3.3　有关计数问题的 DP ……66

2.4　加工并存储数据的数据结构 ……70

2.4.1　树和二叉树 ……70

2.4.2　优先队列和堆 ……71

2.4.3　二叉搜索树 ……77

2.4.4　并查集 ……84

2.5　它们其实都是“图” ……91

2.5.1 图是什么 ····· 91
2.5.2 图的表示 ····· 94
2.5.3 图的搜索 ····· 97
2.5.4 最短路问题 ····· 99
2.5.5 最小生成树 ····· 105
2.5.6 应用问题 ····· 107
2.6 数学问题的解题窍门 ····· 113
2.6.1 辗转相除法 ····· 113
2.6.2 有关素数的基础算法 ····· 117
2.6.3 模运算 ····· 121
2.6.4 快速幂运算 ····· 122
2.7 一起来挑战 GCJ 的题目（1） ····· 125
2.7.1 Minimum Scalar Product ····· 125
2.7.2 Crazy Rows ····· 127
2.7.3 Bribe the Prisoners ····· 129
2.7.4 Millionaire ····· 132

第 3 章 出类拔萃——中级篇 ····· 137

3.1 不光是查找值！“二分搜索” ····· 138
3.1.1 从有序数组中查找某个值 ····· 138
3.1.2 假定一个解并判断是否可行 ····· 140
3.1.3 最大化最小值 ····· 142
3.1.4 最大化平均值 ····· 143
3.2 常用技巧精选（一） ····· 146
3.2.1 尺取法 ····· 146
3.2.2 反转（开关问题） ····· 150
3.2.3 弹性碰撞 ····· 158
3.2.4 折半枚举（双向搜索） ····· 160
3.2.5 坐标离散化 ····· 164
3.3 活用各种数据结构 ····· 167
3.3.1 线段树 ····· 167
3.3.2 Binary Indexed Tree ····· 174
3.3.3 分桶法和平方分割 ····· 183
3.4 熟练掌握动态规划 ····· 191
3.4.1 状态压缩 DP ····· 191
3.4.2 矩阵的幂 ····· 199
3.4.3 利用数据结构高效求解 ····· 206
3.5 借助水流解决问题的网络流 ····· 209
3.5.1 最大流 ····· 209
3.5.2 最小割 ····· 212
3.5.3 二分图匹配 ····· 217
3.5.4 一般图匹配 ····· 220
3.5.5 匹配、边覆盖、独立集和顶点覆盖 ····· 221
3.5.6 最小费用流 ····· 222
3.5.7 应用问题 ····· 228
3.6 与平面和空间打交道的计算几何 ····· 250
3.6.1 计算几何基础 ····· 250
3.6.2 极限情况 ····· 255
3.6.3 平面扫描 ····· 258
3.6.4 凸包 ····· 260
3.6.5 数值积分 ····· 263
3.7 一起来挑战 GCJ 的题目（2） ····· 267
3.7.1 Numbers ····· 267
3.7.2 No Cheating ····· 269
3.7.3 Stock Charts ····· 271
3.7.4 Watering Plants ····· 273
3.7.5 Number Sets ····· 278
3.7.6 Wi-fi Towers ····· 280

第 4 章 登峰造极——高级篇 ····· 285

4.1 更加复杂的数学问题 ····· 286
4.1.1 矩阵 ····· 286
4.1.2 模运算的世界 ····· 291
4.1.3 计数 ····· 295
4.1.4 具有对称性的计数 ····· 300
4.2 找出游戏的必胜策略 ····· 305
4.2.1 游戏与必胜策略 ····· 305
4.2.2 Nim ····· 311
4.2.3 Grundy 数 ····· 315
4.3 成为图论大师之路 ····· 320
4.3.1 强连通分量分解 ····· 320

4.3.2　2-SAT ……324
4.3.3　LCA ……328
4.4　常用技巧精选（二）……335
4.4.1　栈的运用 ……335
4.4.2　双端队列的运用 ……337
4.4.3　倍增法 ……345
4.5　开动脑筋智慧搜索 ……350
4.5.1　剪枝 ……350
4.5.2　A*与 IDA* ……356
4.6　划分、解决、合并：分治法 ……359
4.6.1　数列上的分治法 ……359
4.6.2　树上的分治法 ……360
4.6.3　平面上的分治法 ……364
4.7　华丽地处理字符串 ……368
4.7.1　字符串上的动态规划算法 ……368
4.7.2　字符串匹配 ……373
4.7.3　后缀数组 ……378
4.8　一起来挑战 GCJ 的题目（3）……387
4.8.1　Mine Layer ……387
4.8.2　Year of More Code Jam ……392
4.8.3　Football Team ……395
4.8.4　Endless Knight ……399
4.8.5　The Year of Code Jam ……403
本书中未涉及的拓展主题 ……408
书中例题列表 ……411
参考文献 ……413

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 $k_1, k_2, \cdots, k_n$ 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

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

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 <cstdio>

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

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$1 \leqslant n \leqslant 50$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$1 \leqslant n \leqslant 1000$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$1 \leqslant n \leqslant 1000$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（challege 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，所以团队配合显得异常重要。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程序设计竞赛。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。
- Codecorces——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，理所当然是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

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。

1.4 如何提交解答

➠这里以POJ和GCJ为例，介绍提交解答的方法。

1.4.1 POJ的提交方法

接下来，我们试着在POJ里提交写好的程序。POJ的地址是http://poj.org/。用户需要在POJ注册后才能提交。注册页面除了用户名和密码外，还有email地址等信息的输入框，这些不是非填不可的。

注册页面

成功登录之后，让我们试提交一下测试题A+B Problem。

题目描述页面

A+B Problem从标准输入读入两个整数a和b，并将它们的和$a+b$输出到标准输出。A+B Problem里还提供了提交示例。让我们试提交下面这个程序。

```
#include <cstdio>

int main() {
  int a, b;
  scanf("%d %d", &a, &b);
  printf("%d\n", a + b);
  return 0;
}
```

可以通过题目描述最下方的Submit链接提交。结果如下。

Accepted

顺利获得Accepted，表示所提交解答是正确的。如果程序输入了错误的答案，则会变成Wrong Answer。试将`printf("%d\n", a + b)`替换成`printf("%d\n", a * b)`再提交，果然就返回了Wrong Answer。

Wrong Answer

和比赛一样，Online Judge对程序的运行时间是有限制的。对于A+B Problem，这个限制是1000ms。让我们试提交一个超时的程序。在*printf*的后面加上一行*for* (;;); 后就返回了Time Limit Exceeded。

Time Limit Exceeded

系统的使用方法大致就是这样。此外，提交后的返回结果还有以下其他几种。

Runtime Error	表示程序因为非法内存访问或未处理异常而结束。
Memory Limit Exceeded	表示程序使用的内存超过规定的内存限制。
Presentation Error	表示虽然程序输出的答案是对的，但是换行或空格等不符合输出格式要求。
Output Limit Exceeded	表示程序输出了过多的内容。
Compile Error	表示所提交的源代码没能通过编译。这时打开Online Status的“Compile Error”链接还可以看到具体的编译错误信息。
System Error, Validator Error	表示系统发生错误无法正常判题。

1.4.2 GCJ的提交方法

现在来说明GCJ的提交方法。GCJ的地址是http://code.google.com/codejam/。正式的比赛也是通过这个网站通知和进行的。比赛的规则和日程都在这里，请认真查阅。

首页

按下Practice按钮，就会跳转到一个包含历史比赛列表的页面。

历届比赛列表等信息

虽然GCJ无需注册也能练习，不过正式参赛还是需要注册的，所以不妨提前注册好。在Upcoming Contest区域可以看到最近比赛的安排。如果比赛开始，这里则会显示参赛链接。

本书所介绍的GCJ的题目，都标注有场次及题号，便于读者练习。

从Previous Contests的列表中点击比赛名称，就会跳转到练习页面。虽然称为练习页面，和正式比赛的页面是一样的。在左下角的Top Scores中可以看到正式比赛中选手的得分，点击Full scoreboard还能看到更详细的结果。

在Full scoreboard中不光有详细的得分，还可以下载到正式比赛中选手所提交的源代码。页面左侧中央位置还有一个Submissions框，显示有正式比赛中各个问题的提交情况和分数。

Round 1A 2008

页面右侧从上到下依次是提交表单、题目描述、输入格式、输出格式、限制条件和样例。限制条件除了有Small和Large之分外，与POJ的基本无异。

点击名为Download A-small.in的链接，就会开始下载输入文件。正式比赛中，一下载输入文件就开始倒计时，如果超过了时间限制，就会得到Time expired，相当于Incorrect。下载好输入文件后，就可以将其交给写好的程序处理并将答案保存到输出文件。点击Submit按钮就能看到提交表单。在your output file中选择包含答案的输出文件，在source file(s)中选择源代码[①]，然后点击Sumbit file按钮提交。练习时，结果会马上返回。

正式比赛中，提交Small后马上就会返回Correct或是Incorrect。如果是Incorrect，则可以再反复不断尝试[②]。Large则要等到比赛结束后才知道是否正确。在提交时间限制内可以多次提交，但一旦过了这个时间，就不能再提交了。所以比赛中意外点击下载了Large的输入数据就糟糕了，比赛中需要多加注意[③]。

答案错误的情况（Incorrect）

按下Submit按钮后的页面

答案正确的情况（Correct）

① 只有正式比赛中才需要提交源代码。

② 每次Incorrect的尝试都会追加罚时。

③ 为此，GCJ的系统被设计成用户通过Small后才能下载Large的输入数据。——译者注

比赛最终是按照总分从大到小排名的。比赛时没有必要在解决了Small之后勉强去尝试对应的Large。解不出Large时，优先尝试已经有较多人解决的题目或估计能很快解决的Small是更为有效的策略。

在不同年份不同轮次，GCJ的规则都可能有变化，详细的规则请参见GCJ的主页。

1.5 以高效的算法为目标

➠本节介绍算法设计中至关重要的复杂度的估算方法。

1.5.1 什么是复杂度

在设计满足问题要求的算法时，复杂度的估算是非常重要的。我们不可能把每个想到的算法都去实现一遍看看是否足够快。应当通过估算算法的复杂度来判断所想的算法是否足够高效。

在分析复杂度时，我们通常考虑它与什么成正比，并称之为算法的阶。例如1.1节的程序执行了四重循环，每重n次，运行时间与n^4成正比。我们将与n^4成正比写作$O(n^4)$，将对应的运行时间写作$O(n^4)$时间[①]。

1.5.2 关于运行时间

程序的运行时间不光取决于复杂度，也会受诸如循环体的复杂性等因素的影响。但是，因此造成的差距多数情况下最多也就几十倍。另一方面，忽略其余因素，n=1000时，$O(n^3)$时间的算法和$O(n^2)$时间的算法的差距就是1000倍。因此要缩短程序的运行时间，主要应该从复杂度入手。

估算出算法的复杂度后，只要将数值可能的最大值代入复杂度的渐近式中，就能简单地判断算法是否能够满足运行时间限制的要求。例如，考虑$O(n^2)$时间的算法，假设题目描述中的限制条件为$n \leqslant 1000$，将n=1000代入n^2就得到了1000000。在这个数值的基础上，我们就可以结合下表进行判断了。

假设时间限制为1秒

1000000	游刃有余
10000000	勉勉强强
100000000	很悬，仅限循环体非常简单的情况

① 这里介绍的大O记号，严格来说并不是通过正比关系定义的，不过刚开始时这样理解也无妨。另外，我们也常常省略“时间”二字，用$O(n^4)$来表示与n^4成正比的运行时间。

1.6 轻松热身

➠本节通过对几个问题的解析，和大家一起了解程序设计竞赛中题目的风格，学习设计算法并估算复杂度的过程。其中有一些问题比较复杂，不能想到它们的解法也不要紧，能够通过阅读题解体会到其中的趣味就好。

1.6.1 先从简单题开始

三角形

有 n 根棍子，棍子 i 的长度为 a_i。想要从中选出 3 根棍子组成周长尽可能长的三角形。请输出最大的周长，若无法组成三角形则输出 0。

用5根棍子组成三角形的例子

⚠限制条件

- $3 \leqslant n \leqslant 100$
- $1 \leqslant a_i \leqslant 10^6$

样例 1

输入

```
n = 5
a = {2, 3, 4, 5, 10}
```

输出

```
12（选择3、4、5时）
```

样例 2

输入

```
n = 4
a = {4, 5, 10, 20}
```

输出

```
0（无论怎么选都无法组成三角形）
```

选择3根棍子，它们能组成三角形的充要条件为

最长棍子的长度 < 其余两根棍子的长度之和。

能够组成三角形的条件

于是我们可以试想这样一种算法：首先用三重循环枚举所有的棍子选择方案，再利用上式判断能否组成三角形。如果可以，那么该三角形的周长就是备选答案。

这里用了三种循环，所以复杂度是$O(n^3)$。将n=100代入n^3得到10^6，可知这个复杂度是足够低的[①]。

```
// 输入
int n, a[MAX_N];

void solve() {
  int ans = 0; // 答案

  // 让i < j < k，这样棍子就不会被重复选中了
  for (int i = 0; i < n; i++) {
```

① 本题还有$O(n\log n)$时间更高效的算法，留给有兴趣的读者思考。

```
      for (int j = i + 1; j < n; j++) {
        for (int k = j + 1; k < n; k++) {
          int len = a[i] + a[j] + a[k];         // 周长
          int ma = max(a[i], max(a[j], a[k])); // 最长棍子的长度
          int rest = len - ma;                  // 其余两根棍子的长度之和

          if (ma < rest) {
            // 可以组成三角形，如果可以更新答案则更新
            ans = max(ans, len);
          }
        }
      }
    }

    // 输出
    printf("%d\n", ans);
  }
```

1.6.2　POJ的题目Ants

Ants（POJ No.1852）

n 只蚂蚁以每秒 1cm 的速度在长为 Lcm 的竿子上爬行。当蚂蚁爬到竿子的端点时就会掉落。由于竿子太细，两只蚂蚁相遇时，它们不能交错通过，只能各自反向爬回去。对于每只蚂蚁，我们知道它距离竿子左端的距离 x_i，但不知道它当前的朝向。请计算所有蚂蚁落下竿子所需的最短时间和最长时间。

各个蚂蚁正朝向哪边是不知道的

竿子和蚂蚁的情况

限制条件

- $1 \leqslant L \leqslant 10^6$
- $1 \leqslant n \leqslant 10^6$
- $0 \leqslant x_i \leqslant L$

样例

输入

```
L = 10
n = 3
x = {2, 6, 7}
```

输出

```
min = 4（左、右、右）
max = 8（右、右、右）
```

首先很容易想到一个穷竭搜索[①]算法，即枚举所有蚂蚁的初始朝向的组合，这可以利用递归函数实现（详见2.1节）。

每只蚂蚁的初始朝向都有2种可能，n只蚂蚁就是$2 \times 2 \times \cdots \times 2 = 2^n$种。如果$n$比较小，这个算法还是可行的，但指数函数随着$n$的增长会急剧增长。

2^n增长的趋势

n	1	5	10	20	30	100	10000	1000000
2^n	2	32	1024	1048576	10^9	10^{30}	10^{3010}	10^{301030}

穷竭搜索的运行时间也随之急剧增长。一般把指数阶的运行时间叫做指数时间。指数时间的算法无法处理稍大规模的输入。

接下来，让我们来考虑比穷竭搜索更高效的算法。首先对于最短时间，看起来所有蚂蚁都朝向较近的端点走会比较好。事实上，这种情况下不会发生两只蚂蚁相遇的情况，而且也不可能在比此更短的时间内走到竿子的端点。

接下来，为了思考最长时间的情况，让我们看看蚂蚁相遇时会发生什么。

相遇后会发生什么

事实上，可以知道两只蚂蚁相遇后，当它们保持原样交错而过继续前进也不会有任何问题。这样

① 也叫蛮力搜索，口语中常简称暴搜。——译者注

看来，可以认为每只蚂蚁都是独立运动的，所以要求最长时间，只要求蚂蚁到竿子端点的最大距离就好了。

这样，不论最长时间还是最短时间，都只要对每只蚂蚁检查一次就好了，这是$O(n)$时间的算法。对于限制条件$n \leqslant 10^6$，这个算法是够用的，于是问题得解。

```
// 输入
int L, n;
int x[MAX_N];

void solve() {
  // 计算最短时间
  int minT = 0;
  for (int i = 0; i < n; i++) {
    minT = max(minT, min(x[i], L - x[i]));
  }

  // 计算最长时间
  int maxT = 0;
  for (int i = 0; i < n; i++) {
    maxT = max(maxT, max(x[i], L - x[i]));
  }

  printf("%d %d\n", minT, maxT);
}
```

这个问题可以说是考察想象力类型问题的经典例子。有很多这样的问题，虽然开始不太明白，但想通之后，最后的程序却是出乎意料地简单。

1.6.3　难度增加的抽签问题

如果将最开始的抽签问题中关于n的限制条件改为$1 \leqslant n \leqslant 1000$（题目描述参见1.1节），那么应该如何求解呢？最初的四重循环算法是$O(n^4)$时间的，将n=1000带入n^4得到10^{12}，这是远远不够的，必须改进算法。

```
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;
        }
      }
    }
  }
}
```

上面是最初所记载的程序的循环部分。最内侧关于d的循环所做的事就是

$$\text{检查是否有}d\text{使得}\ k_a + k_b + k_c + k_d = m$$

通过对式子移向，就能得到另一种表达方式

$$\text{检查是否有}d\text{使得}\ k_d = m - k_a - k_b - k_c$$

就是说，检查数组k中所有元素，判断是否有$m-k_a-k_b-k_c$。

让我们着眼这一点来考虑快速的检查方法。虽然也有利用数据结构优化的方法（在2.4节介绍），这里要介绍的是名为二分搜索的算法。

1. 二分搜索与$O(n^3\log n)$的算法

记所要查找的值$m-k_a-k_b-k_c$为x。预先把数组k排好序，然后看k中央的数字[①]，可知

- 如果它比x小，x只可能在它的后面半段。
- 如果它比x大，x只可能在它的前面半段。

如果再将上述方法运用在已经减半的x的存在区间上，x的存在区间就变成了初始的1/4。这样反复操作就可以不断缩小x的存在区间，最终可以确定x存在与否。

从数列中查找55的例子

二分搜索算法每次将候选区间减小至大约原来的一半。因此，要判断长度为n的有序数组k中是否包含x，只要反复执行约$\log_2 n$次就完成了。二分查找的复杂度是$O(\log n)$时间的，我们称这种阶的运行时间为对数时间。即便n变得很大时，对数时间的算法依然非常快速。

① 如果元素个数是偶数，是没有中间数字的，这时候观察离中间最近的某边的值。

即使n变得很大，$\log_2 n$也依然很小

n	1	10	100	1000	10^6	10^9
$\log_2 n$	0	3	7	10	20	30

将最内侧的循环替换成二分搜索算法之后，就变成

- 排序$O(n\log n)$时间
- 循环$O(n^3\log n)$时间

$n^3\log n$比$n\log n$大，所以这里合起来当作$O(n^3\log n)$时间。于是，我们得到了在$O(n^3\log n)$时间内解决抽签问题的算法。

```
// 输入
int n, m, k[MAX_N];

bool binary_search(int x) {
  // x的存在范围是k[l], k[l+1], …, k[r-1].
  int l = 0, r = n;

  // 反复操作直到存在范围为空
  while (r - l >= 1) {
    int i = (l + r) / 2;
    if (k[i] == x) return true; // 找到x
    else if (k[i] < x) l = i + 1;
    else r = i;
  }

  // 没找到x
  return false;
}

void solve() {
  // 为了执行二分查找需要先排序
  sort(k, k + n);

  bool f = false;

  for (int a = 0; a < n; a++) {
    for (int b = 0; b < n; b++) {
      for (int c = 0; c < n; c++) {
        // 将最内侧的循环替换成二分查找
        if (binary_search(m - k[a] - k[b] - k[c])) {
          f = true;
        }
      }
    }
  }

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

事实上，像binary_search这样的函数，多数情况下无需自己实现，可以使用标准实现。例如C++的STL中就含有提供基本同样功能的函数。

2. $O(n^2\log n)$的算法

但是，将n=1000带入$n^3\log n$，会发现这依然是远远不够的，必须要对算法做进一步优化。刚才我们只着眼于四重循环程序中最内层的循环。接下来，让我们着眼于内侧的两个循环。

同刚才一样的思路，内侧的两个循环是在

检查是否有c和d使得$k_c + k_d = m - k_a - k_b$。

这种情况并不能直接使用二分搜索。但是，如果预先枚举出k_c+k_d所得的n^2个数字并排好序，便可以利用二分搜索了[①]。

该算法

- 排序$O(n^2\log n)$时间
- 循环$O(n^2\log n)$时间

总的也是$O(n^2\log n)$时间。这样就可以确信即便n=1000也能妥善应对了。

```
// 输入
int n, m, k[MAX_N];

// 保存2个数的和的数列
int kk[MAX_N * MAX_N];

bool binary_search(int x) {
  // x的存在范围是kk[l], kk[l+1], …, kk[r-1].
  int l = 0, r = n * n;

  // 反复操作直到存在范围为空
  while (r - l >= 1) {
    int i = (l + r) / 2;
    if (kk[i] == x) return true; // 找到x
    else if (kk[i] < x) l = i + 1;
    else r = i;
  }

  // 没找到x
  return false;
}

void solve() {
  // 枚举k[c]+k[d]的和
  for (int c = 0; c < n; c++) {
```

① 确切地说，去除重复后$n(n+1)/2$个数字就够了，不过为了方便可以写成枚举n^2个数字。

```
      for (int d = 0; d < n; d++) {
        kk[c * n + d] = k[c] + k[d];
      }
    }

    // 排序以便进行二分搜索
    sort(kk, kk + n * n);

    bool f = false;
    for (int a = 0; a < n; a++) {
      for (int b = 0; b < n; b++) {
        // 将内侧的两个循环替换成二分搜索
        if (binary_search(m - k[a] - k[b])) {
          f = true;
        }
      }
    }

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

本问题既需要二分搜索这一基础算法知识，也需要将四个数分成两两考虑的想象力。此外，像这样从复杂度较高的算法出发，不断降低复杂度直到满足问题要求的过程，也是设计算法时常会经历的一个过程。

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，我们想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等程序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）。

栈的操作

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、…、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

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

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
```

```
    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

- $N, M \leqslant 100$

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(状态数×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

- $N, M \leqslant 100$

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$0 \leqslant a_i \leqslant 10^8$，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。

2.2 一往直前！贪心法

➠贪心法就是遵循某种规则，不断贪心地选取当前最优策略的算法设计方法。本节通过几个经典贪心问题的展示，来介绍贪心法。

2.2.1 硬币问题

硬币问题

有 1 元、5 元、10 元、50 元、100 元、500 元的硬币各 C_1、C_5、C_{10}、C_{50}、C_{100}、C_{500} 枚。现在要用这些硬币来支付 A 元，最少需要多少枚硬币？假定本题至少存在一种支付方案。

限制条件

- $0 \leqslant C_1, C_5, C_{10}, C_{50}, C_{100}, C_{500} \leqslant 10^9$
- $0 \leqslant A \leqslant 10^9$

样例

输入

```
C1 = 3, C5 = 2, C10 = 1, C50 = 3, C100 = 0, C500 = 2, A = 620
```

输出

```
6（500元硬币1枚，50元硬币2枚，10元硬币1枚，5元硬币2枚，合计6枚）
```

这是个贴近生活的简单问题。凭直觉，可以得出如下正确的解答。

- 首先尽可能多地使用500元硬币；
- 剩余部分尽可能多地使用100元硬币；
- 剩余部分尽可能多地使用50元硬币；
- 剩余部分尽可能多地使用10元硬币；
- 剩余部分尽可能多地使用5元硬币；
- 最后的剩余部分使用1元硬币支付。

或者，简而言之，

- 优先使用面值大的硬币。

该算法可以说是贪心算法中最简单的例子。上节的搜索算法和下节的动态规划算法是在多种策略中选取最优解，而贪心算法则不同，它遵循某种规则，不断地选取当前最优策略。例如在此题中，“优先使用面值大的硬币”就是在计算过程中所遵循的规则。并且，我们只考虑“尽可能多的使用面值大的硬币”这一种当前最优策略。

如果问题能够用贪心算法来求解的话，那么它通常是非常高效的。如果把这里的硬币问题当作一种背包问题（参考下节的第一个问题），那么比起用下节将要介绍的动态规划算法求解，贪心算法更简单高效。

```
// 硬币的面值
const int V[6] = {1, 5, 10, 50, 100, 500};

// 输入
int C[6]; // C[0] = C_1, C[1] = C_5, ...
int A;

void solve() {
  int ans = 0;

  for (int i = 5; i >= 0; i--) {
    int t = min(A / V[i], C[i]); // 使用硬币i的枚数
    A -= t * V[i];
    ans += t;
  }

  printf("%d\n", ans);
}
```

2.2.2　区间问题

现在，让我们来考虑一下如下问题：

区间调度问题

有 n 项工作，每项工作分别在 s_i 时间开始，在 t_i 时间结束。对于每项工作，你都可以选择参与与否。如果选择了参与，那么自始至终都必须全程参与。此外，参与工作的时间段不能重叠（即使是开始的瞬间和结束的瞬间的重叠也是不允许的）。

工作的例子　　参与工作的例子

你的目标是参与尽可能多的工作，那么最多能参与多少项工作呢？

限制条件

- $1 \leqslant N \leqslant 100000$
- $1 \leqslant s_i \leqslant t_i \leqslant 10^9$

样例

输入

```
n = 5, s = {1, 2, 4, 6, 8}, t = {3, 5, 7, 9, 10}
（对应上图）
```

输出

```
3（选取工作1、3、5）
```

这个问题也可以通过贪心算法来求解，但不像前面的硬币问题那么简单。我们可以设计出各种各样的贪心算法，例如下面的算法就是其中最容易想到的一种。

- 在可选的工作（也就是和目前已选的工作都不重叠的工作）中，每次都选取开始时间最早的工作。

该算法有一些不能正确处理的情况，例如对于下面的情况，该算法就无法得到正确的结果。

不能正确处理的例子

因此，如果我们不慎重地选择一个正确的规则，就会得到错误的算法。另外，我们还能够想到下列几种算法。

(1) 在可选的工作中，每次都选取结束时间最早的工作。
(2) 在可选的工作中，每次都选取用时最短的工作。

(3) 在可选的工作中，每次都选取与最少可选工作有重叠的工作。

算法一是正确的，而其余两种都可以找到对应的反例。或者说，在有些情况下，它们所选取的工作并非最优。

算法二的反例

算法三的反例

```
const int MAX_N = 100000;

// 输入
int N, S[MAX_N], T[MAX_N];

// 用于对工作排序的pair数组
pair<int, int> itv[MAX_N];

void solve() {
  // 对pair进行的是字典序比较
  // 为了让结束时间早的工作排在前面，把T存入first，把S存入second
  for (int i = 0; i < N; i++) {
    itv[i].first = T[i];
    itv[i].second = S[i];
  }
  sort(itv, itv + N);

  // t是最后所选工作的结束时间
  int ans = 0, t = 0;
  for (int i = 0; i < N; i++) {
    if (t < itv[i].second) {
      ans++;
      t = itv[i].first;
    }
  }

  printf("%d\n", ans);
}
```

专栏 贪心算法的证明

结束时间越早之后可选的工作也就越多。这是该算法能够正确处理问题的一个直观解释。但是，这不能算是严格意义上的证明。我们可以按下面的方法来证明。

(1) 与其他选择方案相比，该算法的选择方案在选取了相同数量的更早开始的工作时，其最终结束时间不会比其他方案的更晚。

(2) 所以，不存在选取更多工作的选择方案。

(1) 使用归纳法，(2) 使用反证法，就可以完成严格意义上的证明（证明过程较长，在此不再赘述）。

在程序设计竞赛中，只要程序能够正确运行就好了，严格意义上的算法证明并不是必须的。因此可以说，有足够自信的话是不需要证明的。但是，如果不能坚信算法是正确的，当程序不能正确运行时，就会搞不明白到底是算法设计有问题还是程序实现有问题。有时候在头脑中简单地思考一下证明也是挺好的。

2.2.3 字典序最小问题

Best Cow Line （POJ 3617）

给定长度为 N 的字符串 S，要构造一个长度为 N 的字符串 T。起初，T 是一个空串，随后反复进行下列任意操作。

- 从 S 的头部删除一个字符，加到 T 的尾部
- 从 S 的尾部删除一个字符，加到 T 的尾部

目标是要构造字典序[1]尽可能小的字符串 T。

操作示例

限制条件

- $1 \leqslant N \leqslant 2000$
- 字符串 S 只包含大写英文字母

① 字典序是指从前到后比较两个字符串大小的方法。首先比较第1个字符，如果不同则第1个字符较小的字符串更小，如果相同则继续比较第2个字符……如此继续，来比较整个字符串的大小。

样例

输入

```
N = 6
S = "ACDBCB"
```

输出

ABCBCD（如下图所示进行操作）

输入数据对应的最优操作

从字典序的性质上看，无论T的末尾有多大，只要前面部分的较小就可以。所以我们可以试一下如下贪心算法：

- 不断取S的开头和末尾中较小的一个字符放到T的末尾。

这个算法已经接近正确了，只是针对S的开头和末尾字符相同的情形还没有定义。在这种情形下，因为我们希望能够尽早使用更小的字符，所以就要比较下一个字符的大小。下一个字符也有可能相同，因此就有如下算法：

- 按照字典序比较S和将S反转后的字符串S'。
- 如果S较小，就从S的开头取出一个文字，追加到T的末尾。
- 如果S'较小，就从S的末尾取出一个文字，追加到T的末尾。
  （如果相同则取哪个都可以）

根据前面提到的性质，字典序比较类的问题经常能用得上贪心法。

```
// 输入
int N;
char S[MAX_N + 1];

void solve() {
  // 剩余的字符串为S[a], S[a+1], ..., S[b]
  int a = 0, b = N - 1;

  while (a <= b) {
    // 将从左起和从右起的字符串比较
    bool left = false;
```

```
    for (int i = 0; a + i <= b; i++) {
      if (S[a + i] < S[b - i]) {
        left = true;
        break;
      }
      else if (S[a + i] > S[b - i]) {
        left = false;
        break;
      }
    }

    if (left) putchar(S[a++]);
    else putchar(S[b--]);
  }

  putchar('\n');
}
```

①

2.2.4 其他例题

1. Saruman's Army

Saruman's Army （POJ 3069）

直线上有 N 个点。点 i 的位置是 X_i。从这 N 个点中选择若干个，给它们加上标记。对每一个点，其距离为 R 以内的区域里必须有带有标记的点（自己本身带有标记的点，可以认为与其距离为 0 的地方有一个带有标记的点）。在满足这个条件的情况下，希望能为尽可能少的点添加标记。请问至少要有多少点被加上标记？

添加标记的例子

限制条件

- $1 \leqslant N \leqslant 1000$
- $0 \leqslant R \leqslant 1000$
- $0 \leqslant X_i \leqslant 1000$

① 原始的POJ3617题目要求输出的字符串按照80字符换行，这里的代码还是行不通的。

样例

输入

```
N = 6
R = 10
X = {1, 7, 15, 20, 30, 50}
```

输出

```
3（如上图所示）
```

我们从最左边开始考虑。对于这个点，到距其R以内的区域内必须要有带有标记的点。（此点位于最左边，所以显然）带有标记的这个点一定在此点右侧（包含这个点自身）。

于是，究竟要给哪个点加上标记呢？答案应该是从最左边的点开始，距离为R以内的最远的点。因为更左的区域没有覆盖的意义，所以应该尽可能覆盖更靠右的点。

对于最左边的点的观察

如上所示，加上了第一个标记后，剩下的部分也用同样的办法处理。对于添加了符号的点右侧相距超过R的下一个点，采用同样的方法找到其右侧R距离以内最远的点添加标记。在所有的点都被覆盖之前不断地重复这一过程。

反复操作的情况

```
int N, R;
int X[MAX_N];

void solve() {
  sort(X, X + N);

  int i = 0, ans = 0;
  while (i < N) {
    // s是没有被覆盖的最左的点的位置
    int s = X[i++];
```

```
    // 一直向右前进直到距s的距离大于R的点
    while (i < N && X[i] <= s + R) i++;

    // p是新加上标记的点的位置
    int p = X[i - 1];
    // 一直向右前进直到距p的距离大于R的点
    while (i < N && X[i] <= p + R) i++;

    ans++;
  }

  printf("%d\n", ans);
}
```

2. Fence Repair

Fence Repair （POJ 3253）

农夫约翰为了修理栅栏，要将一块很长的木板切割成 N 块。准备切成的木板的长度为 L_1、L_2、…、L_N，未切割前木板的长度恰好为切割后木板长度的总和。每次切断木板时，需要的开销为这块木板的长度。例如长度为 21 的木板要切成长度为 5、8、8 的三块木板。长 21 的木板切成长为 13 和 8 的板时，开销为 21。再将长度为 13 的板切成长度为 5 和 8 的板时，开销是 13。于是合计开销是 34。请求出按照目标要求将木板切割完最小的开销是多少。

限制条件

- $1 \leqslant N \leqslant 20000$
- $0 \leqslant L_i \leqslant 50000$

样例

输入

```
N = 3, L = {8, 5, 8}
```

输出

```
34（对应于题目中的例子）
```

由于木板的切割顺序不确定，自由度很高，这个题目貌似很难入手。但是其实可以用略微奇特的贪心法来求解。

首先，切割的方法可以参见下图的二叉树来描述。二叉树的介绍详见2.4节，如果对这一概念不了解可以先移步学习一下。

对应切割方法的二叉树的例子

这里每一个叶子节点就对应了切割出的一块块木板。叶子节点的深度就对应了为了得到对应木板所需的切割次数，开销的合计就是各叶子节点的

木板的长度×节点的深度

的总和。例如，上图示例的全部开销就可以这样计算：

$3\times2+4\times2+5\times2+1\times3+2\times3=33$

于是，此时的最佳切割方法首先应该具有如下性质：

最短的板与次短的板的节点应当是兄弟节点

兄弟

对于最优解来讲，最短的板应当是深度最大的叶子节点之一。所以与这个叶子节点同一深度的兄弟节点一定存在，并且由于同样是最深的叶子节点，所以应该对应于次短的板。

不妨将L_i按照大小顺序排列，那么最短的板应该是L_1而次短的则是L_2。如果它们在二叉树中是兄弟节点，就意味着它们是从一块长度为(L_1+L_2)的板切割得来的。由于切割顺序是自由的，不妨当作是最后被切割。这样一来，在这次切割前就有

$$(L_1+L_2), L_3, L_4, \cdots, L_N$$

这样的N–1块木板存在。与以上讨论的方式相同，递归地将这N–1块木板的问题求解，就可以求出整个问题的答案。这样实现的话，虽然复杂度是$O(N^2)$，对于题目的输入规模来说，已经足以在时间限制内通过了。不过本题可以用$O(N\log N)$的时间求解，我们将在2.4节进行介绍。

```
typedef long long ll;

// 输入
int N, L[MAX_N];

void solve() {
  ll ans = 0;

  // 直到计算到木板为1块时为止
  while (N > 1) {
    // 求出最短的板mii1和次短的板mii2
    int mii1 = 0, mii2 = 1;
    if (L[mii1] > L[mii2]) swap(mii1, mii2);

    for (int i = 2; i < N; i++) {
      if (L[i] < L[mii1]) {
        mii2 = mii1;
        mii1 = i;
      }
      else if (L[i] < L[mii2]) {
        mii2 = i;
      }
    }

    // 将两块板拼合
    int t = L[mii1] + L[mii2];
    ans += t;

    if (mii1 == N - 1) swap(mii1, mii2);
    L[mii1] = t;
    L[mii2] = L[N - 1];
    N--;
  }

  printf("%lld\n", ans);
}
```

专栏　霍夫曼（Huffman）编码

这个问题的解法作为计算霍夫曼编码的算法而被熟知。例如将字符用 0 和 1 组成的序列对应起来后，一篇文章可以用一连串 0 和 1 的序列来表达。一般的字符编码就是这些对应关系当中的一种。但是，不同的字符在文章中出现的频度会有所不同，由此将频度比较高的'e'对应到较短的序列，而将频度比较低的'z'对应到较长的序列的话，文章可以用更短的序列来表达。

霍夫曼编码就是这类编码方法中的一种。在上述算法中，将木板换成字符，长度换成频度就可以了。这样，在生成的二叉树中，从根出发，走向左边就将 0，走向右边就将 1 追加到编码的末尾，到达每个叶子节点时就能得到该节点对应的码字了。

a	110
b	111
c	00
d	01
e	10

二叉树对应到编码的例子

霍夫曼编码是当码字长度为整数情况时最优的编码方案。

2.3 记录结果再利用的“动态规划”

➠动态规划（DP: Dynamic Programming）是算法的设计方法之一，在程序设计竞赛中经常被选作题材。在此，我们考察一些经典的DP问题，来看看DP究竟是何种类型的算法。

2.3.1 记忆化搜索与动态规划

01 背包问题

有 n 个重量和价值分别为 w_i, v_i 的物品。从这些物品中挑选出总重量不超过 W 的物品，求所有挑选方案中价值总和的最大值。

限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant w_i, v_i \leqslant 100$
- $1 \leqslant W \leqslant 10000$

样例

输入

```
n = 4
(w, v) = {(2, 3), (1, 2), (3, 4), (2, 2)}
W = 5
```

输出

```
7（选择第0、1、3号物品）
```

这是被称为背包问题的一个著名问题。这个问题要如何求解比较好呢？不妨先用最朴素的方法，针对每个物品是否放入背包进行搜索试试看。这个想法实现后的结果请参见如下代码：

```
// 输入
int n, W;
int w[MAX_N], v[MAX_N];

// 从第i个物品开始挑选总重小于j的部分
```

```
int rec(int i, int j) {
  int res;
  if (i == n) {
    // 已经没有剩余物品了
    res = 0;
  } else if (j < w[i]) {
    // 无法挑选这个物品
    res = rec(i + 1, j);
  } else {
    // 挑选和不挑选的两种情况都尝试一下
    res = max(rec(i + 1, j), rec(i + 1, j - w[i]) + v[i]);
  }
  return res;
}

void solve() {
  printf("%d\n", rec(0, W));
}
```

只不过，这种方法的搜索深度是n，而且每一层的搜索都需要两次分支，最坏就需要$O(2^n)$的时间，当n比较大时就没办法解了。所以要怎么办才好呢？为了优化之前的算法，我们看一下针对样例输入的情形下rec递归调用的情况。

递归地调用

如图所示，rec以(3,2)为参数调用了两次。如果参数相同，返回的结果也应该相同，于是第二次调用时已经知道了结果却白白浪费了计算时间。让我们在这里把第一次计算时的结果记录下来，省略掉第二次以后的重复计算试试看。

```
int dp[MAX_N + 1][MAX_W + 1]; // 记忆化数组

int rec(int i, int j) {
  if (dp[i][j] >= 0) {
    // 已经计算过的话直接使用之前的结果
    return dp[i][j];
  }
  int res;
  if (i == n) {
    res = 0;
  } else if (j < w[i]) {
```

```
    res = rec(i + 1, j);
  } else {
    res = max(rec(i + 1, j), rec(i + 1, j - w[i]) + v[i]);
  }
  // 将结果记录在数组中
  return dp[i][j] = res;
}

void solve() {
  // 用-1表示尚未计算过，初始化整个数组
  memset(dp, -1, sizeof(dp));
  printf("%d\n", rec(0, W));
}
```

这微小的改进能降低多少复杂度呢？对于同样的参数，只会在第一次被调用到时执行递归部分，第二次之后都会直接返回。参数的组合不过nW种，而函数内只调用2次递归，所以只需要$O(nW)$的复杂度就能解决这个问题。只需略微改良，可解的问题的规模就可以大幅提高。这种方法一般被称为记忆化搜索。

专栏　使用memset进行初始化

虽然 memset 按照 1 字节为单位对内存进行填充，–1 的每一位二进制位都是 1，所以可以像 0 一样用 memset 进行初始化。通过使用 memset 可以快速地对高维数组等进行初始化，但是需要注意无法初始化成 1 之类的数值。

专栏　穷竭搜索的写法

如果对记忆化搜索还不是很熟练的话，可能会把前面的搜索写成下面这样

```
// 目前选择的物品价值总和是sum，从第i个物品之后的物品中挑选重量总和小于j的物品
int rec(int i, int j, int sum) {
  int res;
  if (i == n) {
    // 已经没有剩余物品了
    res = sum;
  } else if (j < w[i]) {
    // 无法挑选这个物品
    res = rec(i + 1, j, sum);
  } else {
    // 挑选和不挑选的两种情况都尝试一下
    res = max(rec(i + 1, j, sum), rec(i + 1, j - w[i], sum + v[i]));
  }
  return res;
}
```

在需要剪枝的情况下，可能会像这样把各种参数都写在函数上，但是在这种情况下会让记忆化搜索难以实现，需要注意。

接下来，我们来仔细研究一下前面的算法利用到的这个记忆化数组。记$dp[i][j]$为根据rec的定义，从第i个物品开始挑选总重小于j时，总价值的最大值。于是我们有如下递推式

$$dp[n][j]=0$$

$$dp[i][j]=\begin{cases} dp[i+1][j] & (j<w[i]) \\ \max(dp[i+1][j], dp[i+1][j-w[i]]+v[i]) & (\text{其他}) \end{cases}$$

如上所示，不用写递归函数，直接利用递推式将各项的值计算出来，简单地用二重循环也可以解决这一问题。

i \ j	0	1	2	3	4	5
0	-	-	-	-	-	-
1	-	-	-	-	-	-
2	-	-	-	-	-	-
3	0	0	2	2	-	-
4	0	0	0	0	0	0

dp[3][3]=max(dp[4][3], dp[4][1]+2)

i \ j	0	1	2	3	4	5
0	0	2	3	5	6	7
1	0	2	2	4	6	6
2	0	0	2	4	4	6
3	0	0	2	2	2	2
4	0	0	0	0	0	0

dp[0][5]=max(dp[1][5], dp[1][3]+3)

```
int dp[MAX_N + 1][MAX_W + 1]; / DP数组

void solve() {
  for (int i = n - 1; i >= 0; i--) {
    for (int j = 0; j <= W; j++) {
      if (j < w[i]) {
        dp[i][j] = dp[i + 1][j];
      } else {
        dp[i][j] = max(dp[i + 1][j], dp[i + 1][j - w[i]] + v[i]);
      }
    }
  }
  printf("%d\n", dp[0][W]);
}
```

这个算法的复杂度与前面相同，也是$O(nW)$，但是简洁了很多。以这种方式一步步按顺序求出问题的解的方法被称作动态规划法，也就是常说的DP。解决问题时既可以按照如上方法从记忆化搜索出发推导出递推式，熟练后也可以直接得出递推式。

专栏　注意不要忘记初始化

因为全局数组的内容会被初始化为 0，所以前面的源代码中并没有显式地将初项=0 进行赋值，不过当一次运行要处理多组输入数据时，必须要进行初始化，这点一定要注意。

专栏 各种各样的DP

刚刚讲到 DP 中关于 i 的循环是逆向进行的。反之，如果按照如下的方式定义递推关系的话，关于 i 的循环就能正向进行。

$dp[i+1][j]$:= 从前 i 个物品中选出总重量不超过 j 的物品时总价值的最大值

$dp[0][j]=0$

$$dp[i+1][j]=\begin{cases} dp[i][j] & (j<w[i]) \\ \max(dp[i][j], dp[i][j-w[i]]+v[i]) & (\text{其他}) \end{cases}$$

N	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	2	3	5	5	5
3	0	2	3	5	6	-
4	-	-	-	-	-	-

dp[3][4]=max(dp[2][4], dp[2][1]+4)

```
void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      if (j < w[i]) {
        dp[i + 1][j] = dp[i][j];
      } else {
        dp[i + 1][j] = max(dp[i][j], dp[i][j - w[i]] + v[i]);
      }
    }
  }
  printf("%d\n", dp[n][W]);
}
```

此外，除了运用递推方式逐项求解之外，还可以把状态转移想象成从“前i个物品中选取总重不超过j时的状态”向“前i+1个物品中选取总重不超过j”和“前i+1个物品中选取总重不超过j+$w[i]$时的状态”的转移，于是可以实现成如下形式：

N	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	2	3	5	5	5
3	0	2	0	4	6	0
4	0	0	0	0	0	0

dp[3][1]=max(dp[3][1],dp[2][1]), dp[3][4]=max(dp[3][4],dp[2][1]+4)

```
void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      dp[i + 1][j] = max(dp[i + 1][j], dp[i][j]);
```

```
        if (j + w[i] <= W) {
          dp[i + 1][j + w[i]] = max(dp[i + 1][j + w[i]], dp[i][j] + v[i]);
        }
      }
    }
    printf("%d\n", dp[n][W]);
  }
```

如果像上述所示，把问题写成从当前状态迁移成下一状态的形式的话，需要注意初项之外也需要初始化（这个问题中，因为价值的总和至少是0，所以初值设为0就可以了，不过根据问题也有可能需要初始化成无穷大）。同一个问题可能会有各种各样的解决方法，诸如搜索的记忆化或者利用递推关系的DP，再或者从状态转移考虑的DP等，不妨先把自己最喜欢的形式掌握熟练。但是，有些问题不用记忆化搜索也许很难求解，反之，不用DP复杂度就会变大的情况也会有，所以最好要掌握各种形式的DP。

最长公共子序列问题

给定两个字符串 $s_1s_2\cdots s_n$ 和 $t_1t_2\cdots t_n$。求出这两个字符串最长的公共子序列的长度。字符串 $s_1s_2\cdots s_n$ 的子序列指可以表示为 $s_{i_1}s_{i_2}\cdots s_{i_m}$ $(i_1<i_2<\cdots<i_m)$ 的序列。

限制条件

- $1 \leqslant n, m \leqslant 1000$

样例

输入

```
n = 4
m = 4
s = "abcd"
t = "becd"
```

输出

```
3 ("bcd")
```

这个问题是被称为最长公共子序列问题（LCS，Longest Common Subsequence）的著名问题。不妨用如下方式定义试试看：

$dp[i][j]$:=$s_1\cdots s_i$和$t_1\cdots t_j$对应的LCS的长度

由此，$s_1\cdots s_{i+1}$和$t_1\cdots t_{j+1}$对应的公共子列可能是

当$s_{i+1}=t_{j+1}$时，在$s_1\cdots s_i$和$t_1\cdots t_j$的公共子列末尾追加上s_{i+1}

$s_1\cdots s_i$和$t_1\cdots t_{j+1}$的公共子列

$s_1\cdots s_{i+1}$和$t_1\cdots t_j$的公共子列

三者中的某一个，所以就有如下的递推关系成立[①]

$$dp[i+1][j+1]=\begin{cases}\max(dp[i][j]+1, dp[i][j+1], dp[i+1][j]) & (s_{i+1}=t_{j+1})\\ \max(dp[i][j+1], dp[i+1][j]) & (\text{其他})\end{cases}$$

这个递推式可用$O(nm)$计算出来，$dp[n][m]$就是LCS的长度。

j\i	0	1(b)	2(e)	3(c)	4(d)
0	0	0	0	0	0
1(a)	0	0	0	0	0
2(b)	0	1	1	1	1
3(c)	0	1	1	2	2
4(d)	0	1	1	2	3

DP数组

```
// 输入
int n, m;
char s[MAX_N], t[MAX_M];

int dp[MAX_N + 1][MAX_M + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (s[i] == t[j]) {
        dp[i + 1][j + 1] = dp[i][j] + 1;
      } else {
        dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j]);
      }
    }
  }
  printf("%d\n", dp[n][m]);
}
```

2.3.2 进一步探讨递推关系

完全背包问题

有 n 种重量和价值分别为 w_i, v_i 的物品。从这些物品中挑选总重量不超过 W 的物品，求出挑选物品价值总和的最大值。在这里，每种物品可以挑选任意多件。

限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant w_i, v_i \leqslant 100$
- $1 \leqslant W \leqslant 10000$

① 如果稍微思考一下就能发现$s_{i+1}=t_{j+1}$时，只需令$dp[i+1][j+1]=dp[i][j]+1$就可以了。

样例

输入

```
n = 3
(w, v) = {(3, 4), (4, 5), (2, 3)}
W = 7
```

输出

```
10（0号物品选1个，2号物品选2个）
```

这次同一种类的物品可以选择任意多件了。我们再试着写出递推关系。

令$dp[i+1][j]$:=从前i种物品中挑选总重量不超过j时总价值的最大值。那么递推关系为：

$$dp[0][j]=0$$
$$dp[i+1][j]=\max\{dp[i-k\times w[i]]+k\times v[i]\mid 0\leqslant k\}$$

让我们试着编写一下按照这个递推关系求解的程序：

```
int dp[MAX_N + 1][MAX_W + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      for (int k = 0; k * w[i] <= j; k++) {
        dp[i + 1][j] = max(dp[i + 1][j], dp[i][j - k * w[i]] + k * v[i]);
      }
    }
  }
  printf("%d\n", dp[n][W]);
}
```

这次的程序成了三重循环。关于k的循环最坏可能从0循环到W，所以这个算法的复杂度为$O(nW^2)$，这样并不够好。

我们来找一下这个算法中多余的计算（即已经知道结果的计算）。

i \ j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	8	8
2	0	0	0	4	5	5	8	9
3	0	0	3	4	6	7	9	10

DP数组

在$dp[i+1][j]$的计算中选择$k(k\geqslant 1)$个的情况，与在$dp[i+1][j-w[i]]$的计算中选择$k-1$个的情况是相同

的，所以$dp[i+1][j]$的递推中$k \geqslant 1$部分的计算已经在$dp[i+1][j-w[i]]$的计算中完成了。那么可以按照如下方式进行变形：

$$
\begin{aligned}
&\max\{dp[i][j-k\times w[i]]+k\times v[i] \mid 0 \leqslant k\} \\
&=\max(dp[i][j],\max\{dp[i][j-k\times w[i]]+k\times v[i] \mid 1 \leqslant k\}) \\
&=\max(dp[i][j],\max\{dp[i][(j-w[i])-k\times w[i]+k\times v[i] \mid 0 \leqslant k\}+v[i]) \\
&=\max(dp[i][j],dp[i+1][j-w[i]]+v[i])
\end{aligned}
$$

这样一来就不需要关于k的循环了，便可以用$O(nW)$时间解决问题。

N	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	8	8
2	0	0	0	4	5	5	8	9
3	0	0	3	4	6	7	9	10

DP数组

```
void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      if (j < w[i]) {
        dp[i + 1][j] = dp[i][j];
      } else {
        dp[i + 1][j] = max(dp[i][j], dp[i + 1][j - w[i]] + v[i]);
      }
    }
  }
  printf("%d\n", dp[n][W]);
}
```

此外，此前提到的01背包问题和这里的完全背包问题，可以通过不断重复利用一个数组来实现。

01背包问题的情况

```
int dp[MAX_W + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = W; j >= w[i]; j--) {
      dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
  }
  printf("%d\n", dp[W]);
}
```

完全背包问题的情况

```
int dp[MAX_W + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = w[i]; j <= W; j++) {
      dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
  }
  printf("%d\n", dp[W]);
}
```

像这样书写的话，两者的差异就变成只有循环的方向了。重复利用数组虽然可以节省内存空间，但使用得不好将有可能留下bug，所以要格外小心。不过出于节约内存的考虑，有时候必须要重复利用数组。也存在通过重复利用能够进一步降低复杂度的问题。这些我们会在后面介绍。

专栏　DP数组的再利用

除上面的情况之外，还有可能通过将两个数组滚动使用来实现重复利用。例如此前的

```
dp[i+1][j]=max(dp[i][j], dp[i+1][j-w[i]]+v[i])
```

这一递推式中，$dp[i+1]$计算时只需要 $dp[i]$和 $dp[i+1]$，所以可以结合奇偶性写成如下形式：

```
int dp[2][MAX_W + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      if (j < w[i]) {
        dp[(i + 1) & 1][j] = dp[i & 1][j];
      } else {
        dp[(i + 1) & 1][j] = max(dp[i & 1][j], dp[(i + 1) & 1][j - w[i]] + v[i]);
      }
    }
  }
  printf("%d\n", dp[n & 1][W]);
}
```

01 背包问题之 2

有 n 个重量和价值分别为 w_i, v_i 的物品。从这些物品中挑选总重量不超过 W 的物品，求所有挑选方案中价值总和的最大值。

限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant w_i \leqslant 10^7$
- $1 \leqslant v_i \leqslant 100$
- $1 \leqslant W \leqslant 10^9$

样例

输入

```
n = 4
(w, v) = {(2, 3), (1, 2), (3, 4), (2, 2)}
W = 5
```

输出

```
7（选择第0、1、3号物品）
```

这一问题与最初的01背包问题相比，只是修改了限制条件的大小。此前求解这一问题的方法的复杂度是$O(nW)$，对于这一问题的规模来讲就不够用了。在这个问题中，相比较重量而言，价值的范围比较小，所以可以试着改变DP的对象。之前的方法中，我们用DP针对不同的重量限制计算最大的价值。这次不妨用DP针对不同的价值计算最小的重量。

定义$dp[i+1][j]$:=前i个物品中挑选出价值总和为j时总重量的最小值（不存在时就是一个充分大的数值INF）。由于前0个物品中什么都挑选不了，所以初始值为

```
dp[0][0]=0
dp[0][j]=INF
```

此外，前i个物品中挑选出价值总和为j时，一定有

前$i-1$个物品中挑选价值总和为j的部分

前$i-1$个物品中挑选价值总和为$j-v[i]$的部分，然后再选中第i个物品

这两种方法之一，所以就能得到

$$dp[i+1][j]=\min(dp[i][j],dp[i][j-v[i]]+w[i])$$

这一递推式。最终的答案就对应于令$dp[n][j]\leqslant W$的最大的j。

通过这样求解，复杂度就成了$O(n\Sigma_i v_i)$，对此题限制条件下的输入就可以在时间限制内解出了。当然，如果价值变大了的话，这里的算法也变得不可行了。也就会有像这样需要依据问题的规模来改变算法的情况存在。

```
int dp[MAX_N + 1][MAX_N * MAX_V + 1]; // DP数组

void solve() {
  fill(dp[0], dp[0] + MAX_N * MAX_V + 1, INF);
  dp[0][0] = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= MAX_N * MAX_V; j++) {
```

```
      if (j < v[i]) {
        dp[i + 1][j] = dp[i][j];
      } else {
        dp[i + 1][j] = min(dp[i][j], dp[i][j - v[i]] + w[i]);
      }
    }
  }
  int res = 0;
  for (int i = 0; i <= MAX_N * MAX_V; i++) if (dp[n][i] <= W) res = i;
  printf("%d\n", res);
}
```

多重部分和问题

有 n 种不同大小的数字 a_i，每种各 m_i 个。判断是否可以从这些数字之中选出若干使它们的和恰好为 K。

限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant a_i, m_i \leqslant 100000$
- $1 \leqslant K \leqslant 100000$

样例

输入

```
n = 3
a = {3, 5, 8}
m = {3, 2, 2}
K = 17
```

输出

```
Yes (3*3+8=17)
```

这个问题可以用DP求解，不过如何定义递推关系会影响到最终的复杂度。首先我们看一下如下定义：

$dp[i+1][j]$:=用前i种数字是否能加和成j

为了用前i种数字加和成j，也就需要能用前$i-1$种数字加和成$j, j-a_i, \cdots, j-m_i \times a_i$中的某一种。由此我们可以定义如下递推关系：

$dp[i+1][j]$=($0 \leqslant k \leqslant m_i$且$k \times a_i \leqslant j$时存在使$dp[i][j-k \times a_i]$为真的$k$)

```
// 输入
int n; // 数列的长度
int K; // 目标的和数
int a[MAX_N]; // 值
int m[MAX_N]; // 个数

bool dp[MAX_N + 1][MAX_K + 1]; // DP数组

void solve() {
  dp[0][0] = true;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= K; j++) {
      for (int k = 0; k <= m[i] && k * a[i] <= j; k++) {
        dp[i + 1][j] |= dp[i][j - k * a[i]];
      }
    }
  }
  if (dp[n][K]) printf("Yes\n");
  else printf("No\n");
}
```

这个算法的复杂度是$O(K\Sigma_i m_i)$，这样并不够好。一般用DP求取bool结果的话会有不少浪费，同样的复杂度通常能获得更多的信息。在这个问题中，我们不光求出能否得到目标的和数，同时把得到时a_i这个数还剩下多少个可以使用计算出来，这样就可以减少复杂度。

$dp[i+1][j]$:=用前i种数加和得到j时第i种数最多能剩余多少个（不能加和得到i的情况下为-1）

按照如上所述定义递推关系，这样如果前$i-1$个数加和能得到j的话，第i个数就可以留下m_i个。此外，前i种数加和出$j-a_i$时第i种数还剩下$k(k>0)$的话，用这i种数加和j时第i种数就能剩下$k-1$个。由此我们能得出如下递推式。

$$dp[i+1][j]=\begin{cases} m_i & (dp[i][j]\geqslant 0) \\ -1 & (j<a_i\text{或者}dp[i+1][j-a_i]\leqslant 0) \\ dp[i+1][j-a_i]-1 & (\text{其他}) \end{cases}$$

这样，只要看最终是否满足$dp[n][K]\geqslant 0$就可以知道答案了。

这个递推式可以在$O(nK)$时间内计算出结果。再将数列重复利用的话，就得到了如下代码：

```
int dp[MAX_K + 1]; // DP数组

void solve() {
  memset(dp, -1, sizeof(dp));
  dp[0] = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= K; j++) {
      if (dp[j] >= 0) {
        dp[j] = m[i];
      } else if (j < a[i] || dp[j - a[i]] <= 0) {
```

```
        dp[j] = -1;
      } else {
        dp[j] = dp[j - a[i]] - 1;
      }
    }
  }
  if (dp[K] >= 0) printf("Yes\n");
  else printf("No\n");
}
```

最长上升子序列问题

有一个长为 n 的数列 $a_0, a_1, ..., a_{n-1}$。请求出这个序列中最长的上升子序列的长度。上升子序列指的是对于任意的 $i<j$ 都满足 $a_i<a_j$ 的子序列。

限制条件

- $1 \leqslant n \leqslant 1000$
- $0 \leqslant a_i \leqslant 1000000$

样例

输入

```
n = 5
a = {4, 2, 3, 1, 5}
```

输出

```
3（a₁，a₂，a₄构成的子序列2，3，5最长）
```

这个问题是被称作最长上升子序列（LIS，Longest Increasing Subsequence）的著名问题。这一问题通过使用DP也能很有效率地求解。我们首先来建立一下递推关系。

定义$dp[i]$:=以a_i为末尾的最长上升子序列的长度

以a_i结尾的上升子序列是

只包含a_i的子序列

在满足$j<i$并且$a_j<a_i$的以a_j为结尾的上升子列末尾，追加上a_i后得到的子序列

这二者之一。这样就能得到如下的递推关系：

$$dp[i]=\max\{1,dp[j]+1 | j<i \text{且} a_j<a_i\}$$

使用这一递推公式可以在$O(n^2)$时间内解决这个问题。

```
// 输入
int n;
int a[MAX_N];

int dp[MAX_N]; // DP数组

void solve() {
  int res = 0;
  for (int i = 0; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++) if (a[j] < a[i]) {
      dp[i] = max(dp[i], dp[j] + 1);
    }
    res = max(res, dp[i]);
  }
  printf("%d\n", res);
}
```

此外还可以定义其他的递推关系。前面我们利用DP求取针对最末位的元素的最长的子序列。如果子序列的长度相同，那么最末位的元素较小的在之后会更加有优势，所以我们再反过来用DP针对相同长度情况下最小的末尾元素进行求解。

$dp[i]$:=长度为i+1的上升子序列中末尾元素的最小值（不存在的话就是INF）

我们来看看如何用DP来更新这个数组。

最开始全部$dp[i]$的值都初始化为INF。然后由前到后逐个考虑数列的元素，对于每个a_j，如果i=0或者$dp[i-1]<a_j$的话，就用$dp[i]=\min(dp[i], a_j)$进行更新。最终找出使得$dp[i]<\mathrm{INF}$的最大的i+1就是结果了。这个DP直接实现的话，能够与前面的方法一样在$O(n^2)$的时间内给出结果，但这一算法还可以进一步优化。首先dp数列中除INF之外是单调递增的，所以可以知道对于每个a_j最多只需要1次更新。对于这次更新究竟应该在什么位置，不必逐个遍历，可以利用二分搜索，这样就可以在$O(n\log n)$时间内求出结果。

```
int dp[MAX_N]; // DP数组

void solve() {
  fill(dp, dp + n, INF);
  for (int i = 0; i < n; i++) {
    *lower_bound(dp, dp + n, a[i]) = a[i];
  }
  printf("%d\n", lower_bound(dp, dp + n, INF) - dp);
}
```

i	0	1	2	3	4
dp[i]	∞	∞	∞	∞	∞

↓ 插入4

i	0	1	2	3	4
dp[i]	4	∞	∞	∞	∞

↓ 插入2

i	0	1	2	3	4
dp[i]	2	∞	∞	∞	∞

↓ 插入3

i	0	1	2	3	4
dp[i]	2	3	∞	∞	∞

↓ 插入1

i	0	1	2	3	4
dp[i]	1	3	∞	∞	∞

↓ 插入5

i	0	1	2	3	4
dp[i]	1	3	5	∞	∞

DP数组的变化

专栏　lower_bound

上面的源代码中使用了 lower_bound 这个 STL 函数。这个函数从已排好序的序列 a 中利用二分搜索找出指向满足 $a_i \geqslant k$ 的 a_i 的最小的指针。类似的函数还有 upper_bound，这一函数求出的是指向满足 $a_i > k$ 的 a_i 的最小的指针。也许大家觉得这稍微有点复杂，但有了它们，比如长度为 n 的有序数组 a 中的 k 的个数，可以用下面的代码求出

```
upper_bound(a, a+n, k) - lower_bound(a, a+n, k)
```

所以使用熟练了会非常方便。

2.3.3　有关计数问题的DP

划分数

有 n 个无区别的物品，将它们划分成不超过 m 组，求出划分方法数模 M 的余数。

限制条件

- $1 \leqslant m \leqslant n \leqslant 1000$
- $2 \leqslant M \leqslant 10000$

样例

输入

```
n = 4
m = 3
M = 10000
```

输出

```
4 (1+1+2=1+3=2+2=4)
```

这样的划分被称作n的m划分，特别地，$m=n$时称作n的划分数[1]。DP不仅对于求解最优问题有效，对于各种排列组合的个数、概率或者期望之类的计算同样很有用。在此，我们定义如下。

$dp[i][j]$=j的i划分的总数

根据这一定义可以得到怎样的递推关系呢？将j分划分i个的话，可以先取出k个，然后将剩下的$j-k$个分成$i-1$份，这时大家是不是认为也许就可以得到下面的递推式了？

$$dp[i][j]=\sum_{k=0}^{j}dp[i-1][j-k]$$

但很不幸的是，这个递推是不正确的。用这个办法的话，例如1+1+2和1+2+1的划分就被当成是不同的划分来计数了[2]。为了不重复计数，我们需要寻找别的递推关系。考虑n的m划分$a_i(\sum_{i=1}^{m}a_i=n)$，如果对于每个i都有$a_i>0$，那么$\{a_i-1\}$就对应了$n-m$的m划分。另外，如果存在$a_i=0$，那么这就对应了n的$m-1$划分。综上，我们可得出了如下递推关系。

$$dp[i][j]=dp[i][j-i]+dp[i-1][j]$$

这个递推式可以不重复地计算所有的划分，复杂度为$O(nm)$。像这样需要在计数问题中解决重复计算问题时，需要特别小心。

```
// 输入
int n, m;

int dp[MAX_M + 1][MAX_N + 1]; // DP数组

void solve() {
  dp[0][0] = 1;
  for (int i = 1; i <= m; i++) {
    for (int j = 0; j <= n; j++) {
```

① 将基数为n的集合划分为恰好k个非空集的方法的数目称为第二类Stirling数；而将基数为n的集合划分为任意个空集的方法的数目称为Bell数。——译者注

② 即划分后装入m个不同的箱子。

```
      if (j - i >= 0) {
        dp[i][j] = (dp[i - 1][j] + dp[i][j - i]) % M;
      } else {
        dp[i][j] = dp[i - 1][j];
      }
    }
  }
  printf("%d\n", dp[m][n]);
}
```

多重集组合数

有 n 种物品，第 i 种物品有 a_i 个。不同种类的物品可以互相区分但相同种类的无法区分。从这些物品中取出 m 个的话，有多少种取法？求出方案数模 M 的余数。

限制条件

- $1 \leqslant n \leqslant 1000$
- $1 \leqslant m \leqslant 1000$
- $1 \leqslant a_i \leqslant 1000$
- $2 \leqslant M \leqslant 10000$

样例

输入

```
n = 3
m = 3
a = {1, 2, 3}
M = 10000
```

输出

```
6 (0+0+3, 0+1+2, 0+2+1, 1+0+2, 1+1+1, 1+2+0)
```

为了不重复计数，同一种类的物品最好一次性处理好。于是我们按照如下方式进行定义。

$dp[i+1][j]$:=从前i种物品中取出j个的组合总数

为了从前i种物品中取出j个，可以从前$i-1$种物品中取出$j-k$个，再从第i种物品中取出k个添加进来，所以可以得到如下递推关系

$$dp[i+1][j] = \sum_{k=0}^{\min(j,a[i])} dp[i][j-k]$$

直接计算这个递推关系的话复杂度是$O(nm^2)$，不过因为我们有

$$\sum_{k=0}^{\min(j,a[i])} dp[i][j-k] = \sum_{k=0}^{\min(j-1,a[i])} dp[i][j-1-k] + dp[i][j] - dp[i][j-1-a_i]$$

所以可以变形为如下形式

$$dp[i+1][j] = dp[i+1][j-1] + dp[i][j] - dp[i][j-1-a_i]$$

这样复杂度就下降到$O(nm)$了。

```
// 输入
int n, m;
int a[MAX_N];

int dp[MAX_N + 1][MAX_M + 1]; // DP数组

void solve() {
  // 一个都不取的方法总是只有一种
  for (int i = 0; i <= n; i++) {
    dp[i][0] = 1;
  }
  for (int i = 0; i < n; i++) {
    for (int j = 1; j <= m; j++) {
      if (j - 1 - a[i] >= 0) {
        // 在有取余的情况下，要避免减法运算的结果出现负数①
        dp[i + 1][j] = (dp[i + 1][j - 1] + dp[i][j] - dp[i][j - 1 - a[i]] + M) % M;
      } else {
        dp[i + 1][j] = (dp[i + 1][j - 1] + dp[i][j]) % M;
      }
    }
  }
  printf("%d\n", dp[n][m]);
}
```

① 通常，为了方便，我们在计算过程中只需要保证不会溢出，输出答案时再保证不会是负数。——译者注

2.4 加工并存储数据的数据结构

➠数据结构指的是存储数据的方式。用不同的方式存储数据，可以对数据做不同的高效操作。本节我们将会讨论“堆”、“二叉搜索树”、“并查集”这三种数据结构。

2.4.1 树和二叉树

节点、边、根、叶子的例子

父亲、兄弟、儿子的例子

图中的圆圈代表“节点”，线代表“边”。在这一节里，我们把树的根所表示的节点提到整棵树的最上面。每个节点在保存了各自的信息之外，都还拥有儿子节点。把自己作为儿子的节点叫做父亲节点。拥有同一父亲的节点互为兄弟节点。没有儿子的节点叫做叶子节点。“二叉树”是树中所有节点的儿子个数都不超过2的树。

二叉树的例子和非二叉树的树的例子

2.4.2 优先队列和堆

1. 优先队列

能够完成下列操作的数据结构叫做优先队列。

- 插入一个数值
- 取出最小的数值（获得数值，并且删除）

优先队列的示意图

能够使用二叉树高效地解决上述问题的，是一种叫做“堆”[①]的数据结构。

2. 堆的结构

堆就是像下图这样的二叉树。

堆的例子

堆最重要的性质就是儿子的值一定不小于父亲的值。除此之外，树的节点是按从上倒下、从左到右的顺序紧凑排列的。

① 严格来讲，堆也有不同的种类。这是一种叫做二叉堆的数据结构。

插入数值的例子

如上图所示，在向堆中插入数值时，首先在堆的末尾[1]插入该数值，然后不断向上提升直到没有大小颠倒为止。

取出最小值的例子

如上图所示，从堆中删除最小值时，首先把堆的最后一个节点的数值复制到根节点上，并且删除最后一个节点。然后不断向下交换直到没有大小颠倒为止。在向下交换的过程中，如果有2个儿子，那么选择数值较小的儿子（如果儿子比自己小的话）进行交换。

3. 堆的操作的复杂度

堆的两种操作所花的时间都和树的深度成正比。因此，如果一共有n个元素，那么每个操作可以在$O(\log n)$的时间内完成。

4. 堆的实现

下面我们来看一下堆的实现的例子。我们不使用指针来表示二叉树，而是如下图所示，给每个节点赋予一个编号，并用数组来存储。此时，儿子的编号就满足如下性质。

- 左儿子的编号是自己的编号×2+1
- 右儿子的编号是自己的编号×2+2

① 为了使数据按照从上倒下、从左到右的顺序紧凑排列，在最下面一层尽可能靠左的节点插入该数值。

用数组表示二叉树时的编号

```
int heap[MAX_N], sz = 0;
void push(int x) {
  // 自己节点的编号
  int i = sz++;

  while (i > 0) {
    // 父亲节点的编号
    int p = (i - 1) / 2;

    // 如果已经没有大小颠倒则退出
    if (heap[p] <= x) break;

    // 把父亲节点的数值放下来，而把自己提上去
    heap[i] = heap[p];
    i = p;
  }

  heap[i] = x;
}


int pop() {
  // 最小值
  int ret = heap[0];

  // 要提到根的数值
  int x = heap[--sz];

  // 从根开始向下交换
  int i = 0;
  while (i * 2 + 1 < sz) {
    // 比较儿子的值
    int a = i * 2 + 1, b = i * 2 + 2;
    if (b < sz && heap[b] < heap[a]) a = b;

    // 如果已经没有大小颠倒则退出
    if (heap[a] >= x) break;

    // 把儿子的数值提上来
    heap[i] = heap[a];
```

```
    i = a;
  }

  heap[i] = x;

  return ret;
}
```

5. 编程语言的标准库

实际上，大部分情况下并不需要自己实现堆。在许多编程语言的标准中，都包含了优先队列的高效实现。例如在C++中，STL里的priority_queue就是其中之一。不过需要注意的是，priority_queue与上面讲的优先队列有所不同，取出数值时得到的是最大值。下面是一些使用priority_queue的简单例子。

```
#include <queue>
#include <cstdio>
using namespace std;

int main() {
  // 声明
  priority_queue<int> pque;

  // 插入元素
  pque.push(3);
  pque.push(5);
  pque.push(1);

  // 不断循环直到空为止
  while (!pque.empty()) {
    // 获取并删除最大值
    printf("%d\n", pque.top());
    pque.pop();
  }

  return 0;
}
```

6. 需要运用优先队列的题目

Expedition （POJ 2431）

你需要驾驶一辆卡车行驶 L 单位距离。最开始时，卡车上有 P 单位的汽油。卡车每开 1 单位距离需要消耗 1 单位的汽油。如果在途中车上的汽油耗尽，卡车就无法继续前行，因而无法到达终点。在途中一共有 N 个加油站。第 i 个加油站在距离起点 A_i 单位距离的地方[①]，最多

① 实际上POJ 2431的输入数据中给的是加油站到终点的距离。在这里我们为了方便起见，改成了从起点到加油站的距离。