算法基础

打开程序设计之门

梁冰 冯林 刘胜蓝 / 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

算法是一系列解决问题的清晰指令，是程序设计的灵魂。同一问题可采用不同的算法解决，而一个算法的优劣将直接影响程序的执行效率。

本书以 ACM 程序设计竞赛的题目为基础，详细介绍一些常用的算法以及相关的理论知识，主要内容包括高级数据结构、字符串、动态规划进阶算法、图论高级算法、经典算法问题、组合数学、计算几何、组合游戏论。

本书适合计算机专业的学生以及对程序设计竞赛感兴趣的读者阅读。

本书提供源代码下载，读者可登录华信教育资源网（www.hxedu.com.cn）免费注册后下载。

图书在版编目（CIP）数据

算法基础：打开程序设计之门 / 梁冰，冯林，刘胜蓝编著. —北京：电子工业出版社，2019.1
ISBN 978-7-121-35868-5

Ⅰ. ①算… Ⅱ. ①梁… ②冯… ③刘… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字（2018）第 296484 号

责任编辑：田宏峰　　特约编辑：李秦华
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：17.5　字数：392 千字
版　　次：2019 年 1 月第 1 版
印　　次：2019 年 1 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：tianhf@phei.com.cn。

前　言

这是一本关于算法的教程。算法是一系列解决问题的清晰指令，可以说它是程序设计的灵魂。同一问题可用不同的算法解决，而一个算法的质量优劣将影响程序的执行效率。算法分析的目的在于选择合适算法和改进算法。评价一个算法的好坏主要是通过算法运行的时间长短和占用空间的大小来评估的。对于计算机专业或者爱好计算机的人士来说，无论学习还是工作，或多或少都会应用一些算法的知识。而目前国内外大型互联网公司在招聘时的笔试和面试都以算法为主。可见，算法的重要性是不言而喻的。

ACM/ICPC（ACM International Collegiate Programming Contest）是一项由美国计算机协会主办的，旨在展示大学生创新能力、团队精神和在压力下编写程序、分析和解决问题能力的年度竞赛。ACM 程序设计竞赛的题目强调算法的高效性与正确性。参赛选手只有编写出能够在规定时间内运行完成若干组数严格的测试数据，并且结果全部正确的程序才能得到分数。本书将以 ACM 程序设计竞赛的题目为基础，介绍一些经典的算法。

本书的目的是将更多对计算机算法感兴趣，但又苦于无从入手的读者带进程序设计的大门，让刚迈入大学校门的学生学会使用 C++语言解决简单的问题。本书主要介绍高级数据结构、字符串、动态规划、图论、组合数学方面的经典算法，相信当读者掌握了这些内容之后，会对算法和程序设计有一个新层次的认识，并会产生浓厚的兴趣。对于每个算法，本书都有图文并茂的讲解；在每章节的最后，都有针对该部分知识点的例题讲解，每道例题都是国内外著名程序在线判题系统中的原题，而且对于每道例题，都会从理解题意开始，详细讲解解题的思路，并附有完整的可以正确通过测试样例的代码，供读者研究学习。除了例题，在每章的最后还有一些练习题供读者巩固学到的知识，如果读者对这些习题仍感觉无从下手，可以参考每道练习题后附带的思路分析来帮助整理解题思路。

大连理工大学是在全国高校中较早倡导并开展创新创业教育的学校。自 1984 年以来，学校大力开展以突出创新创业实践为特色的创新创业教育。1995 年，在全国率先成立以学生创新创业教育为主体的教学改革示范区——创新教育实践中心，开展创新创业教育课程体系、教学内容、教学方法、教学模式等方面的改革，探索与之配套的管理运行机制，将创造性思维与创新方法融入教学实践中，在课堂教学中树立“CDIO 工程教育”新理念，倡导“做中学”，在实践环节构建了“个性化、双渠道、三结合、四层次、多模式”的创新教育实践教学新体系，取得了一系列成果，在全国高校产生了很大的影响。“创造性思维与创新方法”和“创新教育基础与实践（系列）”课程分别被评为国家级精品资源开放课程。“大学生程序设计竞赛初级教材”是“创新教育基础与实践”系列课程的核心课程，是面向大连理工大学 ACM 创新实践班的学生开设的。

此外，本书在撰写过程中，除了参考文献和正文中标出的引用来源，还参考了国内外的相关研究成果和网站资源，但没有一一列出，在此感谢所涉及的所有单位、专家和研究人员。

因编者水平有限，书中的错误和不足之处在所难免，欢迎广大读者来信批评指正，提出宝贵意见，帮助我们不断地完善本书。

编　者

2018 年 12 月

目　　录

第 1 章

高级数据结构

本章介绍一些高级数据结构。正确地选取数据结构能大大提高程序的效率。基础数据结构，如线性表（栈、队列、链表等）、二叉树、图等，是高级数据结构的基础。

堆是一种常见的数据结构。堆排序是利用堆这种数据结构所设计的一种选择排序。堆可以实现优先队列。树状数组可以简单高效地求得区间和。平衡二叉树是一棵平衡的二叉树，能让二叉树的操作维持在 $O(\log_2 N)$左右。Treap 通过随机数来优化二叉查找树（Binary Search Tree）防止其退化。Splay 树通过其特有的 Splay 操作来维持平衡。左倾堆是一种可并堆，具有神奇的“左倾”性质。

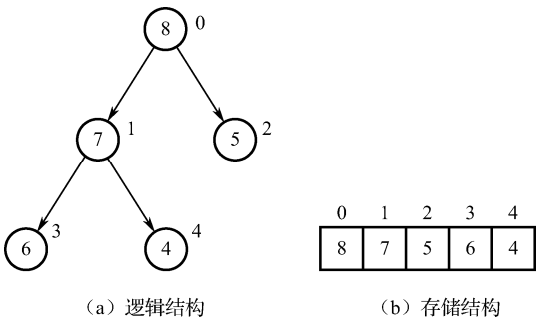
1.1 堆

本节介绍一种常见的数据结构——二叉堆（简称堆），通过图文讲解和代码实现来了解这个重要的数据结构。

1.1.1 堆的定义

堆（Heap）是一棵完全二叉树。其最重要的性质就是“儿子”的值一定不小于（或大于）“父亲”的值（分别称为小顶堆和大顶堆，下文使用大顶堆）。应用场景包括堆排序和优先队列等。用数组存堆时，对于任意一个节点 x，其左、右子节点的标号分别为 $2x+1$ 和 $2x+2$。

堆的逻辑结构和存储结构如图 1.1 所示。



（a）逻辑结构　　（b）存储结构

图 1.1　堆的逻辑结构和存储结构

1.1.2 建堆

建堆的核心内容是调整堆，使二叉树满足堆的定义（每个节点的值都不大于其父节点的

值）。调堆的过程应该从最后一个非叶子节点开始，一直调整到根节点。

堆排序过程示例如图 1.2 所示。

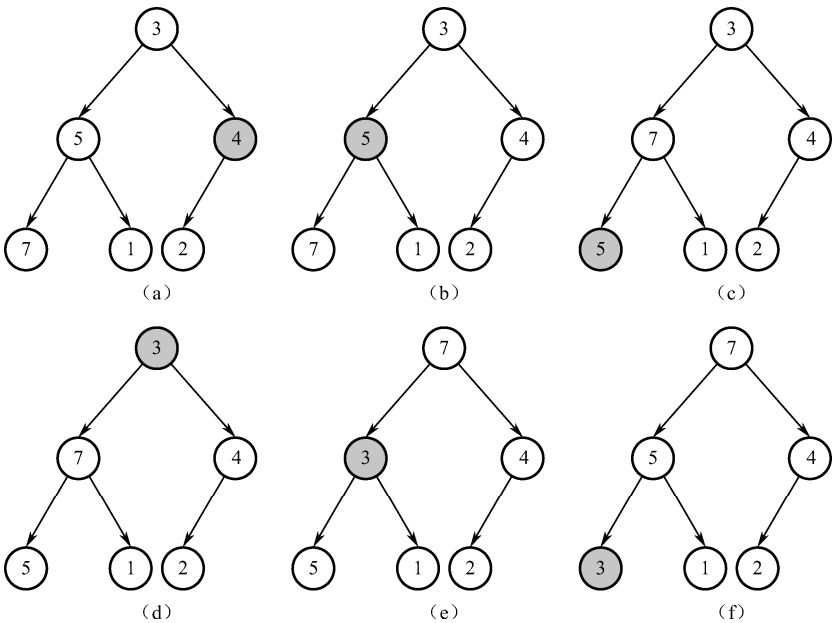


图 1.2　堆排序过程示例

1.1.3　堆排序算法

设堆有 n 个元素，每一次调整堆，堆顶位置将得到堆的最大值，然后将堆顶元素和最后一个元素互换，对前 n−1 个元素继续调整，直到整个堆有序为止。堆排序时间复杂度为 $O(\log_2 N)$。堆排序是不稳定排序。

代码实现如下所述。

```
#include <cstdio>

void heap_adjust(int arr[], int father, int n)
{
    //调整为大顶堆
    int child = father * 2 + 1;                    // 左子节点
    int tmp = arr[father];
    while (child < n) {
        //如果该节点有两个子节点，则取其中较大的一个，否则取左子节点
        if (child + 1 < n && arr[child] < arr[child + 1]) child++;
        if (arr[father] >= arr[child]) break;
```

```
12              //如果较大的子节点小于该节点，则不需要继续调整，退出
13              arr[father] = arr[child];              //否则交换该节点和较大的子节点，继续向下调整
14              father = child;
15              child = father * 2 + 1;
16              arr[father] = tmp;
17          }
18      }
19
20      void build_heap(int arr[], int n)
21      {
22          //建堆时从非叶子节点开始调整
23          for (int i = (n - 1) / 2; i >= 0; --i) {
24              heap_adjust(arr, i, n);
25          }
26      }
27
28      void heap_sort(int arr[], int beg, int end)
29      {
30          //堆排序，先建堆
31          //然后每一次交换未调整好的最后一个元素和第一个元素
32          build_heap(arr + beg, end - beg);
33          for (int tmp, i = end - 1; i > beg; --i) {
34              tmp = arr[i]; arr[i] = arr[0]; arr[0] = tmp;
35              heap_adjust(arr + beg, 0, i);
36          }
37      }
38
39      int main()
40      {
41          int arr[100];
42          int n; scanf("%d", &n);
43          for (int i = 0; i < n; ++i) scanf("%d", &arr[i]);
44          heap_sort(arr, 0, n);
45          for (int i = 0; i < n; ++i) printf("%d ", arr[i]);
46          return 0;
47      }
```

1.2　树状数组

树状数组（Binary Indexed Tree，BIT）能够高效地求序列区间和。树状数组的实现简单，

巧妙地运用了二进制的思想。

1.2.1 树状数组的定义

给定一个数组进行两个操作：一是更新某点的值；二是求某段区间的和。对于普通的数组，单点更新的复杂度为 $O(1)$，求区间和的复杂度为 $O(n)$，而树状数组能够把单点更新和区间求和的复杂度都变为 $O(n\log n)$。

设数组 $A[]$为原数组，定义 $C[i]=A[i-2^k+1]+\cdots+A[i]$。其中，$k$ 为 i 用二进制表示时的末尾 0 的个数，如 $C[1]=A[1]$，$C[2]=A[1]+A[2]$，$C[3]=A[3]$，$C[4]=A[1]+A[2]+A[3]+A[4]$，…。也就是说，$C[i]$就是从 $A[i]$开始前 2^k 项的和，如图 1.3 所示。

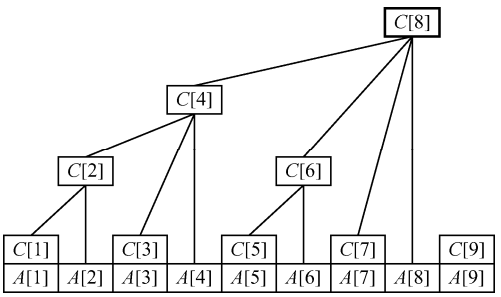


图 1.3　树状数组

1.2.2 树状数组的实现和使用

k 为 x 用二进制表示时末尾 0 的个数，对于 2^k，有一种快速的求解方法：

```
1    int lowbit(int x) {
2        return x & (-x);
3    }
```

当修改某一点的值时，只需要修改某一点的所有父节点就可以了。对于一个节点 i 来说，它的父节点的编号为 i+lowbit(i)。因此对于单点修改的函数为

```
1    //在 position 处加 value，len 是数组长度
2    void change(int c[], int position, int value, int len) {
3        while (position <= len) {
4            c[position] += value;
5            position += lowbit(position);
6        }
7    }
```

前 n 个数的和记为 sum(n)，已知 $C[i]$是从 i 开始向前 lowbit(i)个数的和，所以

sum(n)=C[n]+sum[n−lowbit(n)]。

```
//求前 n 个数的和
int sum(int c[], int n) {
    int answer = 0;
    while (n > 0) {
        answer += c[n];
        n -= lowbit(n);
    }
    return answer;
}
```

区间[start,end]的区间和可以通过 sum(end)−sum(start)来求得。树状数组也可以进行区间增减更新和单点查询操作。A[]是原数组，构造差分数组 D[]，令 D[1]=A[1]，D[i]=A[i]−A[i−1](i>1)，则数组 A 是数组 D 的前缀和，即 A[i]=D[1]+D[2]+⋯+D[i]。这样求 A 的单点值就是求 D 的区间和。更新 A 某段区间[start,end]的值只需要更改 D[start]和 D[end+1]的值就可以了。这两个操作可以通过构造 D 的树状数组求得。

1.2.3　例题讲解

例 1-1　Sort it

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

给定一个由 n 个不同的整数组成的序列，通过交换相邻的两个数，使得序列变成上升的。问最少需要交换多少次，如“1 2 3 5 4”，需要进行一次操作，交换 5 和 4。

输入：

输入包含多组数据，每一组数据包含两行，第一行为一个正整数 n（n≤1000），第二行为从 1 到 n 的 n 个整数的一个序列。

输出：

输出为 1 行，输出最少需要交换的次数。

样例输入：

3
1 2 3
4
4 3 2 1

样例输出：

0
6

题目来源：HDOJ 2689。

解题思路：

题目可以转化成求数组中逆序对的数量。

逆序对：设数组 A 为一个有 n 个数字的有序集（n>1），其中的数字均不相同，如果存在正整数 i，j，使得 $1 \leqslant i < j \leqslant n$ 而且 $A[i] > A[j]$，则<$A[i], A[j]$>这个有序对称为 A 的一个逆序对。

对于每一个数字，求它前面有多少个数字大于它，即该数字的贡献，所有数字的贡献和即逆序对的数量。对于 $A[i]=x$，在树状数组的 x 处加 1，然后求 sum(x)就是在 x 前面有多少个数小于等于 x，那么 i–sum(x)就是 $A[i]$的贡献。从左到右对于每一个数边插入边求和。

题目实现：

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

const int N = 1000;

int lowbit(int x) {
    return x & (-x);
}

void change(int c[], int position, int value, int len) {
    //…
}

int sum(int c[], int n) {
    //…
}

int main()
{
    int n;
    int c[N];                          //树状数组
    while (~scanf("%d", &n)) {         //序列包含多少个数字
        memset(c, 0, sizeof(c));
        int x;
        int answer = 0;
        for (int i = 1; i <= n; ++i) {
            scanf("%d", &x);           //读入每一个数字
            change(c, x, 1, n);        //在树状数组 x 处加 1
            answer += i - sum(c, x);   //把每一个数字的贡献相加
```

```
31                }
32                printf("%d\n", answer);
33            }
34            return 0;
35    }
```

1.3　左倾堆

前面介绍过堆，当需要合并两个堆时，往往效率较低。本节介绍的左倾堆能高效地实现两个堆的合并，称之为可并堆。左倾堆又称为左偏树、左偏堆、最左堆等。

1.3.1　左倾堆相关定义和性质

零距离（Null Path Length，NPL）是指从一个节点到一个“最近的不满节点”的路径长度。不满节点是指该节点的左、右子节点至少有一个为空。叶子节点的 NPL 为 0；空节点的 NPL 为−1。

在左倾堆中的每个节点均维护两个值——键值和零距离。左倾堆满足：

（1）节点的键值小于或等于它的子节点的键值，即堆性质。

（2）节点的左子节点的 NPL 大于或等于右子节点的 NPL，即左倾性质。

（3）节点的 NPL 等于它的右子节点的 NPL+1。

（4）左倾堆的任意子树也是左倾堆。

1.3.2　左倾堆的操作

（1）两个左倾堆的合并。将 *A* 和 *B* 两个左倾堆合并为 *C*，如果其中一个为空，只需返回另一棵树即可；当 *A* 和 *B* 都为非空时，假设 *A* 的根节点小于等于 *B* 的根节点（否则交换 *A*、*B*），把 *A* 的根节点作为新树 *C* 的根节点，然后合并 *A* 的右子树和 *B*，之后右子树的 NPL 可能会变大，当 *A* 的右子树的 NPL 大于其左子树的 NPL 时，左倾堆的左倾会被破坏。在这种情况下，只需要交换左、右子树。最后，由于 *A* 的右子树的 NPL 可能发生改变，必须更新 *A* 的 NPL：NPL（*A*）←NPL（*A* 的右子树）+1。

（2）插入新节点。单个节点也可以看成一个左倾堆，因此向左倾堆插入一个节点可以看成两个左倾堆的合并。

（3）删除最小节点。左倾堆的根节点就是最小节点，在删除根节点后，需要将两棵子树合并。

左倾堆的实现如下所述。

```
struct LHeap {
    int l, r, sz;
    int key, dis;
    bool operator<(const LHeap lh) const {
        return key < lh.key;
    }
} tr[N];
int cnt_tr;

int NewTree(int k) {
    tr[++cnt_tr].key = k;
    tr[cnt_tr].l = tr[cnt_tr].r = tr[cnt_tr].dis = 0;
    tr[cnt_tr].sz = 1;
    return cnt_tr;
}
//合并两个堆
int Merge(int x, int y) {
    if (!x || !y) return x + y;
    if (tr[x] < tr[y]) swap(x, y);
    tr[x].r = Merge(tr[x].r, y);
    if (tr[tr[x].l].dis < tr[tr[x].r].dis) swap(tr[x].l, tr[x].r);
    tr[x].dis = tr[tr[x].r].dis + 1;
    tr[x].sz = tr[tr[x].l].sz + tr[tr[x].r].sz + 1;
    return x;
}
//返回堆顶元素
int Top(int x) {
    return tr[x].key;
}
//删除根节点
void Pop(int &x) {
    x = Merge(tr[x].l, tr[x].r);
}
```

1.3.3 例题讲解

例 1-2 Financial Fraud

Time Limit: 3000 ms　　Memory Limit: 65536 KB

题目描述：

给定一个整数序列 $A_1,A_2,\cdots,A_N$，求一个非递减序列 $B_1,B_2,\cdots,B_N$（$1\leqslant i<j\leqslant N$，$B_i\leqslant B_j$），使得

风险值最小，即$|A_1-B_1|+|A_2-B_2|+\cdots+|A_N-B_N|$最小。

输入：

第一行输入 N（$N\leqslant 50000$），第二行输入 N 个整数表示序列 A（$-10^9\leqslant A_i\leqslant 10^9$），$N$ 等于 0 时结束输入。

输出：

最小风险值。

样例输入：

4

300 400 200 100

0

样例输出：

400

题目来源：ZOJ3512。

解题思路：

当 A 是一个非递减序列时，只需让 $B_i=A_i$ 即可；当 A 是一个非递增序列时，最优解是让 $B_1=B_2=\cdots=B_N=$序列 A 的中位数（中位数是指序列中第$\lfloor N/2\rfloor$大的数）；当 A 不是以上两种特殊情况时，可以考虑把序列 A 分为一些区间段，相对应的序列 B 为那一段的中位数。

假设已经找到前 k 个数 $A_1,A_2,\cdots,A_k$（$k<N$）的最优解，需要求前 $k+1$ 个数的最优解。先把第 $k+1$ 个数单独作为一个区间，则中位数就是 A_{k+1}，因为要求 B 是非递减序列，如果 A_{k+1} 大于或等于前一个区间的中位数，就保留 A_{k+1} 作为单独一个区间；否则，需要将 A_{k+1} 和前一个区间合并。

因为涉及区间合并，很容易想到本节介绍的可合并堆——左倾堆。如何用左倾堆维护区间中位数呢？只需要用大顶堆维护区间较小的一半元素，这样堆顶元素就是中位数。在每次从左向右求解时，可把每一个数单独建一个左倾堆，如果当前区间的中位数小于前一个区间的中位数时，就将两个左倾堆合并，并弹出堆顶多余的元素。

题目实现：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int N = 50005;

//左倾堆相关代码
```

```
10
11  int a[N], root[N], num[N];
12
13  int main() {
14      int n;
15      while (~scanf("%d",&n) && n) {
16          long long sum, tmp, ans;
17          cnt_tr = sum = tmp = 0;
18          for (int i = 0; i < n; ++i) {
19              scanf("%d", a+i);
20              sum += a[i];
21          }
22          int cnt = 0;
23          for (int i = 0; i < n; ++i) {
24              root[++cnt] = NewTree(a[i]);
25              num[cnt] = 1;
26              while (cnt > 1 && Top(root[cnt]) < Top(root[cnt-1])) {
27                  cnt--;
28                  root[cnt] = Merge(root[cnt], root[cnt+1]);
29                  num[cnt] += num[cnt+1];
30                  while (tr[root[cnt]].sz*2 > num[cnt]+1) Pop(root[cnt]);
31              }
32          }
33          int px = 0;
34          for (int i = 1; i <= cnt; ++i)
35              for (int j = 0, x = Top(root[i]); j < num[i]; ++j)
36                  tmp += abs(a[px++]-x);
37          ans = tmp;
38
39          printf("%lld\n", ans);
40      }
41      return 0;
42  }
```

1.4 平衡二叉树

平衡二叉树（Balanced Binary Tree）是对二叉查找树的一种改进，又被称为 AVL 树。二叉查找树的查找复杂度与高度有关，但是在一些情况下，二叉查找树会退化成线性，导致复杂度变高。平衡二叉树能很好地维持二叉查找树的平衡，将复杂度维持在 $O(\log_2 N)$。平衡二

叉树具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。

AVL 树编程复杂度较高。本节将讲解两种平衡树：Treap 和 Splay 树。它们不严格平衡，但是实现相对简单。

1.4.1　Treap

由二叉查找树和堆合并构成的新数据结构被称为 Treap。它的名字取了 Tree 和 Heap 各一半，又称为树堆。Treap 巧妙地通过随机数实现了平衡。

1．Treap 的数据结构

Treap 每个节点的数据域包含 2 个值，即 key 和 weight。其中，key 值和原来的二叉查找树一样，满足左子树<根节点<右子树；weight 值是随机产生的，在 Treap 中，weight 值满足堆的性质，根节点的 weight 值不大于（或不小于）左右子节点。Treap 的数据结构如图 1.4 所示。

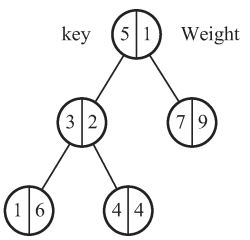


图 1.4　Treap 的数据结构

Treap 节点的数据结构如下所述。

```
1    struct Node {
2        int size;              //以该节点为根的子树大小
3        int key;               //节点的值
4        int weight;            //随机生成，满足堆性质
5        Node *left;            //左子节点
6        Node *right;           //右子节点
7
8        Node(int key): key(key) {
9            size = 1;
10           weight = rand();
11           left = right = NULL;
12       }
13   };
```

2．Treap 的操作

简单来说，为了防止二叉查找树退化成一条链，Treap 为每一个节点赋予一个随机值，然后对这个随机值按照堆的性质去调整。所以，Treap 的大部分操作和二叉查找树相同，不过在每一次操作后需要做出调整。这个调整的过程被称为旋转，每次旋转在不改变 key 值顺序的情况下，可使 weight 值满足堆性质。旋转分为左旋和右旋。将右子节点旋转至根，所以称为左旋，反之将左子节点旋转至根，称为右旋，如图 1.5 和图 1.6 所示。

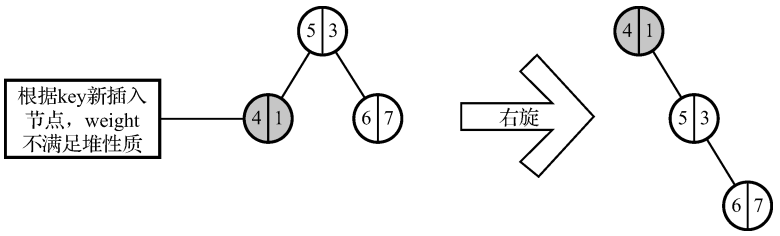


图 1.5　Treap 右旋（一）

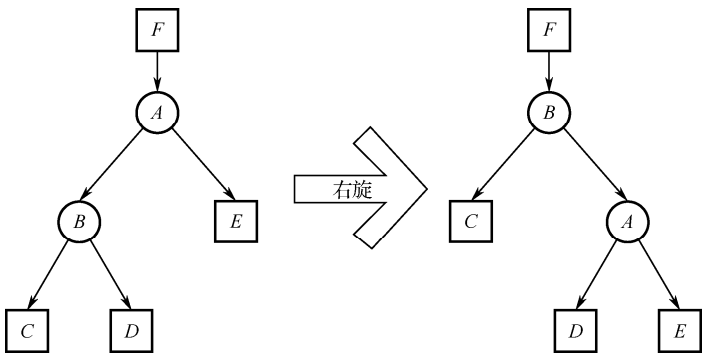


图 1.6　Treap 右旋（二）

左旋是右旋的镜像操作。

旋转操作的实现如下所述。

```
int get_size(Node *node) {
    return node == NULL ? 0 : node->size;
}

void left_rotate(Node *&a) {                //左旋，把节点 A 的右子节点 B 转到 A 的父节点
    Node *b = a->right;
    a->right = b->left;
    b->left = a;
    b->size = a->size;
    a->size = get_size(a->left) + get_size(a->right) + 1;
    a = b;
}

void right_rotate(Node *&a) {               //右旋把节点 A 的左子节点 B 转到 A 的父节点
    Node *b = a->left;
    a->left = b->right;
```

```
17          b->right = a;
18          b->size = a->size;
19          a->size = get_size(a->left) + get_size(a->right) + 1;
20          a = b;
21      }
```

1.4.2　Splay 树

Splay 树又称为伸展树，也称为自适应查找树，是一种用于保存有序集合的、简单高效的数据结构。伸展树实质上是一个二叉查找树，允许查找、插入、删除、删除最小、删除最大、分割、合并等许多操作。这些操作的时间复杂度为 $O(\log_2 N)$。

1．Splay 树的原理

伸展树的出发点是这样的：考虑到局部性原理（刚被访问的内容下次可能仍会被访问，查找次数多的内容可能下一次会被访问），为了使整个查找时间更少，被查频率高的那些节点应当经常处于靠近树根的位置。因此，可得到以下方案：每次查找节点之后对树进行重构，把被查找的节点移到树根，这种自调整形式的二叉查找树就是伸展树。每次对 Splay 树进行操作后，它均会通过旋转的方式把被访问节点旋转到树根的位置。

2．Splay 树的基本操作

Splay 树的操作是在保持 Splay 树有序性的前提下，通过一系列旋转操作将树中的元素 X 调整至树根部的操作，Zig 表示右旋，Zag 表示左旋。

Splay 树的基本操作包括三种情况：

（1）Zig 或 Zag：当目标节点是根节点的左子节点或右子节点时，进行一次单旋转，将目标节点调整到根节点的位置。Splay 树的 Zig 操作如图 1.7 所示。

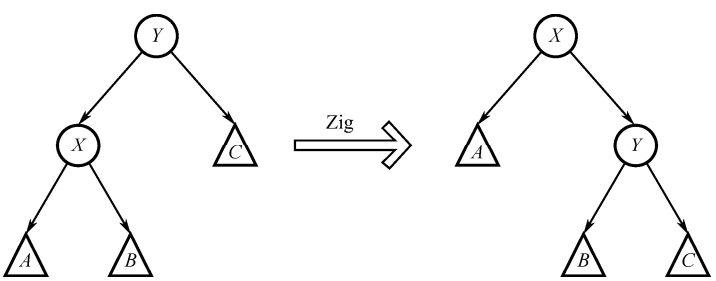


图 1.7　Splay 树的 Zig 操作

（2）Zig-Zig 或 Zag-Zag：目标节点 X 的父节点 Y 的父节点是根节点，其中 X 和 Y 都是其父节点的左子节点或者都是其父节点的右子节点。Splay 树的 Zig-Zig 操作如图 1.8 所示。

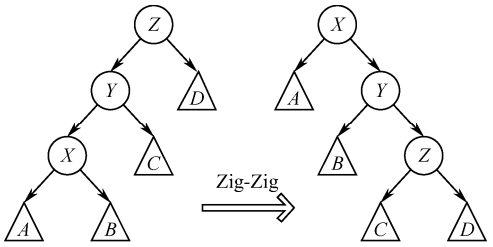


图 1.8　Splay 树的 Zig-Zig 操作

（3）Zig-Zag 或 Zag-Zig：目标节点 *X* 的父节点 *Y* 的父节点是根节点，其中 *X* 和 *Y*，一个是其父节点的左子节点，另一个是其父节点的右子节点。Splay 树的 Zig-Zag 操作如图 1.9 所示。

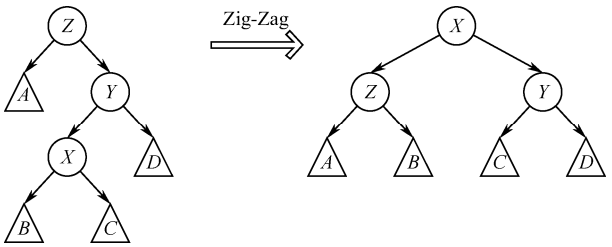


图 1.9　Splay 树的 Zig-Zag 操作

例如，将节点 1 旋转到根节点的操作示例如图 1.10 所示。

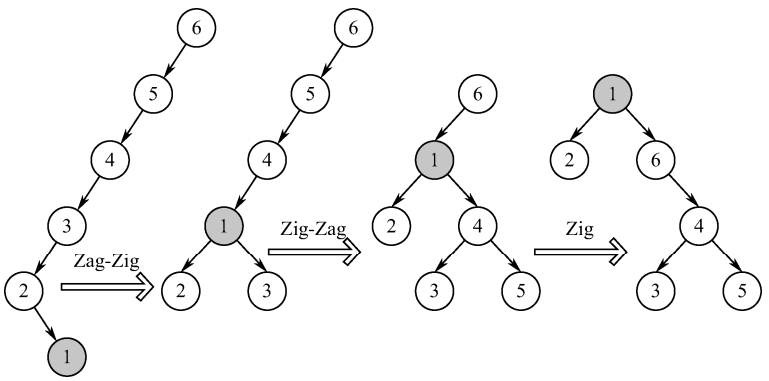


图 1.10　Splay 树的操作示例

Splay 树代码实现如下所述。

```
1    struct splay_tree {
2        unsigned long p_size;                    //记录 Splay 树中一共有多少个节点
3
```

```
4       struct node {
5           node *left, *right;                 //左子节点和右子节点
6           node *parent;                       //父节点
7           int key;
8           node(int key) : left(NULL), right(NULL), parent(NULL), key(key) { }
9           ~node() {
10              if(left) delete left;
11              if(right) delete right;
12              if(parent) delete parent;
13          }
14      } *root;
15      void left_rotate(node *x) {             // Zag
16          node *y = x->right;
17          if(y) {
18              x->right = y->left;
19              if(y->left) y->left->parent = x;
20              y->parent = x->parent;
21          }
22          if(!x->parent) root = y;
23          else if(x == x->parent->left) x->parent->left = y;
24          else x->parent->right = y;
25          if(y) y->left = x;
26          x->parent = y;
27      }
28      void right_rotate(node *x) {            // Zig
29          node *y = x->left;
30          if(y) {
31              x->left = y->right;
32              if(y->right) y->right->parent = x;
33              y->parent = x->parent;
34          }
35          if(!x->parent) root = y;
36          else if(x == x->parent->left) x->parent->left = y;
37          else x->parent->right = y;
38          if(y) y->right = x;
39          x->parent = y;
40      }
41      void splay(node *x) {                   // Splay 树操作，将节点 X 调整到根节点
42          while(x->parent) {
43              if(!x->parent->parent) {
44                  if(x->parent->left == x) right_rotate(x->parent);
```

```
45                      else left_rotate(x->parent);
46                  } else if(x->parent->left == x && x->parent->parent->left == x->parent) {
47                      right_rotate(x->parent->parent);
48                      right_rotate(x->parent);
49                  } else if(x->parent->right == x && x->parent->parent->right == x->parent) {
50                      left_rotate(x->parent->parent);
51                      left_rotate(x->parent);
52                  } else if(x->parent->left == x && x->parent->parent->right == x->parent) {
53                      right_rotate(x->parent);
54                      left_rotate(x->parent);
55                  } else {
56                      left_rotate(x->parent);
57                      right_rotate(x->parent);
58                  }
59              }
60          }
61          void replace(node *u, node *v) {
62              if(!u->parent) root = v;
63              else if(u == u->parent->left) u->parent->left = v;
64              else u->parent->right = v;
65              if(v) v->parent = u->parent;
66          }
67          node* subtree_minimum(node *u) {       //寻找子树中最小的值
68              while(u->left) u = u->left;
69              return u;
70          }
71          node* subtree_maximum(node *u) {       //寻找子树中最大的值
72              while(u->right) u = u->right;
73              return u;
74          }
75
76          splay_tree() : root(NULL), p_size(0) { }
77
78          void insert(int key) {                 //插入节点
79              node *z = root;
80              node *p = NULL;
81              while(z) {
82                  p = z;
83                  if(z->key < key) z = z->right;
84                  else z = z->left;
85              }
```

```
86              z = new node(key);
87              z->parent = p;
88              if(!p) root = z;
89              else if(p->key < z->key) p->right = z;
90              else p->left = z;
91              splay(z);
92              p_size++;
93          }
94          node* find(int key) {                       //查询
95              node *z = root;
96              while(z) {
97                  if(z->key < key) z = z->right;
98                  else if(key < z->key) z = z->left;
99                  else return z;
100
101             }
102             return NULL;
103         }
104         void erase(int key) {                       //删除值为 key 的节点
105             node *z = find(key);
106             if(!z) return;
107             splay(z);
108             if(!z->left) replace(z, z->right);
109             else if(!z->right) replace(z, z->left);
110             else {
111                 node *y = subtree_minimum(z->right);
112                if(y->parent != z) {
113                     replace(y, y->right);
114                     y->right = z->right;
115                     y->right->parent = y;
116                 }
117                 replace(z, y);
118                 y->left = z->left;
119                 y->left->parent = y;
120             }
121             delete z;
122             p_size--;
123         }
```

1.4.3 例题讲解

例 1-3 Black Box

Time Limit: 1000 ms　　Memory Limit: 10000 KB

题目描述：

ADD 和 GET 操作。ADD *x* 操作表示向序列中添加一个数 *x*，第 *i* 次 GET 表示获取序列中第 *i* 小的数。ADD 和 GET 操作都不超过 30000 次。*A*(1),*A*(2)…,*A*(*M*)表示依次向序列中添加的数，*A* 的值不超过 2000000000。序列 *u*(1),*u*(2),…,*u*(*N*)是非递减序列，*N*≤*M* 且对于每一个 *p*(1≤*p*≤*N*)，*p*≤*u*(*p*)≤*M*，*u*(*p*)表示分别在进行多少次 ADD 操作后再进行 GET 操作。例如，*u*(*p*)表示插入 *u*(*p*)个数后输出第 *p* 小的数。

输入：

第一行输入两个整数 *M* 和 *N*，第二行包含 *M* 个数表示 *A*(1),*A*(2),…,*A*(*M*)，第三行包含 *N* 个整数表示 *u*(1),*u*(2),…,*u*(*N*)。

输出：

对于每一个 GET 输出结果，每行输出一个数。

样例输入：

7 4

3 1 -4 2 8 -1000 2

1 2 6 6

样例输出：

3

3

1

2

题目来源：POJ 1442。

解题思路：

给定一个序列包含两个操作：一个是向序列中插入一个数；另一个是查询序列中第 *k* 小的值。这两个操作都可以由 Treap 实现。

题目实现（节点定义和旋转操作同上，略）：

```
1    void insert(Node *&node, int key) {                    // 在 node 中插入值 key
2        //按照普通二叉查找树的性质插入 key
3        //插入后通过旋转维持 weight 的小顶堆性质
4        if (!node) {
5            node = new Node(key);
6        } else if (key <= node->key) {
```

```
7                 node->size++;
8                 insert(node->left, key);
9                 if (node->left->weight < node->weight) {
10                    right_rotate(node);
11                }
12            } else {
13                node->size++;
14                insert(node->right, key);
15                if (node->right->weight < node->weight) {
16                    left_rotate(node);
17                }
18            }
19    }
20
21    int find_kth(Node *node, int k) {
22        int left_size = get_size(node->left);
23        if (k == left_size + 1) return node->key;
24        else if (k <= left_size) return find_kth(node->left, k);
25        else return find_kth(node->right, k - 1 - left_size);
26    }
27
28    int A[30000];
29    int main() {
30        Node *root = NULL;
31        int m, n;
32        int now = 1;                                    //表示 ADD 操作进行到第几次
33        scanf("%d%d", &m, &n);
34        for (int i = 1; i <= m; ++i)
35            scanf("%d", &A[i]);
36        int u;
37        for (int i = 1; i <= n; ++i) {
38            scanf("%d", &u);
39            for (; now <= u; ++now)
40                insert(root, A[now]);
41            printf("%d\n", find_kth(root, i));
42        }
43        return 0;
44    }
```

例 1-4　营业额统计

Time Limit: 5000 ms　　Memory Limit: 162 MB

题目描述：

Tiger 最近被公司升任为营业部经理。他上任后接受公司交给的第一项任务便是统计并分析公司成立以来的营业情况。Tiger 拿出了公司的账本，账本上记录了公司成立以来每天的营业额。分析营业情况是一项相当复杂的工作。由于在节假日、大减价或者其他情况的时候，营业额会出现一定的波动，当然一定的波动是能够接受的，但是在某些时候营业额突变得很高或者很低，证明公司此时的经营状况出现了问题。经济管理学上定义了一种最小波动值来衡量这种情况：该天的最小波动值越大时，说明营业情况越不稳定。而分析整个公司从成立到现在营业情况是否稳定，只需要把每一天的最小波动值加起来就可以了。你的任务就是编写一个程序帮助 Tiger 来计算这一个值。第一天的最小波动值为第一天的营业额。

输入：

第一行为正整数，表示该公司从成立一直到现在的天数，在接下来的 n 行中的每行都有一个整数（有可能有负数），表示第 i 天公司的营业额。

输出：

在输出文件中仅有一个正整数，即 Sigma（每天最小的波动值）。结果小于 2^{31}。

样例输入：

6
5
1
2
5
4
6

样例输出：

12

提示：

结果说明：5+|1−5|+|2−1|+|5−5|+|4−5|+|6−5|=5+4+1+0+1+1=12

题目来源：HYSBZ 1588。

解题思路：

Splay 入门题，考察对序列的基本操作。

题目实现：

```
1    #include <cstdio>
2    #include <algorithm>
3    using namespace std;
```

```
4   const int inf = 0x7fffffff;
5
6   struct splay_tree {
7       // …同上
8       int find_prev(int key) {                    //查找小于等于 key 的最大值
9           int ans = -inf;
10          node *x = root;
11          while (x) {
12              if (x->key == key) return x->key;
13              if (x->key < key) {
14                  ans = max(ans, x->key);
15                  x = x->right;
16              } else {
17                  x = x->left;
18              }
19          }
20          return ans;
21      }
22      int find_next(int key) {                    //查找大于或等于 key 的最小值
23          int ans = inf;
24          node *x = root;
25          while (x) {
26              if (x->key == key) return x->key;
27              if (x->key > key) {
28                  ans = min(ans, x->key);
29                  x = x->left;
30              } else {
31                  x = x->right;
32              }
33          }
34          return ans;
35      }
36  };
37
38  int main() {
39      splay_tree spt;
40      int n, sale;
41      int ans = 0;
42
43      scanf("%d", &n);
44      scanf("%d", &sale);
```

```
45          ans = sale;
46          spt.insert(sale);
47          for (int i = 1; i < n; ++i) {
48              scanf("%d", &sale);
49              int min_fluc = inf;
50              int prev = spt.find_prev(sale);
51              if (prev != -inf) min_fluc = sale - prev;
52              int next = spt.find_next(sale);
53              if (next != inf) min_fluc = min(min_fluc, next - sale);
54              ans += min_fluc;
55              spt.insert(sale);
56          }
57          printf("%d\n", ans);
58      }
```

1.5 练习题

习题 1-1

题目来源：POJ 2388。

题目类型：排序算法。

解题思路：为一个序列求中位数，排序后取中间的数即可，通过堆排序可解。

习题 1-2

题目来源：HDOJ 2852。

题目类型：树状数组，二分法。

解题思路：给定一个容器，里面存放各种数值，规定三个操作：一个是在容器中增加一个数值；一个是在容器中删掉一个数值；一个是询问容器中比 a 大的数中的第 k 个数，并将其输出。树状数组可以实现点更新，对于询问容器中比 a 大的第 k 个数，可以通过二分法答案求得。

习题 1-3

题目来源：POJ 3016。

题目类型：左倾堆，动态规划。

解题思路：给定 n 个数的序列 A，将其分成 k 个区间，改变其中的一些数使得每个区间严格单调，求改动的最小代价。此题和前面的例题相比，要求严格递增，所以首先需要计算

$A[i]-i$。将$[i,j]$区间调整成单调序列的代价表示为 cost$[i][j]$，显然将前 i 部分分成 k 块单调区间所需要的最小代价就是 $dp[k][i] = \min\{dp[k-1][j] + cost[j+1][i], (j<i)\}$，需要求 cost 数组，这样就转化为前面的例题。在 Treap 维护中位数合并时，如果当前块的数量为偶数，答案不变，否则答案增加。

习题 1-4

题目来源：HDOJ 4557。

题目类型：Treap。

解题思路：人才库存储一些人员信息，每人有一个能力值，公司在招聘时有一个能力值的最低要求，优先把能力值低的人才推荐过去；如果依然有多名人员符合要求，就把其中最早来求职的那位学生推荐过去。用 Treap 保存人员信息，每个节点有能力值和时间两个人信息，排序时需要考虑两个信息。姓名可以用 map 映射到时间。

习题 1-5

题目来源：POJ 3580。

题目类型：Splay 树。

解题思路：题目要求实现一种数据结构，支持 6 种操作：①add x,y D：第 x 个数到第 y 个数之间的数每个加 D；②reverse x y：第 x 个数到第 y 个数之间的数全部翻转；③revolve x y T：第 x 个数到第 y 个数之间的数向后循环流动 T 次，即后面 T 个数变成子序列的最前面 T 个，前面的被挤到后面；④insert x P：在第 x 个数后面插入一个数 P；⑤delete x：删除第 x 个数；⑥min x，y：求第 x 个数到第 y 个数之间的最小数字。很明显的数据结构题，只有强大的 Splay 树才能实现这么多功能。本习题涉及区间更新及单点插入、删除等操作，能全面地了解 Splay 树的功能。

第 2 章

字　符　串

在目前的各种算法竞赛中，字符串都是出题的热门，也是比赛的难点。相关字符串方面的算法有很多，在此仅介绍几种在比赛中常用的几种算法。本章首先介绍 Trie 树、KMP 基本字符串算法，然后介绍 AC 自动机、后缀数组、后缀自动机等高级算法。本章在介绍字符串相关算法的同时，给出相应的算法实现与复杂度分析。

2.1　Trie 树

Trie 树，即字典树，又称单词查找树或键树，是一种树形结构，是哈希树的一种变种。Trie 树的典型应用是排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是可最大限度地减少无谓的字符串比较，查询效率比哈希表高。

Trie 树的核心思想是空间换时间，即利用字符串的公共前缀来降低查询时间的开销，以达到提高效率的目的。

2.1.1　Trie 树的原理

假设有 abc、abd、bcd、abcd、efg、hii 六个单词，构建的 Trie 树如图 2.1 所示。

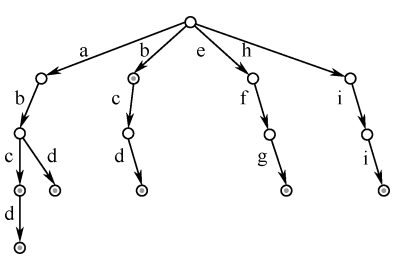


图 2.1　Trie 树

对于每一个节点，从根遍历到它的过程就是一个单词，如果这个节点被标记为红色（图 2.1 中有阴影的小圆圈），就表示这个单词存在，否则不存在。那么，对于一个单词，只要顺着根走到对应的节点，再检查这个节点是否被标记为红色，就可以知道它是否出现过了。把这个节点标记为红色，就相当于插入了这个单词。

查询和插入可以一起完成（重点体会这个查询和插入是如何一起完成的，下文将具体解释），所用时间仅仅为单词长度。Trie 树每一层的节点数是 26^i 级别的，为了节省空间，用动态链表或者用数组来模拟，空间的花费不会超过单词数×单词长度。

Trie 树有 3 个基本性质：

- 根节点不包含字符，除根节点外的每一个节点都只包含一个字符。
- 从根节点到某一节点，将路径上经过的字符连接起来，即可得到该节点对应的字符串。
- 每个节点的所有子节点包含的字符都不相同。

Trie 树是一个很重要的数据结构，主要应用为：

（1）词频统计：如给定一个由 10 万个单词组成的库，现要判断一个单词是否在库中出现，若出现，求出共出现多少次。

（2）前缀匹配：以图 2.1 为例，如果想获取所有以“a”开头的字符串，那么从图中可以很明显地看到 abc、abd、abcd。利用这个特性，可以巧妙地实现搜索提示功能，如输入一个网址，可以自动列出可能的选择。当没有完全匹配的搜索结果，可以列出前缀最相似的可能。

2.1.2　Trie 树的实现

将根节点编号为 0，然后把其余节点编号为从 1 开始的正整数，用一个数组来保存每个节点的所有子节点，用下标直接存取。具体来说，可以用 ch[*i*][*j*]保存节点 *i* 的那个编号为 *j* 的子节点。将字符串中的小写字母按照字典序编号为 0、1、2…，则 ch[*i*][0]表示节点 *i* 的子节点 *a*。ch[*i*][*j*]=1 表示该子节点存在，否则表示不存在。用 sigma_size 表示字符集的大小，如当字符集为全部小写字母时，sigma_size=26。

使用 Trie 树的时候，往往需要在单词节点上加附加信息，其中 val[*i*]表示节点 *i* 的附近信息。例如，如果每个字符串有个权值，可以把权值存于 val[*i*]中。

```
1    const int maxnode = 4000 * 100 + 10;
2    const int sigma_size = 26;
3    //字母表为全体小写字母的 Trie 树
4    struct Trie {
5        int ch[maxnode][sigma_size];
6        int val[maxnode];
7        int sz;
8    }                                                    //节点总数
```

```
9   void clear() { sz = 1; memset(ch[0], 0, sizeof(ch[0])); }    //初始时只有一个根节点
10  int idx(char c) { return c - 'a'; }                          //字符 c 的编号
11
12  //插入字符串 s，附加信息为 v。注意 v 必须非 0，因为 0 代表本节点不是单词节点
13  void insert(const char *s, int v) {
14      int u = 0, n = strlen(s);
15      for(int i = 0; i < n; i++) {
16          int c = idx(s[i]);
17          if(!ch[u][c]) {                                      //节点不存在
18              memset(ch[sz], 0, sizeof(ch[sz]));
19              val[sz] = 0;                                     //中间节点的附加信息为 0
20              ch[u][c] = sz++;                                 //新建节点
21          }
22          u = ch[u][c];                                        //往下走
23      }
24      val[u] = v;                                              //字符串的最后一个字符的附加信息为 v
25  }
26
27  //找字符串 s 的长度不超过 len 的前缀
28  bool find(const char *s, int len) {
29      int u = 0;
30      for(int i = 0; i < len; i++) {
31          if(s[i] == '\0') break;
32          int c = idx(s[i]);
33          if(!ch[u][c]) break;
34          u = ch[u][c];
35          if(val[u] != 0) return true;                         //找到一个前缀
36      }
37      return false;
38  }
```

2.1.3 例题讲解

例 2-1 Xor Sum

Time Limit: 2000/1000 ms (Java/Others) Memory Limit: 132768/132768 KB(Java/Others)

题目描述：

Zeus 和 Prometheus 做了一个游戏，Prometheus 给 Zeus 一个集合，在集合中包含 N 个正整数，随后 Prometheus 将向 Zeus 发起 M 次询问，在每次询问中包含一个正整数 S，之后 Zeus 需要在集合中找出一个正整数 K，使得 K 与 S 的异或结果最大。

输入：

输入包含若干组测试数据，每组测试数据包含若干行。

输入的第一行是一个整数 T（$T < 10$），表示共有 T 组数据。

每组数据的第一行输入两个正整数 N、M（$1 \leqslant N$、$M \leqslant 100000$），在接下来的一行中包含 N 个正整数，代表 Zeus 获得的集合；在之后的 M 行中，每行一个正整数 S，代表 Prometheus 询问的正整数。所有正整数均不超过 2^{32}。

输出：

对于每组数据，首先需要输出单独一行“Case #?:”，其中问号处应填入当前的数据组数，组数从 1 开始计算。

对于每个询问，输出一个正整数 K，使得 K 与 S 异或值最大。

样例输入：

```
2

3 2
3 4 5
1
5

4 1
4 6 5 6
3
```

样例输出：

```
Case #1:
4
3
Case #2:
4
```

题目来源：HDU4825。

解题思路：

把每一个数都转换成二进制（位置要对齐，前面不足则补 0），转换的长度和题目给的数据范围有关，对于该题，33 位就够了。更新 Trie 树的顺序和过程是从左到右地遍历插入各位二进制数值，但显然 next 指针只有两个——next[0]与 next[1]，更新的最后是把原来的值附到结尾节点的 val 上，表示这整条路对应的十进制数值是 val。

如何求最大异或值呢？首先要知道在一般情况下，正项等比序列前 $n-1$ 项求和的值肯定

要小于第 n 项的值，也就是说，在最坏情况下走第 x 个节点，但是 x+1 到 n 个节点都是 0，也比不走 x，但是 x+1 到 n 个节点都是 1 的情况好。

假设已经得到了最大异或值 Max，那么 Max 异或 K 就可以得到某个数 X_i 了，此时不知道 X_i 是什么，可以通过贪心算法找到它。

题目实现：

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define maxn 100000+5
using namespace std;
typedef long long LL;
int a[maxn],n,m;
int ch[32*maxn][2];
LL val[32*maxn];
int node_cnt;
void inti(){
    node_cnt=1;
    memset(ch[0],0,sizeof(ch[0]));
}
void Insert(LL x){
    int cur=0;
    for(int i=32;i>=0;i--)
    {
        int idx=(x>>i)&1;
        if(!ch[cur][idx])
        {
            memset(ch[node_cnt], 0, sizeof(ch[node_cnt]));
            ch[cur][idx]=node_cnt;
            val[node_cnt++]=0;
        }
        cur=ch[cur][idx];
    }
    val[cur]=x;
}
LL Query(LL x){
    int cur=0;
    for(int i=32;i>=0;i--)
    {
        int idx=(x>>i)&1;
        if(ch[cur][idx^1])cur=ch[cur][idx^1];
```

```
36              else cur=ch[cur][idx];
37          }
38          return val[cur];
39      }
40      int main(){
41          int T,t=1,s;
42          scanf("%d",&T);
43          while(t<=T)
44          {
45              inti();
46              scanf("%d%d",&n,&m);
47              for(int i=1;i<=n;i++)
48              {
49                  scanf("%d",&a[i]);
50                  Insert(a[i]);
51              }
52              printf("Case #%d:\n",t++);
53              while(m--)
54              {
55                  scanf("%d",&s);
56                  printf("%lld\n",Query(s));
57              }
58          }
59          return 0;
60      }
```

2.2　KMP 算法

KMP 算法是由 Knuth、Morris、Pratt 共同提出的模式匹配算法。对于任何模式和目标序列，KMP 算法都可以在线性时间内完成匹配查找，而不会发生退化，是一个非常优秀的模式匹配算法。但是相较于其他模式匹配算法，该算法晦涩难懂，第一次接触该算法的读者往往会感觉一头雾水，主要原因是 KMP 算法在构造跳转表 next 的过程中进行了多个层面的优化和抽象，使得 KMP 算法进行模式匹配的原理显得不是那么直白。本节将深入 KMP 算法，把该算法的各个细节彻底讲透，希望能扫除读者对该算法的困扰。

2.2.1　KMP 算法的原理

KMP 算法完成的任务是：给定两个字符串 O 和 f，其长度分别为 n 和 m，判断 f 是否在 O 中出现，如果出现，则返回到出现的位置。常规方法是遍历 a 的每一个位置，然后从该位

置开始和 *b* 进行匹配，但是这种方法的复杂度是 *O*(*nm*)。KMP 算法通过一个 *O*(*m*)的预处理，使匹配的复杂度降为 *O*(*n*+*m*)。

这里首先用一个图来描述 KMP 算法的思想，其原理如图 2.2 所示。在字符串 *O* 中寻找 *f*，当匹配到位置 *i* 时，两个字符串不相等，这时需要将字符串 *f* 向前移动。常规方法是每次向前移动一位，但是它没有考虑前 *i*–1 位已经比较过这个事实，所以效率不高。事实上，如果我们提前计算某些信息，就有可能一次前移多位。假设根据已经获得的信息知道可以前移 *k* 位，分析移位前后的 *f* 有什么特点，则可以得到如下的结论：

（1）*A* 段字符串是 *f* 的一个前缀。

（2）*B* 段字符串是 *f* 的一个后缀。

（3）*A* 段字符串和 *B* 段字符串相等。

前移 *k* 位之后，可以继续比较位置 *i* 的前提是 *f* 的前 *i*–1 个位置满足：长度为 *i*–*k*–1 的前缀 *A* 和后缀 *B* 相同。只有这样，才可以在前移 *k* 位后从新的位置继续比较。

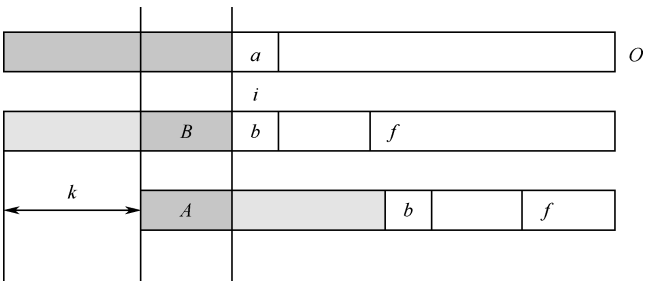


图 2.2　KMP 算法的原理

KMP 算法的核心是计算字符串 *f* 的每一个位置之前的字符串的前缀和后缀公共部分的最大长度（不包括字符串本身，否则最大长度始终是字符串本身）。获得 *f* 的每一个位置的最大公共长度之后，就可以利用该最大公共长度快速地和字符串 *O* 进行比较。当每次比较到两个字符串的字符不同时，就可以根据最大公共长度将字符串 *f* 向前移动“已匹配长度–最大公共长度”位，接着继续比较下一个位置。事实上，字符串 *f* 的前移只是概念上的前移，只要在比较的时候在最大公共长度之后比较 *f* 和 *O* 即可达到字符串 *f* 前移的目的。

next 数组计算：理解了 KMP 算法的基本原理，下一步就是要获得字符串 *f* 的每一个位置的最大公共长度。这个最大公共长度在算法导论里面被记为 next 数组。在这里要注意一点，next 数组表示的是长度，下标从 1 开始；但是在遍历原字符串时，下标还是从 0 开始的。假设现在已经求得 next[1]、next[2]、…、next[*i*]，分别表示长度为 1 到 *i* 的字符串的前缀和后缀最大公共长度，现在要求 next[*i*+1]。由图 2.2 可以看到，如果位置 *i* 和位置 next[*i*]处的两个字符相同（下标从 0 开始），则 next[*i*+1]等于 next[*i*]加 1。如果两个位置的字符不相同，则可以将长度为 next[*i*]的字符串继续分割，获得其最大公共长度 next[next[*i*]]，然后

和位置 *i* 的字符比较。这是因为长度为 next[*i*]的前缀和后缀都可以分割成相同的子串，如果位置 next[next[*i*]]和位置 *i* 的字符相同，则 next[*i*+1]就等于 next[next[*i*]]加 1。如果不相等，就可以继续分割长度为 next[next[*i*]]的字符串，直到字符串的长度为 0 为止。next 数组计算原理如图 2.3 所示。

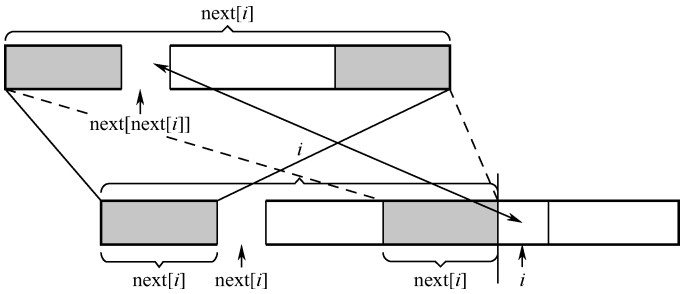


图 2.3　next 数组计算原理

2.2.2　KMP 算法的实现

```
1    void cal_next( char * str, int * next, int len )
2    {
3        int i, j;
4
5        next[0] = -1;
6        for( i = 1; i < len; i++ )
7        {
8            j = next[ i - 1 ];
9            while( str[ j + 1 ] != str[ i ] && ( j >= 0 ) )
10           {
11               j = next[ j ];
12           }
13           if( str[ i ] == str[ j + 1 ] )
14           {
15               next[ i ] = j + 1;
16           }
17           else
18           {
19               next[ i ] = -1;
20           }
21       }
22   }
23
```

```
24  int KMP( char * str, int slen, char * ptr, int plen, int * next )
25  {
26      int s_i = 0, p_i = 0;
27
28      while( s_i < slen && p_i < plen )
29      {
30          if( str[ s_i ] == ptr[ p_i ] )
31          {
32              s_i++;
33              p_i++;
34          }
35          else
36          {
37              if( p_i == 0 )
38              {
39                  s_i++;
40              }
41              else
42              {
43                  p_i = next[ p_i - 1 ] + 1;
44              }
45          }
46      }
47      return ( p_i == plen ) ? ( s_i - plen ) : -1;
48  }
```

和朴素（Brute Force）算法相比，KMP 算法的时间效率就很高了。KMP 算法对模板进行预处理的时间复杂度为 $O(m)$，字符串匹配的时间复杂度为 $O(n)$，这样的复杂度已经是最优了，因为需要检查文本串和模板的每个字符，KMP 算法的时间复杂度为 $O(m+n)$。

2.2.3 例题讲解

例 2-2 剪花布条

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

一条花布条，里面有些图案，另有一条直接可用的小饰条，里面也有一些图案。对于给定的花布条和小饰条，计算一下能从花布条中尽可能剪出多少条小饰条。

输入：

输入中含有的数据分别是成对出现的花布条和小饰条，二者都是用可见 ASCII 字符表示

的，可见 ASCII 字符有多少个，花布上就有多少种花纹。花布条和小饰条不会超过 1000 个字符长，如果遇见“#”字符，则不再进行工作。

输出：

输出能从花布条上剪出的最多小饰条的条数，如果一个都没有，那就输出 0，在每个结果之间应换行。

样例输入：

abcde a3

aaaaaa　aa

#

样例输出：

0

3

题目来源：HDU 2087。

解题思路：

本题可以通过 KMP 算法解决，题目要求的是给定一个文本串和给定一个模式串，求文本串中有几个模式串。注意在文本串为“aaaaaa”、模式串为“aa”的时候，ans 是 3 而不是 5。

题目实现：

```
#include<algorithm>
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;

#define MAXN 1010

int ans;
char text[MAXN];
char pattern[MAXN];
int next[MAXN];

/*求 next 数组*/
void getNext(){
    int m = strlen(pattern);
    next[0] = next[1] = 0;
    for(int i = 1 ; i < m ; i++){
        int j = next[i];
        while(j && pattern[j] != pattern[i])
```

```
21              j = next[j];
22              next[i+1] = pattern[i] == pattern[j] ? j+1 : 0;
23          }
24      }
25
26      /*匹配*/
27      void find(){
28          ans = 0;
29          int m = strlen(pattern);
30          int n = strlen(text);
31          int j = 0;                              /*模式串的下标*/
32          for(int i = 0 ; i < n ; i++){
33              while(j && pattern[j] != text[i])
34              j = next[j];
35              if(pattern[j] == text[i])
36              j++;
37              if(j == m){
38                  ans++;
39                  j = 0;                          /*这个地方注意防止出现输入 aaaaaa aa 后，输出 5 的情况*/
40              }
41          }
42          printf("%d\n" , ans);
43      }
44
45
46      int main(){
47          while(1){
48              scanf("%s" , text);
49              if(!strcmp(text , "#"))
50              break;
51              scanf("%s" , pattern);
52              getNext();
53              find();
54          }
55          return 0;
56      }
```

2.3　Aho-Corasick 自动机

Aho-Corasick 自动机（AC 自动机）算法在 1975 年产生于贝尔实验室，是著名的多模匹配算法之一。一个常见的例子就是给出 n 个单词，再给出一段包含 m 个字符的文章，找出有多少个单词在文章里出现过。要理解 AC 自动机，先得有 Trie 树（字典树）和 KMP 算法的基础知识。AC 自动机算法分为 3 步：构造一棵 Trie 树，构造失败指针（fail 指针），以及模式匹配过程。

2.3.1　Aho-Corasick 自动机原理

以典型应用为例，现给定 3 个单词“china”“hit”“use”，再给定一段文本“chitchat”，求有多少个单词出现在文本中。

（1）根据单词集合{china,hit,use}建立一棵 Trie 树，如图 2.4 所示。

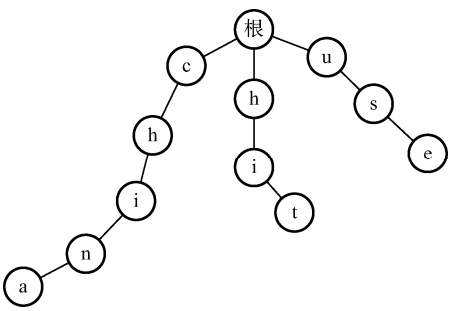


图 2.4　建立 Trie 树

（2）根据所给文本“chitchat”依次匹配，图中所示“chi”为匹配成功的字符串，如图 2.5 所示。

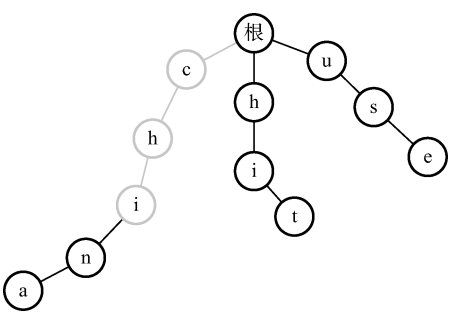


图 2.5　匹配成功的字符串

（3）当匹配到第四个字符时，“t”和“n”匹配失败，如图 2.6 所示。

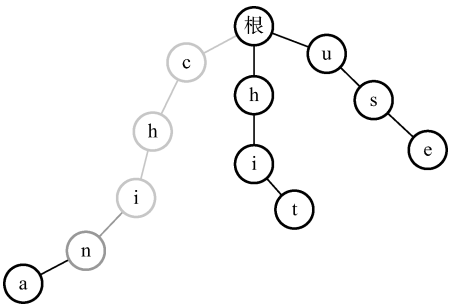


图 2.6　匹配失败

（4）此时知道已匹配成功的字符串为“chi”。

AC 算法的核心就是**在所有给定的单词中，找到这样的一个单词，使其与已匹配成功字符串的相同前后缀最长，利用这个最长的相同前后缀实现搜索跳转。**例如，单词“hit”与已匹配成功字符串“chi”的最长相同前后缀为“hi”，因此下一步从单词“hit”的“t”开始搜索，如图 2.7 所示。

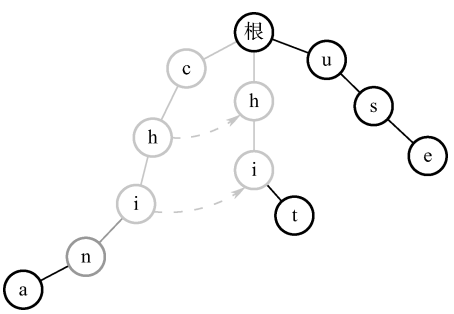


图 2.7　搜索跳转

（5）此时“t”是匹配的，在文本“chitchat”中找到一个单词“hit”。

其实到这里，AC 算法的思想已经基本呈现在读者面前了。剩下的问题就是如何解决所述的“核心”。

AC 算法的关键：在每个节点里设置一个指针（称之为 fail 指针），指向跳转的位置。

对于跳转位置的选择，基于以下两点：

（1）对于根节点的所有子节点，它们的 fail 指针都指向根节点；

（2）而对于其他节点，不妨设该节点上的字符为“ch”，沿着它的父节点的 fail 指针走，直到走到一个节点，它的子节点中也有字符为“ch”的节点，然后把该节点的 fail 指针指向那个字符为“ch”的节点。如果一直走到根节点都没找到，则把 fail 指针指向根节点。

指针跳转如图 2.8 所示。

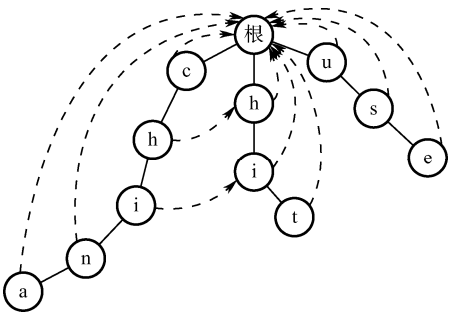


图 2.8　指针跳转

2.3.2　Aho-Corasick 自动机算法的实现

```
#include<cstdio>
#include<cstring>
#include<queue>
using namespace std;
const int maxnode=11000;
const int sigma_size=26;
struct AC_Automata
{
    int ch[maxnode][sigma_size];
    int val[maxnode];       // 每个字符串的结尾节点都有一个非 0 的 val
    int f[maxnode];         // fail 函数
    int last[maxnode];      // last[i]=j，j 节点表示的单词是 i 节点单词的后缀，且 j 节点是单词节点
    int sz;

    //初始化 0 号根节点的相关信息
    void init()
    {
        sz=1;
        memset(ch[0],0,sizeof(ch[0]));
        val[0]=0;
    }

    //insert 函数负责构造 ch 与 val 数组
    //插入字符串，v 必须非 0，表示一个单词节点
    void insert(char *s,int v)
    {
```

```
27          int n=strlen(s),u=0;
28          for(int i=0; i<n; i++)
29          {
30              int id=s[i]-'a';
31              if(ch[u][id]==0)
32              {
33                  ch[u][id]=sz;
34                  memset(ch[sz],0,sizeof(ch[sz]));
35                  val[sz++]=0;
36              }
37              u=ch[u][id];
38          }
39          val[u]=v;
40      }
41
42      //getFail 函数负责构造 f 和 last 数组
43      void getFail()
44      {
45          queue<int> q;
46          last[0]=f[0]=0;
47          for(int i=0; i<sigma_size; i++)
48          {
49              int u=ch[0][i];
50              if(u)
51              {
52                  f[u]=last[u]=0;
53                  q.push(u);
54              }
55          }
56
57          while(!q.empty())              // 按 BFS 顺序计算 fail
58          {
59              int r=q.front(); q.pop();
60              for(int i=0; i<sigma_size; i++)
61              {
62                  int u=ch[r][i];
63                  if(u==0)continue;
64                  q.push(u);
65
66                  int v=f[r];
67                  while(v && ch[v][i]==0) v=f[v];
```

```
                    f[u]= ch[v][i];
                    last[u] =val[f[u]]?f[u]:last[f[u]];
                }
            }
        }

        //递归打印与节点 i 后缀相同的前缀节点编号
        //进入此函数前需要保证 val[i]>0
        void print(int i)
        {
            if(i)
            {
                printf("%d\n",i);
                print(last[i]);
            }
        }

        // 在 s 中找出出现了哪几个模板的单词
        void find(char *s)
        {
            int n=strlen(s),j=0;
            for(int i=0; i<n; i++)
            {
                int id=s[i]-'a';
                while(j && ch[j][id]==0) j=f[j];
                j=ch[j][id];
                if(val[j]) print(j);
                else if(last[j]) print(last[j]);
            }
        }

};
AC_Automata ac;
```

2.3.3　例题讲解

例 2-3　Keywords Search

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 131072/131072 KB(Java/Others)

题目描述：

给定 *n* 个单词以及 1 个字符串，问字符串中出现了多少个单词（如单词“her”“he”为字符串“aher”中出现了两个单词）。

输入：

第一行输入测试数据的组数，然后输入一个整数 n（$n \leqslant 10000$），在接下来的 n 行中每行输入一个单词（每个单词都只包含“a”～“z”，并且单词长度不超过 50），最后输入一个字符串（长度不超过 1000000）。

输出：

对于每组数据，要输出一行，包含一个整数，该整数表示在字符串中出现了多少个单词。

样例输入：

1
5
she
he
say
shr
her
yasherhs

样例输出：

3

题目来源：HDU 2222。

解题思路：

经典的模板题。

```
#include<iostream>
#include<cstdio>
#include<string>
#include<cstring>
#include<vector>
#include<cmath>
#include<queue>
#include<stack>
#include<map>
#include<set>
#include<algorithm>
using namespace std;
const int maxn=1000010;
const int maxm=50*10010;
const int SIGMA_SIZE=26;
int n;
char t[60],s[maxn];
```

```
18
19  struct AC
20  {
21      int ch[maxm][26];
22      int val[maxm];
23      int fail[maxm],last[maxm];
24      int sz;
25      void clear(){memset(ch[0],0,sizeof(ch[0]));sz=1;}
26      int idx(char x){return x-'a';}
27      void insert(char *s)
28      {
29          int u=0;
30          int n=strlen(s);
31          for(int i=0;i<n;i++)
32          {
33              int c=idx(s[i]);
34              if(!ch[u][c])
35              {
36                  memset(ch[sz],0,sizeof(ch[sz]));
37                  val[sz]=0;
38                  ch[u][c]=sz++;
39              }
40              u=ch[u][c];
41          }
42          val[u]++;
43      }
44      void getfail()
45      {
46          queue<int> q;
47          fail[0]=0;
48          int u=0;
49          for(int i=0;i<SIGMA_SIZE;i++)
50          {
51              u=ch[0][i];
52              if(u){q.push(u);fail[u]=0;last[u]=0;}
53          }
54          while(!q.empty())
55          {
56              int r=q.front();q.pop();
57              for(int i=0;i<SIGMA_SIZE;i++)
58              {
```

```
59                          u=ch[r][i];
60                          if(!u){ch[r][i]=ch[fail[r]][i];continue;}
61                          q.push(u);
62                          int v=fail[r];
63                          while(v&&!ch[v][i])v=fail[v];
64                          fail[u]=ch[v][i];
65                          last[u]=val[fail[u]]?fail[u]:last[fail[u]];
66                      }
67                  }
68          }
69          int find(char *s)
70          {
71                  int u=0,cnt=0;
72                  int n=strlen(s);
73                  for(int i=0;i<n;i++)
74                  {
75                      int c=idx(s[i]);
76                      u=ch[u][c];
77                      int temp=0;   //必须赋初值为 0，表示如果下面两个判断都不成立，while 可正常执行
78                      if(val[u])
79                          temp=u;
80                      else if(last[u])
81                          temp=last[u];
82                      while(temp)
83                      {
84                          cnt+=val[temp];
85                          val[temp]=0;
86                          temp=last[temp];
87                      }
88                  }
89                  return cnt;
90          }
91  }tree;
92  int main()
93  {
94          int T;
95          scanf("%d",&T);
96          while(T--)
97          {
98                  scanf("%d",&n);
99                  tree.clear();
```

```
100             for(int i=1;i<=n;i++)
101             {
102                 scanf("%s",t);
103                 tree.insert(t);
104             }
105             tree.getfail();
106             scanf("%s",s);
107             int ans=tree.find(s);
108             printf("%d\n",ans);
109         }
110         return 0;
111     }
```

2.4　后缀数组

前面曾经提到过 Aho-Corasick 自动机算法，用以解决多模板匹配问题。其前提是需要事先知道所有的模板，但在实际应用中，无法事先知道查询内容，比如在搜索引擎中，每次的查询是不可能直接预处理出来的，这就需要预处理文本串而非每次的查询内容。

后缀数组是处理字符串的有力工具。后缀数组是后缀树的一个非常精巧的替代品。它比后缀树容易编程实现，能够实现后缀树的很多功能，时间复杂度也并不逊色，而且比后缀树所占用的内存空间小很多。可以说，在信息学竞赛中，后缀数组比后缀树更为实用。本节首先介绍构造后缀数组的方法，重点介绍如何用简洁高效的代码实现后缀数组；接着介绍后缀数组在各种类型题目中的具体应用。

2.4.1　后缀数组基本原理

首先定义两个概念：

（1）**后缀数组：**后缀数组 sa 是一个一维数组，它保存 1，…，n 的某个排列 sa[1]，sa[2]，…，sa[n]，并且保证 suffix(sa[i]) < suffix(sa[i+1])，$1 \leqslant i < n$。也就是将 sa 的 n 个后缀从小到大进行排序之后，把排好序的后缀的开头位置顺次放入 sa 数组中。

（2）**名次数组：**名次数组 rank[i]保存的是 suffix(i)在所有后缀中从小到大排列的“名次”，如图 2.9 所示。

简单来说，后缀数组是“排第几的是谁？”，名次数组是“排第几？”。容易看出，后缀数组和名次数组为互逆运算。

设字符串的长度为 n，为了方便比较大小，可以在字符串后面添加一个字符，这个字符没有在前面的字符中出现过，而且比前面的字符都要小。在求出名次数组后，可以仅用 $O(1)$

的时间比较任意两个后缀的大小。在求出后缀数组或名次数组中的其中一个后，便可以用 $O(n)$ 的时间求出另外一个。

下面介绍一种常用的用来实现后缀数组的倍增算法。

倍增算法的主要思路是用倍增的方法对每个字符开始的长度为 $2k$ 的子字符串进行排序，求出排名，即 rank 值。k 从 0 开始，每次加 1，当 $2k$ 大于 n 以后，从每个字符开始的长度为 $2k$ 的子字符串就相当于所有的后缀，并且这些子字符串都已经比较出了大小，即 rank 值中没有相同的值，那么此时的 rank 值就是最后的结果。每一次排序都利用上次长度为 $2k$–1 的字符串的 rank 值，那么长度为 $2k$ 的字符串就可以用两个长度为 $2k$–1 字符串的排名作为关键字来表示，然后进行基数排序，便得出了长度为 $2k$ 字符串的 rank 值。以字符串“aabaaaab”为例，整个过程如图 2.10 所示，其中，x、y 是表示长度为 $2k$ 的字符串的两个关键字。

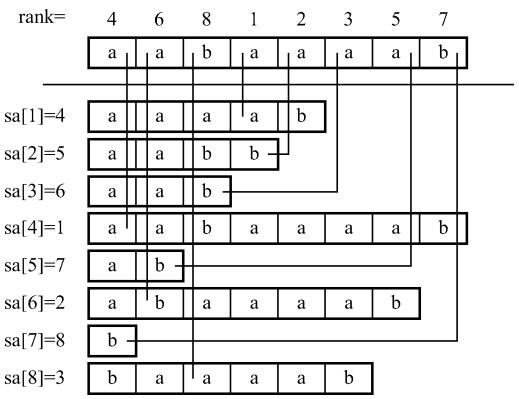


图 2.9　名次数组

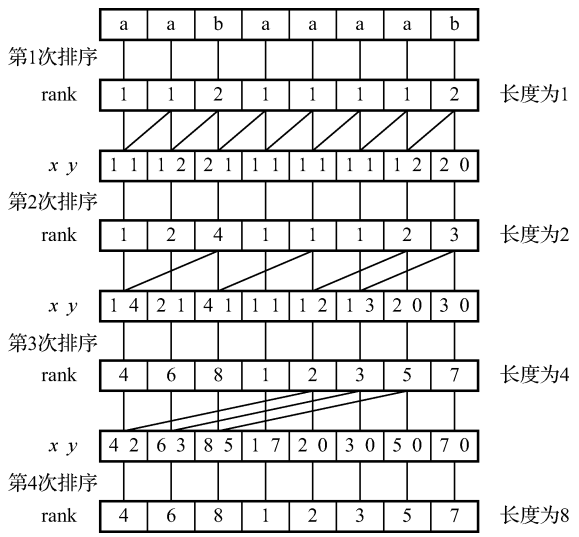


图 2.10　倍增算法

后缀数组倍增算法的实现如下所述。

```
1    int wa[maxn],wb[maxn],wv[maxn],ws[maxn];
2    int cmp(int *r,int a,int b,int l)
3    {return r[a]==r[b]&&r[a+l]==r[b+l];}
4    void da(int *r,int *sa,int n,int m)
5    {
6        int i,j,p,*x=wa,*y=wb,*t;
7        for(i=0;i<m;i++) ws[i]=0;
8        for(i=0;i<n;i++) ws[x[i]=r[i]]++;
9        for(i=1;i<m;i++) ws[i]+=ws[i-1];
```

```
10          for(i=n-1;i>=0;i--) sa[--ws[x[i]]]=i;
11          for(j=1,p=1;p<n;j*=2,m=p)
12          {
13              for(p=0,i=n-j;i<n;i++) y[p++]=i;
14              for(i=0;i<n;i++) if(sa[i]>=j) y[p++]=sa[i]-j;
15              for(i=0;i<n;i++) wv[i]=x[y[i]];
16              for(i=0;i<m;i++) ws[i]=0;
17              for(i=0;i<n;i++) ws[wv[i]]++;
18              for(i=1;i<m;i++) ws[i]+=ws[i-1];
19              for(i=n-1;i>=0;i--) sa[--ws[wv[i]]]=y[i];
20              for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
21              x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
22          }
23          return;
24      }
```

待排序的字符串放在 *r* 数组中，从 *r*[0]到 *r*[*n*−1]，长度为 *n*，且最大值小于 *m*。为了函数操作的方便，约定除 *r*[*n*−1]外的所有 *r*[*i*]都大于 0，*r*[*n*−1]=0。函数结束后，结果放在 sa 数组中，从 sa[0]到 sa[*n*−1]。

函数的第一步，要对长度为 1 的字符串进行排序。一般来说，在字符串的题目中，*r* 的最大值不会很大，所以使用了基数排序。如果 *r* 的最大值很大，那么就把这段代码改成快速排序。代码如下所述。

```
1   for(i=0;i<m;i++) ws[i]=0;
2   for(i=0;i<n;i++) ws[x[i]=r[i]]++;
3   for(i=1;i<m;i++) ws[i]+=ws[i-1];
4   for(i=n-1;i>=0;i--) sa[--ws[x[i]]]=i;
```

x 数组保存的值相当于是 rank 值。下面的操作只是用 *x* 数组比较字符的大小，所以没有必要求出当前真实的 rank 值。

接下来进行若干次基数排序，在实现的时候，有一个优化。基数排序要分两次，第一次是对第二关键字排序，第二次是对第一关键字排序。对第二关键字排序的结果实际上可以利用上一次求得的 sa 直接算出，没有必要再算一次，其代码如下所述。

```
1   for(p=0,i=n-j;i<n;i++) y[p++]=i;
2   for(i=0;i<n;i++) if(sa[i]>=j) y[p++]=sa[i]-j;
```

其中，变量 *j* 是当前字符串的长度，数组 *y* 保存的是对第二关键字排序的结果，然后对第一关键字进行排序，其代码如下所述。

```
1    for(i=0;i<n;i++) wv[i]=x[y[i]];
2    for(i=0;i<m;i++) ws[i]=0;
3    for(i=0;i<n;i++) ws[wv[i]]++;
4    for(i=1;i<m;i++) ws[i]+=ws[i-1];
5    for(i=n-1;i>=0;i--) sa[--ws[wv[i]]]=y[i];
```

这样求出了新的 sa 值。在求出 sa 值后，下一步是计算 rank 值。要注意的是，可能有多个字符串的 rank 值是相同的，所以必须比较两个字符串是否完全相同，*y* 数组的值已经没有必要保存，为了节省空间，用 *y* 数组保存 rank 值。有一个优化，将 *x* 和 *y* 定义为指针类型，复制整个数组的操作可以用交换指针的值代替，不必逐个地复制数组中的值，其代码如下所述。

```
1    for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
2    x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
```

其中，cmp 函数的代码如下。

```
1    int cmp(int *r,int a,int b,int l)
2    {return r[a]==r[b]&&r[a+l]==r[b+l];}
```

规定 *r*[*n*−1]=0 的优点在于如果 *r*[*a*]=*r*[*b*]，说明以 *r*[*a*]或 *r*[*b*]开头的长度为 l 的子串肯定不包括 *r*[*n*−1]，所以调用变量 *r*[*a*+l]和 *r*[*b*+l]不会导致数组下标越界，就不需要做特殊判断。执行完上面的代码后，rank 值保存在 *x* 数组中，而变量 *p* 的结果实际上就是不同的子串的个数。可以优化，如果 *p* 等于 *n*，那么函数可以结束。因为在当前长度的字符串中，已经没有相同的字符串，接下来的排序不会改变 rank 值。对上面的两段代码，循环的初始赋值和终止条件如下所述。

```
1    for(j=1,p=1;p<n;j*=2,m=p)
2    {……}
```

在第一次排序以后，rank 数组中的最大值小于 *p*，所以令 *m*=*p*。整个倍增算法基本写好，代码大约 25 行。倍增算法的时间复杂度为每次基数排序的时间复杂度 *O*(*n*)，排序的次数取决于最长公共子串的长度。最坏的情况下，排序次数为 log*n* 次，所以总的时间复杂度为 *O*(*n*log*n*)。

2.4.2 后缀数组的应用

先介绍后缀数组的一些性质。

height 数组：定义 height[*i*]=suffix(sa[*i*−1])和 suffix(sa[*i*])的最长公共前缀，也就是排名相邻的两个后缀的最长公共前缀。那么对于 *j* 和 *k*，不妨设 rank[*j*]<rank[*k*]，则有以下性质。

suffix(*j*)和 suffix(*k*)的最长公共前缀为 height[rank[*j*]+1]，height[rank[*j*]+2]，height[rank[*j*]+3]，…，height[rank[*k*]]中的最小值。

例如，字符串为“aabaaaab”，求后缀“abaaaab”和后缀“aaab”的最长公共前缀，如

图 2.11 所示。

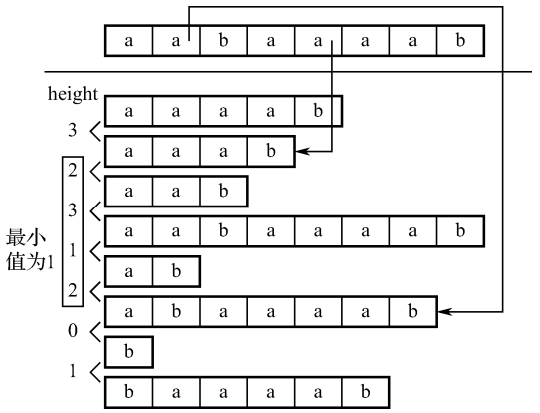


图 2.11　后缀数组

那么应该如何高效地求出 height 值呢？

如果按 height[2]，height[3]，…，height[n]的顺序计算，最坏情况下的时间复杂度为 $O(n^2)$，而没有利用字符串的性质。定义 $h[i]$=height[rank[i]]，也就是 suffix(i)和在它前一名的后缀的最长公共前缀。

h[]数组有以下性质：$h[i] \geqslant h[i-1]-1$。

证明：设 suffix(k)是排在 suffix(i−1)前一名的后缀，则它们的最长公共前缀是 $h[i-1]$。那么 suffix(k+1)将排在 suffix(i)的前面（要求 $h[i-1]>1$，如果 $h[i-1] \leqslant 1$，原式显然成立），并且 suffix(k+1)和 suffix(i)的最长公共前缀是 $h[i-1]-1$，所以 suffix(i)和在它前一名的后缀的最长公共前缀至少是 $h[i-1]-1$。按照 h[1],h[2],…,h[n]的顺序计算，并利用 h 数组的性质，时间复杂度可以降为 $O(n)$。

```
1    int rank[maxn],height[maxn];
2    void calheight(int *r,int *sa,int n)
3    {
4        int i,j,k=0;
5        for(i=1;i<=n;i++) rank[sa[i]]=i;
6        for(i=0;i<n;height[rank[i++]]=k)
7        for(k?k--:0,j=sa[rank[i]-1];r[i+k]==r[j+k];k++);
8        return;
9    }
```

例 1：最长公共前缀。给定一个字符串，询问某两个后缀的最长公共前缀。

算法分析：按照上面所说的做法，求两个后缀的最长公共前缀可以转化为求某个区间上的最小值。对于这个 RMQ（Range Minimum Query）问题，可以用 $O(n\log n)$的时间先预处理，

以后每次回答询问的时间复杂度为 $O(1)$。所以预处理的时间复杂度为 $O(n\log n)$，每次回答询问的时间复杂度为 $O(1)$。如果 RMQ 问题用 $O(n)$的时间复杂度预处理，那么本问题预处理的时间复杂度可以做到 $O(n)$。

例 2：可重叠最长重复子串。给定一个字符串，求最长重复子串，这两个子串可以重叠。

算法分析：这道题是后缀数组的一个简单应用，算法比较简单，只需要求 height 数组里的最大值即可。首先求最长重复子串，等价于求两个后缀的最长公共前缀的最大值。因为任意两个后缀的最长公共前缀都是 height 数组里某一段的最小值，那么这个值一定不大于 height 数组里的最大值，所以最长重复子串的长度就是 height 数组里的最大值。这个算法的时间复杂度为 $O(n)$。

例 3：不可重叠最长重复子串。给定一个字符串，求最长重复子串，这两个子串不能重叠。

算法分析：这题比上一题稍复杂一点。先二分答案，把题目变成判定性问题，判断是否存在两个长度为 k 的子串是相同的，且不重叠。解决这个问题的关键还是利用 height 数组，把排序后的后缀分成若干组。其中每组后缀之间的 height 值都不小于 k。例如，字符串为“aabaaaab”，当 k=2 时，后缀分成了 4 组，如图 2.12 所示。

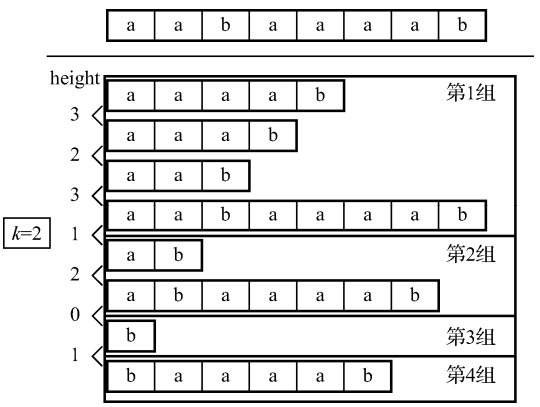


图 2.12　最长重复子串

容易看出，有希望成为最长公共前缀且不小于 k 的两个后缀一定在同一组中。对于每组的后缀，只需要判断每个后缀的 sa 值的最大值和最小值之差是否不小于 k。如果有一组满足，则说明存在，否则不存在。算法的时间复杂度为 $O(n\log n)$。本题中利用 height 值对后缀进行分组的方法很常用。

例 4：可重叠的 k 次最长重复子串。给定一个字符串，求至少出现 k 次的最长重复子串，k 个子串可以重叠。

算法分析：先二分答案，然后将后缀分成若干组，判断有没有一个组的后缀个数不小于 k。如果有，那么存在 k 个相同的子串满足条件，否则不存在。算法的时间复杂度为 $O(n\log n)$。

例 5：不相同子串的个数。给定一个字符串，求不相同的子串的个数。

算法分析：每个子串一定是某个后缀的前缀，那么原问题等价于求所有后缀之间的不相同的前缀的个数。如果所有的后缀按照 suffix(sa[1])，suffix(sa[2])，suffix(sa[3])，…，suffix(sa[n]) 的顺序计算，不难发现，对于每一次新加进来的后缀 suffix(sa[k])，它将产生 n−sa[k]+1 个新的前缀，但其中有 height[k]个和前面的字符串的前缀是相同的，所以 suffix(sa[k])将“贡献”出 n−sa[k]+1−height[k]个不同的子串，累加后便是原问题的答案。算法时间复杂度为 $O(n)$。

2.4.3　例题讲解

例 2-4　Distinct Substrings

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 131072/131072 KB(Java/Others)

题目描述：

给定一个字符串，需要找出所有的不同的子串。

输入：

第一行输入测试数据 T（≤20）组，接下来在每行中包括一个字符串（字符串长度不超过 1000）。

输出：

对于每组数据，都要输出一行，并包含一个整数，该整数表示该字符串有多少个不同的子串。

样例输入：

2
CCCCC
ABABA

样例输出：

5
9

题目来源：SPO J694。

解题思路：

一个字符串中不同子串的总数为∑(len−height[i]−sa[i])。

题目实现：

```
1    #include<iostream>
2    #include<cstdio>
3    #include<cstring>
4    #define MAXN 50005
5    using namespace std;
6    char s[MAXN];
```

```
int sa[MAXN],rank[MAXN],height[MAXN];
int wa[MAXN],wy[MAXN],c[MAXN];
bool cmp(int *y,int i,int k,int n)
{
    int aa=y[sa[i]],bb=y[sa[i-1]];
    int cc=sa[i]+k<n?y[sa[i]+k]:0,dd=sa[i-1]+k<n?y[sa[i-1]+k]:0;
    return aa==bb&&cc==dd;
}
void build(int n,int m)
{
    int *x=wa,*y=wy,i;
    for(i=0;i<m;i++) c[i]=0;
    for(i=0;i<n;i++) c[x[i]=s[i]]++;
    for(i=1;i<m;i++) c[i]+=c[i-1];
    for(i=n-1;i>=0;i--) sa[--c[x[i]]]=i;
    for(int k=1;k<=n;k<<=1)
    {
        int p=0;
        for(i=n-k;i<n;i++) y[p++]=i;
        for(i=0;i<n;i++) if(sa[i]>=k) y[p++]=sa[i]-k;
        for(i=0;i<m;i++) c[i]=0;
        for(i=0;i<n;i++) c[x[y[i]]]++;
        for(i=1;i<m;i++) c[i]+=c[i-1];
        for(i=n-1;i>=0;i--) sa[--c[x[y[i]]]]=y[i];
        m=2; swap(x,y); x[sa[0]]=1;
        for(i=1;i<n;i++)
            x[sa[i]]=cmp(y,i,k,n)?m-1:m++;
        if(m>n) break;
    }
    for(i=0;i<n;i++) rank[sa[i]]=i;
    int h=0;
    for(i=0;i<n;i++)
    {
        if(h) h--;
        if(rank[i]==0) continue;
        int j=sa[rank[i]-1];
        while(s[i+h]==s[j+h]) h++;
        height[rank[i]]=h;
    }
}
long long solve(int n)
```

```
{
    long long ans=0;
    for(int i=0;i<n;i++) ans+=n-sa[i]-height[i];
    return ans;
}
int main()
{
    int k,n;scanf("%d",&k);
    while(k--)
    {
        scanf("%s",s);
        n=strlen(s);
        build(n,300);
        long long ans=solve(n);
        printf("%I64d\n",ans);
    }
    return 0;
}
```

例 2-5　Musical Theme

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 131072/131072 KB(Java/Others)

题目描述：

用 N（$1 \leqslant N \leqslant 20000$）个音符的序列来表示一首乐曲，每个音符都是 1～88 范围内的整数，现在要找一个重复的主题。主题是整个音符序列的一个子串，需要满足如下条件：

（1）长度至少为 5 个音符。

（2）在乐曲中重复出现（可能经过转调，转调的意思是在主题序列的每个音符上都加上或减去同一个整数值）。

（3）重复出现的同一主题不能有公共部分，即给出一串字符，求不重合的最长重复子串，并且长度大于要求的 5。

输入：

输入数据有多组，每组数据的第一行输入一个整数 N，在接下来一行中的 N 个整数表示音符序列，最后一组测试数据以 0 结束。

输出：

对于每组数据，都要输出一行并包含一个整数，该整数表示该字符串满足条件的最长重复子串长度。

样例输入：

30

25 27 30 34 39 45 52 60 69 79 69 60 52 45 39 34 30 26 22 18

82 78 74 70 66 67 64 60 65 80

0

样例输出：

5

题目来源：POJ 1743。

解题思路：

将 height 值分组，然后记录在二分答案时满足 height 值不小于 p 的 sa[i]的最大、最小值，如果最大值减去最小值不小于 p，就说明两个子串的 lcp 值不小于 p，并且它们的坐标也相差不小于 p。另外，转调的影响可通过求相邻序列的差值解决。

题目实现：

```
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
#define MAXN 22222
#define INF (1<<30)
int wa[MAXN],wb[MAXN],wv[MAXN],ws[MAXN];
int cmp(int *r,int a,int b,int l){
    return r[a]==r[b] && r[a+l]==r[b+l];
}
int sa[MAXN],rank[MAXN],height[MAXN];
void SA(int *r,int n,int m){
    int *x=wa,*y=wb;

    for(int i=0; i<m; ++i) ws[i]=0;
    for(int i=0; i<n; ++i) ++ws[x[i]=r[i]];
    for(int i=1; i<m; ++i) ws[i]+=ws[i-1];
    for(int i=n-1; i>=0; --i) sa[--ws[x[i]]]=i;

    int p=1;
    for(int j=1; p<n; j<<=1,m=p){
        p=0;
        for(int i=n-j; i<n; ++i) y[p++]=i;
        for(int i=0; i<n; ++i) if(sa[i]>=j) y[p++]=sa[i]-j;
        for(int i=0; i<n; ++i) wv[i]=x[y[i]];
        for(int i=0; i<m; ++i) ws[i]=0;
        for(int i=0; i<n; ++i) ++ws[wv[i]];
        for(int i=1; i<m; ++i) ws[i]+=ws[i-1];
        for(int i=n-1; i>=0; --i) sa[--ws[wv[i]]]=y[i];
```

```
30              swap(x,y); x[sa[0]]=0; p=1;
31              for(int i=1; i<n; ++i) x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
32          }
33
34          for(int i=1; i<n; ++i) rank[sa[i]]=i;
35          int k=0;
36          for(int i=0; i<n-1; height[rank[i++]]=k){
37              if(k) --k;
38              for(int j=sa[rank[i]-1]; r[i+k]==r[j+k]; ++k);
39          }
40      }
41
42      int n,a[MAXN],r[MAXN];
43      bool isok(int k){
44          bool flag=0;
45          int mx=-INF,mm=INF;
46          for(int i=2; i<=n; ++i){
47              if(height[i]>=k){
48                  mm=min(mm,min(sa[i],sa[i-1]));
49                  mx=max(mx,max(sa[i],sa[i-1]));
50                  if(mx-mm>k) return 1;
51              }else{
52                  mx=-INF,mm=INF;
53              }
54          }
55          return 0;
56      }
57      int main(){
58          while(~scanf("%d",&n) && n){
59              for(int i=0; i<n; ++i) scanf("%d",a+i);
60              --n;
61              for(int i=0; i<n; ++i) r[i]=a[i+1]-a[i]+88;
62              r[n]=0;
63              SA(r,n+1,176);
64              int l=0,r=n>>1;
65              while(l<r){
66                  int mid=l+r+1>>1;
67                  if(isok(mid)) l=mid;
68                  else r=mid-1;
69              }
70              if(l>=4) printf("%d\n",l+1);
```

```
71                else printf("%d\n",0);
72            }
73            return 0;
74    }
```

2.5 练习题

习题 2-1

题目来源：HDU 5536。

题目类型：01 字典树。

题目思路：把每个数字看成一个 01 字符串插入 Trie 树中去，枚举 i 和 j，然后把 s_i 和 s_j 从 Trie 树中删去。然后在 Trie 树中通过**贪心**算法找与 s_i+s_j 异或得到的最大值。具体匹配的过程为：首先看树中最高位能否异或得到 1，能的话就往能的那个方向走，否则往另外一个方向走。另外**删除**操作是这样实现的，每个节点记录一个 val 值，插入时对所有经过节点的 val 值加 1，删除就将对应节点的 val 值减 1。在树上匹配的时候就只走那些 val 值为正的节点。

习题 2-2

题目来源：HDU 482。

题目类型：01 字典树。

题目思路：求某个数与一些数异或的最大值，是字典树应用的一个经典问题。主要思想是贪心算法，将给定的数按照二进制构成一棵字典树，每一层分别对应各个位数上的 01 状态。然后进行查询，如果对应位置为 0，则要往 1 的方向走；如果对应位置为 1，则要往 0 的方向走。但是要注意，走的前提是对应分支是存在的。

习题 2-3

题目来源：POJ 3764。

题目类型：01 字典树。

题目思路：这道题是要在树中找两个节点，且两个节点之间路径唯一，求最长的异或路径。很明显不能用暴力算法，$O(N^2)$时间复杂度为 100000 个点。首先需要知道一个性质：$a \oplus b=(a \oplus c) \oplus (b \oplus c)$，这样就可以考虑找出 a 与 b 公共的 c，实际上就是求出从根节点到每个节点的异或值，这样任意两个点做异或，即是它们之间的异或路径（相同部分异或抵消了）。先使用 DFS（Depth First Search）方法遍历一遍，找出所有节点到树根的路径异或值，这样就将问题转化成了求这些点中任意两个点的异或值，接下来就很简单了，将使用 DFS 方法所得的每

个点的异或值转化成二进制数后保存在字典树中，然后从高位至低位对字典树进行贪心搜索，找出最大的那个值即可。

习题 2-4

题目来源：HDU 3746。

题目类型：KMP。

题目思路：题目要求的是给定一个字符串，还需要添加几个字符可以构成一个由 n 个循环节组成的字符串。由题目可知，应该先求出字符串的最小循环节的长度。假设字符串的长度为 len，那么最小的循环节 cir = len−next[len]；如果有 len%cir == 0，那么这个字符串就已经是要求的字符串了，不用添加任何字符；如果不是要求的那么需要添加的字符数为 cir−(len−(len/cir)×cir))。如果 cir=1，说明字符串只有一种字符，例如“aaa”；如果 cir=m，说明最小的循环节长度为 m，那么至少还需 m 个字符；如果 m%cir == 0，说明已经不用添加了字符。

习题 2-5

题目来源：POJ 2406。

题目类型：KMP。

题目思路：对于 next 数组表示的模式串，如果第 i 位（设 str[0]为第 0 位）与第 j 位不匹配，则要回到第 next[i]位继续与模式串第 j 位匹配，则模式串第 1 位到 next[n]与模式串第 n−next[n]位到 n 位是匹配的。所以思路和上面一样，如果 n%（n−next[n]）=0，则存在重复连续子串，长度为 n−next[n]。

例如，“a　b　a　b　a　b”，next[]={−1,0,0,1,2,3,4}，next[n]=4，代表前缀“abab”与后缀“abab”相等的最长长度，这说明，“ab”这两个字母为一个循环节，长度=n−next[n]。

习题 2-6

题目来源：BZOJ 1717。

题目类型：后缀数组+二分法。

题目思路：先二分答案，然后将后缀分成若干组。判断有没有一个组的后缀个数不小于 k。如果有，那么存在 k 个相同的子串满足条件，否则不存在。

习题 2-7

题目来源：BZOJ 2251。

题目类型：后缀数组。

题目思路：首先根据后缀数组的一些经典应用，可以知道每个后缀对子串个数的贡献是

$n-H[i]-$sa$[i]$，而且顺序就是字典序，枚举每个子串暴力的用 H 数组前后求匹配：L 为向前最多到哪，R 为向后最多到哪，答案是 $R-L$。还有更简单的 Trie 树方法。

习题 2-8

题目来源：BZOJ3238。

题目类型：后缀数组。

题目思路：首先这个式子可以拆成两部分计算，一部分是两个后缀长度之和，一部分是最长公共前缀（Longest Common Prefix，LCP）长度之和。第一部分很简单，发现每个长度的后缀都计算了 $n-1$ 次，所以第一部分的答案为$(n-1)\times[n\times(n+1)/2]$，注意计算过程中需要强制类型转换。而第二部分和之前的一个模型类似，需要用到单调栈。两个后缀的 LCP 在 h 数组里其实就是一段区间的最小值，而反过来 h 数组每段区间的最小值对应着两个后缀的 LCP，然后计算总和。单调栈维护一个递增的、用 $f[i]$表示 h 数组中从 i 到 n 所有区间的 LCP 之和，每次计算 $f[i]$，$f[i]=f[$st[top]$]+h[i]\times($st[top]$-i)$，然后计入总和就行。

习题 2-9

题目来源：HDU6194。

题目类型：后缀数组。

题目思路：先考虑至少出现 k 次的子串，所以枚举排好序的后缀 $i($sa$[i])$。假设当前枚举的是 sa$[i]$～sa$[i+k-1]$，假设这一段的最长公共前缀是 L 的话，那么就有 L 个不同的子串至少出现了 k 次，要减去至少出现 $k+1$ 次的子串，但还要和这个 k 段的 LCP 有关系，因此肯定就是 sa$[i]$～sa$[i+k-1]$这一段向上找一个后缀或者向下找一个后缀。即 sa$[i-1]$～sa$[i+k-1]$和 sa$[i]$～sa$[i+k]$求两次 LCP 减去即可。但是会减多了，减多的显然是 sa$[i-1]$～sa$[i+k]$的 LCP，加上即可。注意 $k=1$ 的情况在求 LCP 会有问题，即求一个字符串的最长公共前缀会有问题，特判一下即可。

第 3 章

动态规划进阶算法

动态规划（Dynamic Programming，DP）是运筹学的一个分支，是求解决策过程最优化的数学方法，其核心思想在于把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关系，逐个求解。

动态规划程序设计是对解最优化问题的一种途径、一种方法，而不是一种特殊算法。不像搜索或数值计算那样，具有一个标准的数学表达式和明确清晰的解题方法。动态规划程序设计往往针对一种最优化问题，由于各种问题的性质不同，确定最优解的条件也互不相同，因而对于不同的问题，动态规划的设计方法有各具特色的解题方法，而不存在一种万能的动态规划算法可以解决各类最优化问题。读者在学习时，除了要正确理解基本概念和方法，必须具体问题具体分析处理，以丰富的想象力去建立模型，用创造性的技巧去求解。读者可以通过对若干有代表性的问题的动态规划算法进行分析、讨论，逐渐学会并掌握这一设计方法。本章将介绍一些动态规划进阶算法，以及一些动态规划的优化方法。

3.1 树状 DP

如果一个问题可以分解成若干相互联系的阶段，在每一个阶段都要做出决策，全部过程的决策则是一个决策序列。使整个活动的总体效果达到最优的问题，称为多阶段决策问题，动态规划是解决多阶段决策最优化问题的一种思想方法。如果一个在树状结构上建立的问题能够分解成它的子树相关阶段，那么就可以在树状结构上进行动态规划，即树状 DP。树状 DP 有以下特殊性：没有环，DFS 不会重复，具有明显且严格的层数关系。利用这一特性，可以很清晰地根据题目写出一个在树状结构上的记忆化搜索程序。使用节点编号可以表示以该节点为根的树，然后根据题目给逻辑关系不断向子树转移，从而找到结果。

3.1.1 树状 DP 的定义

树状 DP，即树状的动态规划，是指在树状结构上的记忆化搜索的程序。这里是指将一个

基于树状结构上的问题，分解成它的子树相关阶段而进行的动态规划算法，而不是单纯地以树为背景题目的动态规划。

3.1.2 树状 DP 解题方法

（1）判断是否是一道树状 DP 题，即判断数据结构是否一棵树，然后判断是否符合动态规划的要求。典型地，我们可以判断问题是否可以分为一些相关的子阶段，并且这些阶段对应着数据结构中的子树。

（2）建树：通过数据量和题目要求，选择合适的树存储方式。如果节点数小于 5000，那么可以用邻接矩阵存储，如果更大可以用邻接表来存储。如果是二叉树或者是需要多叉转二叉，那么可以用两个一维数组 brother[]、child[]来存储。主要的建树过程是通过递归完成的。

（3）写出树状 DP 方程：通过观察孩子和父亲之间的关系建立方程。通常认为，树状 DP 的写法有两种：

① 根到叶子：这种动态规划在实际的问题中运用得不多。

② 叶子到根：即根的子节点传递有用的信息给根，根得出最优解的过程。

一般使用一个节点的编号来表示以该节点为根的子树，这样用一个数来表示一棵树，就可以方便地定义函数 DP。例如，有一棵边上带有权值的树，可以用 DP[i][j]来表示在以 i 为根节点的子树上，保留 j 条边所能获得的最大权值。叶子到根的实质是通过动态规划的方法由主问题不断转移到子问题，求解完各个子问题后不断汇总得到主问题结果的过程。

3.1.3 例题讲解

例 3-1　二叉苹果树

Time Limit: 1000/1000 ms (Java/Others)　　Memory limit:65536/65536 KB(Java/Others)

题目描述：

有一棵苹果树，如果树枝有分叉，一定是二分叉，这棵树共有 N 个节点，编号为 1～N，树根编号一定是 1，用一条树枝两端连接的节点的编号来描述树枝的位置。现在这棵树的树枝太多了，需要剪枝，但是一些树枝上长有苹果。给定需要保留的树枝数量，求出最多能留住多少苹果。

输入：

第 1 行 2 个数，N 和 Q（$1 \leqslant Q \leqslant N, 1 < N \leqslant 100$）。

N 表示树的节点数，Q 表示要保留的树枝数量。接下来 N–1 行描述树枝的信息。

每行 3 个整数，前两个是它连接的节点的编号，第 3 个数表示树枝上苹果的数量，每条树枝上的苹果不超过 30000 个。

输出：

剩余苹果的最大数量。

样例输入：

5 2
1 3 1
1 4 10
2 3 20
3 5 20

样例输出：

21

题目来源：URAL 1018。

思路分析：

（1）确定题目类型：父节点和子节点存在着相互关联的阶段关系，因此这是一道有关树状 DP 的题目。

（2）建树：观察到题目数据量不大，因此选择使用邻接矩阵来表示树。设 ma[x][y]为边的权，因为树是双向的，所以要记录 ma[y][x]。设 tree[v][1]为节点 v 的左子树（实际为左子树根节点值），tree[v][2]为节点 v 的右子树，然后通过递归建树。

（3）列出状态转移方程：为了简化题目，可以将边上的苹果放到这条边连接的两个节点中的子节点上，状态转移图如图 3.1 所示。这样方便 DFS，保留 k 条边，即保留 k+1 个节点。定义 f[v][k]表示以 v 为节点的根保留 k 个节点的苹果最大值，那么有

$$f[v][k]=\max(f[\text{tree}[v][1]][i]+f[\text{tree}[v][2]][k-i-1]+\text{num}[v])),\qquad i=0,\cdots,k-1$$

实际上，是将 k 个节点中的 i 个分配给左子树，分配给右子树 $k-i-1$ 个节点，分配给根节点 1 个节点。

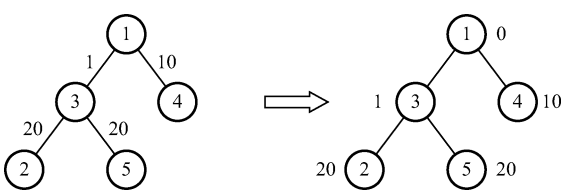


图 3.1　状态转移图

边界条件：剩余 0 个点时，显然 f[v][0]=0。

如果子树仅有一个节点，当 k>1 时，就令 f[v][k]=num[v]。这样，当左子树一共有 m 节点而左子树保留 i（i>m）个节点时，左子树实际保留 m 个节点，这种情况保留的苹果数一定小于左子树保留 i（i≤m）个节点的情况，不会影响结果，因此 i 的取值从 0 到 k–1 就可以了。

注：对于多叉树的情况，处理方法类似，或者将多叉树转化为二叉树。

题目实现：

```
#include<iostream>
#include<iomanip>
#include<cstring>
#include<cstdio>
#include<cmath>
#include<memory>
#include<algorithm>
#include<string>
#include<climits>
#include<queue>
#include<vector>
#include<cstdlib>
#include<map>
using namespace std;

const int ee=105;
int n,q;
int tree[ee][5]={0},ma[ee][ee]={0},num[ee]={0},f[ee][ee]={0};

void preproccess()
{
    for(int i=0;i<=n;i++)
        for(int j=0;j<=n;j++)
        {
            ma[i][j]=-1;
            ma[j][i]=-1;
        }
}

void maketree(int v);

void build(int x,int y,int lor)//lor means left or right
{
    num[y]=ma[x][y];
    tree[x][lor]=y;
    ma[x][y]=-1;ma[y][x]=-1;
    maketree(y);
}

```

```
40  void maketree(int v)
41  {
42      int lr=0;
43      for(int i=0;i<=n;i++)
44          if(ma[v][i]>=0)                  //如果分叉了，则记录
45          {
46              lr++;                        //1 或 2 表示左支或右支
47              build(v,i,lr);               //存入并递归
48              if(lr==2)       return;
49          }
50  }
51
52  void dfs(int v,int k)
53  {
54      if(k==0)       f[v][k]=0;
55      else if(tree[v][1]==0 && tree[v][2]==0) f[v][k]=num[v];
56      else
57      {
58          f[v][k]=0;
59          for(int i=0;i<k;i++)
60          {
61              if(f[tree[v][1]][i]==0)       dfs(tree[v][1],i);
62              if(f[tree[v][2]][k-i-1]==0)       dfs(tree[v][2],k-i-1);
63              f[v][k]=max(f[v][k],(f[tree[v][1]][i]+f[tree[v][2]][k-i-1]+num[v]));
64          }
65      }
66  }
67
68  int main()
69  {
70      cin>>n>>q;
71      preproccess();
72
73      for(int i=0;i<n;i++)
74      {
75          int x,y,xy;
76          scanf("%d%d%d",&x,&y,&xy);
77          ma[x][y]=xy;
78          ma[y][x]=xy;
79      }
80
```

```
81          //建树;
82          maketree(1);
83
84          dfs(1,q+1);
85
86          cout<<f[1][q+1];
87
88          return 0;
89      }
```

3.2 状态压缩 DP

状态压缩 DP 是一种针对集合的 DP，可以用一个整数对应的二进制数表示一个集合，用这个集合来表示当前的状态，通过位运算来转移状态。与普通 DP 相比，状态压缩 DP 的转移状态比较复杂，由于一些特点可以使用一个二进制数来表示，从而达到了压缩状态的效果，其在 DP 思路上与普通 DP 基本类似。

3.2.1 集合的整数表示

当集合元素较少时，可以使用二进制数来表示。集合{0,1,⋯,n−1}的子集 S 可以用如下方式编码成整数：

$$f(S)=\sum_{i\in S}2^i$$

这样表示之后，一些集合的运算可以对应写成如下方式：

（1）空集Φ：0。

（2）只含有第 i 个元素的集合{i}：1<<i。

（3）含有全部 n 个元素的集合{0,1,n−1}：(1<<n)−1。

（4）判断第 i 个元素是否属于集合 S：if(S>>i &1)。

（5）向集合中加入第 i 个元素 S∪{i}：S|1<<i。

（6）从集合中出去第 i 个元素 S\{i}：S&~(1<<i)。

（7）集合 S 和 T 的并集 S∪T：S|T。

（8）集合 S 和 T 的交集 S∩T：S&T。

此外，想要将集合{0,1,⋯,n−1}表示的所有状态枚举出来的，可以写为：

```
for (int S=0; S < 1 << n; S++)
    {//对子集的处理}
```

3.2.2　例题讲解

例 3-2　Travelling by stagecoach

Time Limit: 2000/2000 ms (Java/Others)　　Memory limit:65536/65536 KB(Java/Others)

题目描述：

有一个旅行家计划乘马旅行，他所在的国家里共有 m 个城市，城市之间有若干道路相连。从某个城市沿着某条道路到相邻的城市需要乘坐马车。而乘坐马车需要使用车票，每用一张车票只可以通过一条道路。每张车票上都记有马的匹数，从一个城市移动到另一个城市的所需要时间等于城市之间道路的长度除以马的数量的结果。这位旅行家一共有 n 张车票，第 i 张车票上的马的匹数是 t_i。一张车票只能使用一次，并且换乘所需要的时间可以忽略。求从城市 a 到城市 b 所需要的最短时间。如果无法到达城市 b 则输出“Impossible”。

输入：

多组样例输入输出，以“0 0 0 0 0”结束。例如：

$n\ m\ p\ a\ b$

$t_1\ t_2\ \cdots\ t_n$

$x_1\ y_1\ z_1$

$x_2\ y_2\ z_2$

其中，n 为票数，$1 \leqslant n \leqslant 8$；$m$ 为城市数，$2 \leqslant m \leqslant 30$；$p$ 为道路数，$p \geqslant 0$；a 为起点 $a \geqslant 1$；b 为终点，$b \leqslant m$。

接下来一行 n 个数，表示每张票的马数，$1 \leqslant t_i \leqslant 10$，$1 \leqslant i \leqslant n$。接下来 p 行，每行三个数 x_i、y_i、z_i，表示城市 x_i 和城市 y_i 间的道路长为 z_i，$1 \leqslant z_i \leqslant 100$，$1 \leqslant i \leqslant p$。

输出：

城市 a 到城市 b 的最短时间或“Impossible”，时间误差不超过 0.001。

样例输入：

2 4 4 2 1
3 1
2 3 3
1 3 3
4 1 2
4 2 5
0 0 0 0 0

样例输出

3.66667

题目来源：POJ 2686。

解题思路：

样例的道路网如图 3.2 所示，输出结果 3.66667=5/3+2/1。

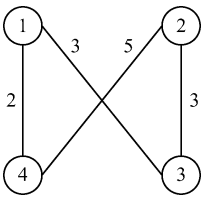


图 3.2　道路网

虽然可以把城市看成顶点、道路看成边来建图，但是由于有车票相关的限制，无法直接使用 Dijkstra 算法求解。不过，这种情况只需要把状态作为顶点，而把状态的转移看成边来建图，就可以很好地避免这个问题。

用一个数 S 来表示当前车票的情况，S 的二进制数的第 i 位表示第 i 张车票是否使用。例如一开始有两张车票，那么当前状态可以用$(11)_2$，即十进制数的 3 来表示，而使用了第一张车票后，状态可以用$(10)_2$，即用十进制数 2 来表示。

考虑一下“现在在城市 v，此时还剩下的车票的集合为 S”这样的状态。从这个状态出发，使用一张车票 $i \in S$ 移动到相邻城市 u，就相当于转移到了“在城市 u，此时还剩下的车票的集合为 $S\backslash\{i\}$”这个状态。把这个转移看成一条边，那么边上的花费是（v–u 间道路的长度）/ t_i。按照上述方法所构成的状态图如图 3.3 所示，这时就可以用普通的 Dijsktra 算法求解了。

集合 S 使用状态压缩的方法表示。由于剩余的车票的集合 S 随着移动元素个数的不断变小，因此这个图实际上是个有向无环图（Directed Acycline Graph，DAG）。计算 DAG 的最短路不需要使用 Dijsktra 算法，可以简单地通过 DP 求解，如图 3.3 所示。

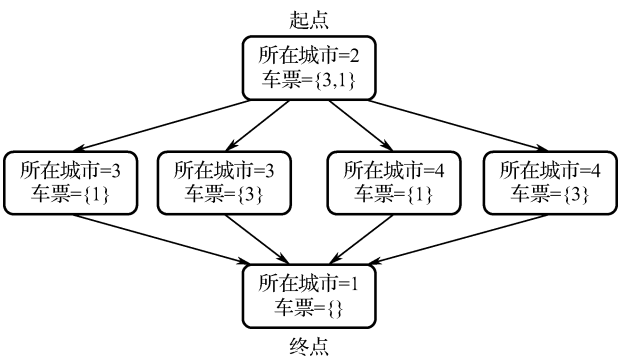


图 3.3　样例所对应的状态图

题目实现：

```
#include <cstdio>
#include <algorithm>
#include <cstring>
using namespace std;
const int maxn = 1 << 10;
const int maxm = 31;
const int INF = 1 << 29;

int n, m, p, a, b;
```

```
int t[maxm];
int d[maxm][maxm];                          //图的邻接矩阵表示（-1 表示没有边）

//dp[S][v] :=  到达剩下的车票集合为 S 并且现在在城市 v 的状态所需要的最小花费
double dp[maxn][maxm];

void solve()
{
    for (int i = 0; i < (1 << n); i++)
        fill(dp[i], dp[i] + m + 1, INF);        //用足够大的值初始化

    dp[(1 << n) - 1][a] = 0;
    double res = INF;
    for (int i = (1 << n) - 1; i >= 0; i--){
        for (int u = 1; u <= m; u++){
            for (int j = 0; j < n; j++){
                if (i & (1 << j)){
                    for (int v = 1; v <= m; v++){
                        if (d[v][u]){       //使用车票 i，从 v 移动到 u
                            dp[i & ~(1 << j)][v] = min(dp[i & ~(1 << j)][v], dp[i][u] +
                                                    (double)d [u][v] / t[j]);
                        }
                    }
                }
            }
        }
    }
    for (int i = 0; i < (1 << n); i++)
        res = min(res, dp[i][b]);
    if (res == INF)                         //无法到达
        printf("Impossible\n");
    else
        printf("%.3f\n", res);
}

int main()
{
    while(scanf("%d%d%d%d%d", &n, &m, &p, &a, &b)!= EOF){
        if(!n && !m)
        break;
        memset(d, 0, sizeof(d));
```

```
50              for (int i = 0; i < n; i++)
51                  scanf("%d", &t[i]);
52              for (int i = 0; i < p; i++){
53                  int u, v, c;
54                  scanf("%d%d%d", &u, &v, &c);
55                  d[u][v] = d[v][u] = c;
56              }
57              solve();
58          }
59          return 0;
60      }
```

3.3 动态规划的优化方法

尽管动态规划是一个时间效率很高的算法，但是有些时候仍然不能满足要求。对于这些情况，可以进一步优化，提高时间效率，主要的优化方向为减少状态总数、每个状态转移的状态数或单个状态的转移时间。本节主要介绍信息学竞赛中常用的三种优化方法。

3.3.1 单调队列优化的动态规划

单调队列，即单调递减或单调递增的队列，是一种特殊的优先队列，提供了两个操作：插入和查询最值。

查询最值：由于优先队列是单调的，所以最值一定是队尾元素，直接取队尾元素即可。

插入操作：当一个数据指针 i（优先级为 A_i）插入单调队列 Q 时，如果队列已空或队头的优先级比 A_i 大，删除队头元素，否则将 i 插入队头。

单调队列主要用于优化形如 dp[i]=max/min {f[k]}+g[i]（k<i && g[i]是与 k 无关的变量）的 dp。

3.3.2 例题讲解

例 3-3　最大子序和

题目描述：

输入一个长度为 n 的整数序列，从中找出一段不超过 M 的连续子序列，使得整个序列的和最大。

输入：

第一行两个数 n 和 m（n、m≤300000）。

第二行有 n 个数。

输出：

一个数，输出最大子序和

样例输入：

6 4

1 −3 5 1 −2 3

样例输出：

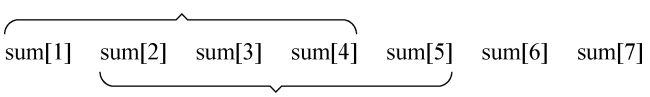
7

题目来源：TYVJ 1305。

解题思路：

如果直接求出所有子序列和，时间复杂度为 $O(n^2)$，显然无法满足要求。可以用线段树的方法来求和，时间复杂度为 $O(n\log n)$，可满足要求。但是当数据规模进一步扩大时，需要更高效的算法。下面介绍一种时间复杂度 $O(n)$的单调队列优化的 DP。

首先，sum[i]表示前 i 个元素的和，$f[i]$表示到第 i 个元素为止的最大子序列和，于是 $f[i]=\text{sum}[i]-\min\{\text{sum}[k]|i-M\leqslant k\leqslant i\}$。$\max\{f[i]\}$即题目所求。现在需要求出所有的 $f[i]$，求 $f[i]$的关键在于 $\min\{\text{sum}[k]|i-M\leqslant k\leqslant i\}$，即对长度为 M 的区间求最小值。对每个区间都单独求解显然是不可能的，但是发现两个相邻的区间有 $M-1$ 个数是相同的，因此直接的想法是利用前一次的结果，当 $f[i]$求解完毕而求解 $f[i+1]$时，可通过维护一个队列 q 来快速获得 min 的部分。



队列 q 保存了 sum 的下标（使用指针同理），并且从队头开始，其中的下标满足对应的 sum 从大到小排列，这样队尾就是最小值的下标。接着求解 $f[i+1]$时，将下标 $i+1$ 入队，并维护队列的单调性，再把不满足 $i+1-M\leqslant k\leqslant i+1$ 的元素出队，就可以不断地从队尾获得 min。关键在于如何插入和维护单调性。

元素入队是按照顺序进行的，较晚入队的元素也较晚才会因为不在区间而出队。因此，当 $i+1$ 入队时，若 sum[$i+1$]比队列中的某些 sum 更小的话，只要 $i+1$ 在区间内，这些队列中的元素就不可能是最小的，而 $i+1$ 入队更晚，出队也一定更晚，因此可以认为那些 sum 比 $i+1$ 大的元素一定是没用的，可以提前从队列剔除。所以，插入一个元素时，会从队头开始一直删除那些 sum 比插入元素对应的 sum 更大的那些元素，最后把元素插入队头。总体来看，每个元素入队和出队一次，因此时间复杂度为 $O(2n)$，即 $O(n)$。下面只给出关键部分的代码，sum 已在读入时求好。

题目实现：

```
long long maxx = 0;
for (int i=1; i<=n; i++) {
    while (!queue.empty() and s[queue.front()] > s[i])
```

```
4               queue.pop_front();
5               // 保持单调性
6               queue.push_front(i);
7               // 插入当前数据
8               while (!queue.empty() and i-m > queue.back())
9                   queue.pop_back();
10              // 维护区间大小，使 i-m >= queue.back()
11              if (i > 1)
12                  maxx = max(maxx, s[i] - s[queue.back()]);
13              else
14                  maxx = max(maxx, s[i]);
15              // 更新最值
16      }
17      cout << maxx << endl;
```

3.3.3 斜率优化的动态规划

单调队列斜率优化 DP 的主要思想是通过对 DP 表达式的处理，得到了类似斜率的式子，通过斜率的单调性来排除一些状态，从而达到优化的效果。在下面的例题中给出了详细的推导过程，请读者结合例题来理解这种优化方法。

3.3.4 例题讲解

例 3-4　Print Article

题目描述：有一台老旧的打印机，要打印 N 个数字，输出的时候可以连续输出。每连续输出一串，它的费用是“这串数字和的平方加上一个常数 M”。给出定 M 和 N 个数，求最小打印费用。

输入：

第一行，两个数 N 和 M，$0 \leqslant N \leqslant 500\,000$, $0 \leqslant M \leqslant 1000$。

接下来的 N 行表示 N 个数。

输出：

仅一行，输出最小打印费用。

样例输入：

```
5 5
5
9
5
7
5
```

样例输出：

230

题目来源：HDU 3507。

思路分析：

设 dp[i]表示输出到 i 的时候最少的花费，sum[i]表示从 a[1]到 a[i]的数字和。于是方程就是：

$$\mathrm{dp}[i]=\mathrm{dp}[j]+M+(\mathrm{sum}[i]-\mathrm{sum}[j])^2$$

很显然是一个二维数组。题目的数字有 500000 个，二维数组就会超时。使用斜率优化就能做到时间复杂度从 $O(n^2)$降到 $O(n)$。

假设 $k<j<i$，如果在 j 的时候决策要比在 k 的时候决策好，那么

$$\mathrm{dp}[j]+M+(\mathrm{sum}[i]-\mathrm{sum}[j])^2<\mathrm{dp}[k]+M+(\mathrm{sum}[i]-\mathrm{sum}[k])^2$$

将平方展开并移项，可以得到：

$$\frac{(\mathrm{dp}[j]+\mathrm{num}[j]^2)-(\mathrm{dp}[k]+\mathrm{num}[k]^2)}{2(\mathrm{num}[j]-\mathrm{num}[k])}<\mathrm{sum}[i]$$

把 dp[j]−num[j]^2 看成 y_j，把 2num[j]看成 x_j。于是有：

$$\frac{y_j-y_k}{x_j-x_k}<\mathrm{sum}[i]$$

左边形似斜率，左边部分记为 g[j,k]。这样，g[j,k]<sum[i]代表这 j 的决策比 k 的决策要更优。对于 $k<j<i$，如果 g[i,j]<g[j,k]，那么 j 点便永远不可能成为最优解，可以直接将它剔除最优解集，证明如下：

（1）g[i,j]<sum[i]，那么就是说 i 点要比 j 点优，排除 j 点。

（2）g[i,j]≥sum[i]，那么 j 点此时是比 i 点要更优，但是同时 g[j,k]>g[i,j]>sum[i]。

这说明还有 k 点会比 j 点更优，同样可排除 j 点。

由于排除了 g[i,j]<g[j,k]的情况，所以整个有效点集呈现一种下凸性质，即 kj 的斜率要大于 ji 的斜率。从左到右，斜率是单调递增的。当最优解取得在 j 点的时候，那么 k 点不可能再取得比 j 点更优的解了，于是 k 点也可以排除。换句话说，j 点之前的点全部不可能再比 j 点更优了，可以全部从解集中排除。

对于斜率优化做法，可以总结如下：

（1）用一个单调队列来维护解集。

（2）假设队列中从头到尾已经有元素 a、b、c。那么当 d 要入队的时候，维护队列的上凸性质，即如果 g[d,c]<g[c,b]，那么就将 c 点删除。直到找到 g[d,x]≥g[x,y]为止，并将 d 点加入该位置中。

（3）求解时候，从队头开始，如果已有元素 a、b、c，当 i 点要求解时，如果 g[b,a]<sum[i]，那么说明 b 点比 a 点更优，a 点可以排除，于是 a 出队。最后 dp[i]=getDp(q[head])。

题目实现：

```
#include<iostream>
#include<string>
using namespace std;

int dp[500005],q[500005],sum[500005];
int head,tail,n,m;

int getDP(int i,int j)
{
    return dp[j]+m+(sum[i]-sum[j])*(sum[i]-sum[j]);
}

int getUP(int j,int k)                          //yj-yk 的部分
{
    return dp[j]+sum[j]*sum[j]-(dp[k]+sum[k]*sum[k]);
}

int getDOWN(int j,int k)                        //xj-xk 的部分
{
    return 2*(sum[j]-sum[k]);
}

int main()
{
    int i;
    freopen("D:\\in.txt","r",stdin);
    while(scanf("%d%d",&n,&m)==2)
    {
        for(i=1;i<=n;i++)
            scanf("%d",&sum[i]);
        sum[0]=dp[0]=0;
        for(i=1;i<=n;i++)
            sum[i]+=sum[i-1];
        head=tail=0;
        q[tail++]=0;
        for(i=1;i<=n;i++)
        {
        while(head+1<tail && getUP(q[head+1],q[head])
            <=sum[i]*getDOWN(q[head+1],q[head]))
```

```
40                      head++;
41                      dp[i]=getDP(i,q[head]);
42              while(head+1<tail && getUP(i,q[tail-1])*getDOWN(q[tail-1],q[tail-2])
43                  <=getUP(q[tail-1],q[tail-2])*getDOWN(i,q[tail-1]))
44                      tail--;
45                  q[tail++]=i;
46              }
47              printf("%d\n",dp[n]);
48          }
49          return 0;
50      }
```

3.3.5　四边形不等式优化的动态规划

四边形不等式优化的动态规划基本原理如下所述。

（1）当函数 $w(i, j)$满足 $w(a,c)+w(b,d)\leqslant w(b,c)+w(a,d)$且 $a\leqslant b<c\leqslant d$ 时，称 $w(i, j)$满足四边形不等式。

（2）当函数 $w(i, j)$满足 $w(i', j)\leqslant w(i, j');i\leqslant i'<j\leqslant j'$时，称 w 具有区间包含的单调性，即如果小区间包含于大区间中，那么小区间的 w 值不超过大区间的 w 值。在动态规划的转移方程中，常见的一种转移方程为：

$$\mathrm{dp}[i][j]=\min_{i\leqslant k\leqslant j}\{\mathrm{dp}[i][k-1]+\mathrm{dp}[k][j]+w[i][j]\}$$

定理 3.1：假如函数 w 满足四边形不等式，那么函数 dp 也满足四边形不等式，即

$$\mathrm{dp}[i][j]+\mathrm{dp}[i'][j']\leqslant \mathrm{dp}[i'][j]+\mathrm{dp}[i][j'],\qquad i<j<i'<j'$$

定义 $s(i,j)$为函数 $\mathrm{dp}(i,j)$对应决策变量 k 的最大值，即

$$s[i][j]=\max\{k\mid \mathrm{dp}[i][j]=\mathrm{dp}[i][k-1]+\mathrm{dp}[k][j]+w[i][j]\}$$

定理 3.2：假如函数 $\mathrm{dp}(i,j)$满足四边形不等式，那么 $s(i,j)$单调，即 $s(i,j)\leqslant s(i+1,j)\leqslant s(i+1,j+1)$这说明决策具有单调性，可以据此来缩小决策枚举的区间，从而进行优化。

由于决策 s 具有单调性，因此状态转移方程可修改为：

$$\mathrm{dp}(i,j)=\min_{s[i][j-1]\leqslant k\leqslant s[i+1][j]}\{\mathrm{dp}[i][k-1]+\mathrm{dp}[k][j]+w[i][j]\}$$

证明过程较为复杂，这里省去。通过限制 k 的范围，可以使时间复杂度从 $O(n^3)$降到 $O(n^2)$。

3.3.6　例题讲解

例 3-5　Post office

题目描述：有 n 个村庄，m 个邮局，给定各村庄的坐标位置。在 n 个村庄建立 m 个邮局，使得所有村庄距它最近邮局的距离和最小，并输出最小值。

输入：

第一行，两个数 n 和 m，$1 \leqslant n \leqslant 300$，$1 \leqslant m \leqslant 30$，$m \leqslant n$。

接下来一行有 n 个数，表示村庄的坐标。

输出：仅一行，最小打印费用。

样例输入：

10 5

1 2 3 6 7 9 11 22 44 50

样例输出：

9

题目来源：POJ 1160。

解题思路：

设 dp[i][j]为前 i 个村庄用 j 个邮局进行覆盖使得每个村庄到其对应的邮局的最短路程和，$w[k,i]$是一个决策变量，表示 k 到 i 这些村庄中建立一个邮局所需的最短路程。转移方程为：

$$dp[i][j]=\min\{dp[k][j-1]+W[k+1,i]\}, \qquad m-1 \leqslant k \leqslant n-1$$

边界条件为：

$$dp[i][1]=W[1,i], \qquad i \geqslant 1 \&\& i \leqslant n$$

状态 dp[i][j]可以由前 k 个村庄用 $j-1$ 个邮局进行覆盖，$k+1$ 到 i 这些村庄可用第 j 个邮局进行覆盖所得的最短路程和转移得到。如果 i 个村庄只用一个邮局进行覆盖的话，该邮局建在村庄的中点位置可以使得总路程最短，那么 $w[k+1,i]$其实就是遍历一遍各个村庄到 $x=[(k+1+i)/2]$这个点的距离，然后求和。

dp 方程本身的时间复杂度是 $O(n^3)$，需要对其应用四边形不等式来优化。根据决策的单调性，有 $k[i]$ $[j-1] \leqslant k \leqslant k[i+1][j]$，由于 $k[i-1][j] \leqslant k \leqslant k[i][j+1]$也满足单调性，因此在这样的状态表示下，这样的区间限制是错的。原因在于 k 的具体含义，$k=k[i][j]$记录的是在动态规划过程中，对应于状态 dp[i][j]的一个最优决策，也可以说 dp[i][j]这个状态是由 k 这个决策而来，通过以上分析，可知决策具有单调性。以上决策可以对后面的状态转移进行决策区间的限制，那么 $k[i][j-1] \leqslant k \leqslant k[i+1][j]$成立，表示前 $j-1$ 个邮局可以覆盖的村庄数 k 一定大于等于前 $j-2$ 个邮局可以覆盖的村庄数，所以左边的不等式成立；而对于右边的不等式，其实只是根据单调性限定的一个上界，四边形不等式进行优化的本质其实是左边的不等式。

题目实现：

```
1   #include<cstdio>
2   #include<algorithm>
3   using namespace std;
4   #define inf 0x7fffffff
5   #define maxn 310
6   int dp[maxn][maxn];                    //dp[i][j]表示前 i 个村庄放 j 个邮局的最短距离
```

```
int w[maxn][maxn];                    //w[i][j]表示[i，j]的最小距离
int val[maxn];
int s[maxn][maxn];                    //s[i][j]记录前 j-1 个邮局的村庄数

int main(){
    int n,m,i,j;
    while(scanf("%d%d",&n,&m)!=EOF)
    {
        for(i = 1 ; i <= n ; i ++) scanf("%d",&val[i]);
        for(int i = 1 ; i <= n ; i ++)      //这里有一个递推公式可以进行预处理
        {   w[i][i] = 0;
            for(int j = i + 1 ; j <= n ; j ++) w[i][j] = w[i][j - 1] + val[j] - val[(j + i)/2];
        }
        for(i = 1 ; i <= n ; i ++)
        {   dp[i][1] = w[1][i];       s[i][1] = 0;
        }
        //for(int i=1;i<=m;i++) s[n+1][i]=n;
        for(i = 2 ; i <= m ; i ++)
        {
            s[n+1][i] = n ;               //s[][]上限的初始化
            for(j = n ; j > i ; j --)
            {
                dp[j][i] = inf;
                for(int k = s[j][i-1]; k <= s[j+1][i] ; k ++)
                {
                    int tmp = dp[k][i-1] + w[k + 1][j];
                    if(tmp < dp[j][i])
                    {       dp[j][i] = tmp;       s[j][i] = k;
                    }
                }
            }
        }
        printf("%d\n",dp[n][m]);
    }
}
```

3.4　练习题

习题 3-1

题目来源：POJ 1655。

题目类型：简单树状 DP。

思路分析：用 num[i]表示以 i 为根节点的子树所包含节点的个数，而 num[i]的求解也很简单，num[i]=sum[num[k]]+1，其中 k 为 i 的子节点。删除一个节点，会把树分割为该节点的子树以及原树去掉以该节点为根的子树的剩余部分，剩余部分=num[1]−num[i]。对于 i，其 balance 可以由 max{num[i]=sum[num[k]]+1}求得，其中 k 为 i 的子节点。

习题 3-2

题目来源：POJ 3254。

题目类型：状态压缩 DP。

思路分析：用一个二进制数来表示一行农田的状态，1 表示放牧，0 表示不放牧，然后将其转换成对应的十进制数。例如，某一行为 101，则可以用 5 来表示。定义 dp[i][j]表示截止到第 i 为止，且第 i 行农田状态为 j 的方法总数。不难得到 dp 方程：dp[i][j]=dp[i−1][k_1]+dp[i−1][k_2]+⋯+dp[i−1][k_n]（k_n 为上一行可行状态的编号，共有 n 种可行状态），ans=dp[m][k_1]+dp[m][k_2]+⋯+dp[m][k_n]为所求的结果。

习题 3-3

题目来源：HDU 3401。

题目类型：单调队列优化 DP。

思路分析：构造状态 dp[i][j]，表示第 i 天拥有 j 只股票的时候赚了多少钱。

状态转移有：

（1）从前一天不买不卖：

$$dp[i][j]=max(dp[i-1][j],dp[i][j])$$

（2）从前 i−W−1 天买进一些股：

$$dp[i][j]=max(dp[i-W-1][k]-(j-k)\times AP[i],dp[i][j])$$

（3）从 i−W−1 天卖掉一些股：

$$dp[i][j]=max(dp[i-W-1][k]+(k-j)\times BP[i],dp[i][j])$$

只考虑第 i−W−1 天的买入卖出情况即可，这是因为 i−W−2 天可以通过不买不卖将自己的最优状态转移到第 i−W−1。以此类推，之前的都不需要考虑，只考虑到 i−W−1 天的情况即可。

对买入股票的情况进行分析，转化成适合单调队列优化的方程形式：

$$dp[i][j]=max(dp[i-W-1][k]+k\times AP[i])-j\times AP[i]$$

令

$$f[i-W-1][k]=dp[i-W-1][k]+k*AP[i]$$

则

$$dp[i][j]=max(f[i-W-1][k])-j*AP[i]$$

可以用单调队列进行优化了。卖出股票的情况可进行类似的分析。

习题 3-4

题目来源：HDU 2829。

题目类型：斜率优化 DP。

思路分析：给出一段 n 个数字，然后切 m 次，分成 m+1 段，每段的 cost(j+1,i)=((sum[i]−sum[j])^2−(sum[i]^2−sum[j]^2))/2；通过记录两个数组，即可在时间复杂度 O(1)内求 cost，二维数组的斜率优化和一维数组其实是一样的，只要在求斜率的时候多传进去一个是第 j 次，然后用 j−1 次的数据即可。

习题 3-5

题目来源：HDU 2829。

题目类型：四边形不等式优化 DP。

思路分析：状态转移方程 dp[i][j] = min(dp[i][j],dp[i−1][k]+w[k+1][j])（1≤k<i），时间复杂度是 O(n×n×m)，当 n 为 1000 时运算量为 10 亿级别，必须优化。

四边形不等式优化主要是减少枚举 k 的次数。w[i][j]是某段区间的权值，当区间变大时，权值也随之变大；区间变小时，权值也随之变小，此时就可以用四边形不等式优化。

设 s[i][j]为 dp[i][j]的前导状态，即 dp[i][j]=dp[i−1][s[i][j]]+w[s[i][j]+1][j]，枚举 k 的时候只要枚举 s[i−1][j]<=k<=s[i][j+1]即可，此时 i 必须从小到大遍历，j 必须从大到小遍历。

第 4 章

图论高级算法

4.1 最大流

最大流算法可分成两大类：增广路（Augmenting Path）算法与预流推进（Preflow Push）算法。本节介绍的三个算法都属于增广路算法，增广路算法涉及三个重要概念：残量网络、增广路和割。

4.1.1 最大流的定义

1．网络流的定义

最大流问题（Maximum Flow Problem）是网络流问题（Network Flow Problem）的一种。网络流问题的研究对象是流网络（Flow Network），在某些文献中流网络也称为网络流图（Network Flow Graph）。流网络 $G=(V,E,c,s,t)$ 是一个有向图，是其点集与边集，点和边的数目分别记为 n、m。c 表示 $V\times V\to\mathbb{N}$ 的容量函数，每条边 (u,v) 都有一容量 $c(u,v)\in\mathbb{N}$，其中 u、v 是图的顶点，$u\in V$，$v\in V$。若 $(u,v)\notin E$ 则 $c(u,v)=0$。s 和 t 是流网络中的两个特殊点，分别称为源点和汇点。为简便计，流网络简称“网络”或“图”，简记为 $G=(V,E)$。

自环在网络中无意义，规定图 G 中不含自环。下面在论述、证明关于网络流的原理、性质或定理时，为了表示上的方便，对流网络做出两条限定：

（1）图中不存在重边。

（2）图中不存在反向边，即若 $(u,v)\in E$，则 $(v,u)\notin E$。

这两条限定都不妨碍一般性。可以通过将容量相加把重边合为一条边，反向边可以通过新增一个点来消除。请读者注意，所谓“表示上的方便”是指一条边可以通过两个端点唯一确定。下面要介绍的算法和代码可以处理含有重边或反向边的图，这两条限定都不是根本性的，仅仅是为了方便表述而已。

2．流的定义

流是满足下述两个性质的实值函数 f：$V\times V\to R$。

容量限制：对任意 $(u,v)\in V$，有 $0\leqslant f(u,v)\leqslant c(u,v)$。

流守恒：对任意 $u\in V-\{s,t\}$，有 $\sum_{v\in V}f(v,u)=\sum_{v\in V}f(u,v)$。

$f(u,v)$ 即边 (u,v) 上的流量，若 $(u,v)\notin E$，则 $f(u,v)=0$。从源点 s 到汇点 t 的总流量称为流 f 的值，记为 $|f|$，不难得出

$$|f|=\sum_{v\in V}f(s,v)-\sum_{v\in V}f(v,s)$$

最大流问题即在给定的网络 G 中求一个值最大的流。

4.1.2　增广路算法涉及的三个重要概念

1．残量网络

给定流网络 $G=(V,E,c,s,t)$ 和 G 上的一个流 f。残量网络 $G_{\mathrm{f}}=(V,E_{\mathrm{f}},c_{\mathrm{f}},s,t)$ 是由 G 和 f 所导出的一个网络，简记为 $G_{\mathrm{f}}=(V,E_{\mathrm{f}})$。首先定义残余容量 c_{f}：

$$c_{\mathrm{f}}(u,v)=\begin{cases}c(u,v)-f(u,v), & \text{若}(u,v)\in E\\ f(v,u), & \text{若}(v,u)\in E\\ 0, & \text{其他情况}\end{cases}$$

$(u,v)\in E$ 和 $(v,u)\in E$ 同时成立会给 c_{f} 的定义带来形式上的不便。残量网络 G_{f} 的边集 E_{f} 定义为

$$E_{\mathrm{f}}=\{(u,v)\in V\times V:c_{\mathrm{f}}(u,v)>0\}。$$

除了可能含有反向边，残量网络也符合流网络的定义；已经指出“不含反向边”并非根本性的要求，借助残余容量 c_{f}，可以类似地定义残量网络上的流，称为残量流。

考虑残量流的原因在于，借助残量网络 G_{f} 上的残量流 f'，可以将网络 G 上的流 f 修改成一个值更大的流 $f\uparrow f'$，即用 f' 增广 f，这正是“增广”二字含义所在。增广方法为

$$(f\uparrow f')(u,v)=\begin{cases}f(u,v)+f'(u,v)-f'(v,u), & \text{若}(u,v)\in E\\ 0, & \text{其他情况}\end{cases}$$

不难证明 $|f\uparrow f'|=|f|+|f'|$。

2．增广路

增广路是残量网络 G_{f} 上从 s 到 t 的一条简单路径。有了增广路 p，很容易得到一个残量流 f_{p}。

$$f_p(u,v)=\begin{cases}c_f(p), & \text{边}(u,v)\text{在路径}p\text{上}\\ 0, & \text{其他情况}\end{cases}$$

式中，$c_f(p)=\min\{c_f(u,v):(u,v)\text{在路径}p\text{上}\}$，$c_f(p)$ 称为路径 p 的残余容量。易见，$|f_p|=c_f(p)>0$。

增广路方法是指，从图 G 上的某个初始流 f（如零流）开始，在 G_f 找一条增广路 p；沿着 p 增广，更新 f 和 G_f；如此循环，直到 G_f 上找不到增广路为止，此时 f 便是 G 上的一个最大流。下面要介绍的最大流最小割定理证明了增广路方法的正确性。

3．割

为了给出最大流最小割定理，先介绍割的概念。将流网络 $G=(V,E)$ 的点集 V 划分成两个子集 S 和 T，$T=V-S$，使得 $s\in S$ 且 $t\in T$，(S,T) 称为 G 的一个割或者 $s-t$ 割。另外，也可以将割定义成两端点分属 S 和 T 的边的集合。将满足 $u\in S$ 且 $v\in T$ 的边 (u,v) 称为割 (S,T) 的前向边（Forward Edge），将满足 $u\in T$ 且 $v\in S$ 的边 (u,v) 称为割 (S,T) 的后向边（Backward Edge）。

令 f 为 G 的一个流，割 (S,T) 之间的净流 $f(S,T)$ 定义为

$$f(S,T)=\sum_{u\in S}\sum_{v\in T}f(u,v)-\sum_{u\in S}\sum_{v\in T}f(v,u)$$

不难证明，对于 G 的任意一个割 (S,T)，都有 $f(S,T)=|f|$。割 (S,T) 的容量 $c(S,T)$ 定义为

$$c(S,T)=\sum_{u\in S}\sum_{v\in T}c(u,v)$$

网络的最小割即所有割之中容量的最小者。显然，对于 G 上的任意一个流 f 和 G 的任意一个割 (S,T)，都有 $f\leqslant c(S,T)$。

定理 4.1（最大流最小割定理） 若 f 是流网络 $G=(V,E,c,s,t)$ 上的一个流，则下列三个命题等价。

（1）f 是 G 上的一个最大流。

（2）残量网络 G_f 上无增广路。

（3）存在某个割 (S,T)，满足 $|f|=c(S,T)$。

证明：如果命题（1）成立，那么可推导出命题（2）成立，显然如果命题（2）成立，可推导出命题（3）成立。假设 G_f 中无增广路，即 G_f 上不存在从 s 到 t 的路径。令 $S=\{v\in V:G_f\text{上有从}s\text{到}v\text{的路径}\}$，$T=V-S$，易见 $t\notin S$，故 (S,T) 是一个割。考虑两点 $u\in S$ 和 $v\in T$。若 $(u,v)\in E$，则必有 $f(u,v)=c(u,v)$；否则有 $(u,v)\in E_f$，即 $v\in S$。若 $(v,u)\in E$，则必有 $f(v,u)=0$；否则有 $c_f(u,v)=f(v,u)>0$，即 $(u,v)\in E_f$，仍有 $v\in S$。若 $(u,v)\notin E$ 且 $(v,u)\notin E$，则 $f(u,v)=f(v,u)=0$。因此有

$$\begin{aligned} f(S,T) &= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) \\ &= \sum_{u\in S}\sum_{v\in T} c(u,v) - \sum_{v\in T}\sum_{u\in S} 0 \\ &= c(S,T) \end{aligned}$$

所以$|f| = f(S,T) = c(S,T)$。

如果命题（3）成立，可推导出命题（1）成立。由于对任意割(S,T)都有$|f| \leqslant c(S,T)$，故$|f| = c(S,T)$蕴含着f是一个最大流。

证毕。

不难看出，高效地实现增广路方法应从两个方面考虑：如何快速地在残量网络G_f上找一条增广路；如何减少增广的次数。

已经知道，通过深度优先搜索（DFS）或宽度优先搜索（BFS）可在$O(V+E)$的时间复杂度内找到一条增广路。流网络G上的孤立点是无意义的，所以每个点至少有一条边与之相连，故有$|V| \leqslant 2|E|$，因此在G_f上找一条增广路的时间复杂度为$O(E)$，从而增广路方法的复杂度不超过$O(|f^*|E)$，$|f^*|$表示最大流的值。

下面将证明，如果每次都沿着最短增广路（Shortest Augmenting Path，SAP）增广，那么增广次数是$O(VE)$的。沿着最短增广路增广的算法统称为最短增广路算法。下面三节要介绍的算法都属于最短增广路算法。

4.1.3　Edmonds-Karp 算法

Edmonds-Karp 是 SAP 算法的朴素实现。采用链式前向星存图，对任意边$e \in E$，e在边数组中的下标$\mathrm{idx}(e)$为偶数，e的反向边e'的下标$\mathrm{idx}(e') = \mathrm{idx}(e) + 1$。代码如下所述。

```
#include <climits>
#include <algorithm>
const int N = 1e5 + 5, M = 1e5 + 5;
struct Edge{
    int v, rc, next;      //rc: residual capacity
}E[M * 2];
int head[N], sz, n, m, s, t;
void add_edge(int u, int v, int c){
    E[sz] = {v, c, head[u]};
    head[u] = sz++;
    E[sz] = {u, 0, head[v]};
    head[v] = sz++;
}
void init(){
    sz = 0;
```

```
16        memset(head, -1, sizeof(int[n + 1]));
17    }
18    int pre[N], q[N];
19    int ek(){
20        for(int ans = 0; ; ){
21            int beg = 0, end = 0;
22            memset(pre, -1, sizeof(int[n + 1])); q[end++] = s;
23            while(beg < end){
24                int u = q[beg++];
25                for(int i = 0; i != -1; i = E[i].next){
26                    if(E[i].rc > 0 && pre[E[i].v] == -1)
27                    if(E[i].v == t){
28                        int cp = INT_MAX;
29                        for(int j = i; j != -1; j = pre[E[j^1].v])
30                        cp = std::min(cp, E[j].rc);
31                        for(int j = i; j != -1; j = pre[E[j^1].v])
32                        E[j].rc -= cp, E[j ^ 1].rc += cp;
33                        ans += cp; break;
34                    }
35                    else{
36                        pre[E[i].v] = i;
37                        q[end++] = E[i].v;
38                    }
39                }
40            }
41            if(pre[t] == -1) return ans;
42        }
43    }
```

下面分析 Edmonds-Karp 算法的时间复杂度。用 $\delta_f(u,v)$ 表示残量网络 G_f 上从 u 到 v 的距离，G_f 中边的长度都是1。

定理 4.1 在采用 Edmonds-Karp 算法求流网络 $G=(V,E,c,s,t)$ 的最大流的过程中，对于任意点 $v\in V-\{s,t\}$，残量网络 G_f 上从源点 s 到 v 的距离 $\delta_f(s,v)$ 在每次增广之后不会减小。

证明：假设此命题不成立。设 v 是 $V-\{s,t\}$ 中一点；令 f 为“s 到 v 的距离减小”首次出现之前的流，令 f' 为 f 增广之后的流；再令 v 为满足 $\delta_f(s,v)>\delta_{f'}(s,v)$ 的点中使得 $\delta_{f'}(s,v)$ 最小的一个点。令从 s 经过 u 到达 v 的某条路径 p 为 $G_{f'}$ 中从 s 到 v 的一条最短路，因而有 $(u,v)\in E_{f'}$ 且

$$\delta_{f'}(s,u)=\delta_{f'}(s,v)-1 \tag{4.1}$$

又由于 v 是满足 $\delta_f(s,v)>\delta_{f'}(s,v)$ 的点中使得 $\delta_{f'}(s,v)$ 最小者，有

$$\delta_{f'}(s,u)\geqslant\delta_f(s,u) \tag{4.2}$$

由上两式能推导出 $(u,v)\notin E_f$。若不然，根据 $(u,v)\in E_f$，有

$$\begin{aligned}\delta_f(s,v) &\leqslant \delta_f(s,u)+1 && \text{依据三角形不等式}\\ &\leqslant \delta_{f'}(s,u)+1 && \text{依据式（4.2）}\\ &=\delta_{f'}(s,v) && \text{依据式（4.1）}\end{aligned}$$

这与 $\delta_{f'}(s,v)<\delta_f(s,v)$ 矛盾。

由 $(u,v)\notin E_f$ 且 $(u,v)\in E_{f'}$，可以得知在 G_f 上所选的那条增广路一定经过了边 (v,u)。因此有

$$\begin{aligned}\delta_f(s,v) &= \delta_f(s,u)-1\\ &\leqslant \delta_{f'}(s,u)-1 && \text{依据式（4.2）}\\ &=\delta_{f'}(s,v)-2 && \text{依据式（4.1）}\end{aligned}$$

这与假设 $\delta_{f'}(s,v)<\delta_f(s,v)$ 相矛盾。证毕。

定理 4.2　在流网络 $G=(V,E,c,s,t)$ 上，Edmonds-Karp 算法的总增广次数为 $O(VE)$。

证明：设 p 为残量网络 G_f 中的一条增广路，(u,v) 为 p 上的一条边。若有 $c\_f(p)=c\_f(u,v)$，则称(u,v)为 p 的瓶颈边。不难看出：①沿着 p 增广后，p 上的瓶颈边都消失了；② p 上至少有一条瓶颈边。下面证明：一条边在 G_f 上成为瓶颈边的次数至多为 $\frac{|V|}{2}$。

令(u,v)为残量网络 G_f 中的一条边，当 (u,v) 首次成为瓶颈边时，有

$$\delta_f(s,v)=\delta_f(s,u)+1$$

增广之后，边 (u,v) 将从残余网络中消失。下一次 (u,v) 出现在残量网络中时，必然是在某次 (v,u) 出现在增广路上之后。设上述“(v,u) 成为增广路上的边”这一情况发生时 G 上的流为 f'，则有

$$\delta_{f'}(s,u)=\delta_{f'}(s,v)+1$$

根据定理 4.1，有 $\delta_f(s,v)\leqslant\delta_{f'}(s,v)$，因而有

$$\begin{aligned}\delta_{f'}(s,u) &= \delta_{f'}(s,v)+1\\ &\geqslant \delta_f(s,v)+1\\ &=\delta_f(s,u)+2\end{aligned}$$

所以从某次 (u,v) 成为瓶颈边到 (u,v) 下一次成为瓶颈边时，从源点 s 到 u 的距离至少增加 2。初始时 s 到 u 的距离至少为 0，从 s 到 u 的最短路上的中间点必定不包含 s、u 或者 t[边 (u,v) 在最短路上蕴含着 $u\neq t$]。因此，只要从 s 到 u 的路径存在，s 到 u 的距离至多为 $|V|-2$。所以在 (u,v) 首次成为瓶颈边之后，它最多还能再成为 $\frac{|V|-2}{2}=\frac{|V|}{2}-1$ 次瓶颈边，共计 $\frac{|V|}{2}$ 次。又由于在残量网络上有 $O(E)$ 对点之间可能有边相连，在 Edmonds-Karp 算法运行过程中瓶颈边的总数是 $O(VE)$ 的。证毕。

前面已经指出，通过 BFS 可在 $O(E)$ 的时间复杂度内找到一条最短增广路，因此

Edmonds-Karp 算法的时间复杂度为 $O(VE^2)$。

4.1.4 Dinic 算法

Dinic 算法是对 Edmonds-Karp 算法的改进，其时间复杂度是 $O(n^2m)$，下面给出 Dinic 算法的代码，其中图的表示部分与 Edmonds-Karp 算法的代码相同，故略去。

```
#include <algorithm>
#include <climits>
using namespace std;
int q[N], n, m, s, t;
bool bfs(){
    memset(level, -1, sizeof(int[n + 1]));
    int beg = 0, end = 0;
    level[s] = 0; q[end++] = s;
    while(beg < end){
        int u = q[beg++];
        for(int i = head[u]; i != -1; i = E[i].next)
        if(E[i].rc && level[E[i].v] == -1){
            level[E[i].v] = level[u] + 1;
            if(E[i].v == t) return true;
            q[end++] = E[i].v;
        }
    }
    return false;
}
using LL = long long;
LL dfs(int u, LL rc){
    if(u == t) return rc;
    LL total = 0;
    for(int &i = cur[u]; i != -1; i = E[i].next)
    if(level[E[i].v] == level[u] + 1 && E[i].rc > 0){
        int tmp = dfs(E[i].v, min(rc, LL(E[i].rc)));
        if(tmp > 0){
            E[i].rc -= tmp;
            E[i ^ 1].rc += tmp;
            total += tmp;
            rc -= tmp;
            if(rc == 0) break;
        }
    }
```

```
35          return total;
36      }
37      LL dinic(){
38          LL ans = 0;
39          while(bfs()){
40              memcpy(cur, head, sizeof(int[n + 1]));
41              for(LL f; f = dfs(s, LLONG_MAX); ans += f);
42          }
43          return ans;
44      }
```

先介绍 Dinic 算法用到的两个概念：分层图（Level Graph）和阻塞流（Blocking Flow）。

1．分层图

设 f 是网络 G 上的一个流，以 s 为起点对 G_f 进行一次 BFS，将 s 到 u 的距离记为 $\text{level}(u)$。G_f 的分层图 G'_f 是由 G_f 所导出的一个流网络。G'_f 定义为

$$G'_f = (V, E'_f, c'_f, s, t)$$

式中

$$E'_f = \{(u,v) \in E_f : \text{level}(v) = \text{level}(u) + 1\}$$

$$c'_f(u,v) = \begin{cases} c_f(u,v), & (u,v) \in E'_f \\ 0, & (u,v) \notin E'_f \end{cases}$$

简记为 $G'_f = (V, E'_f)$。

2．阻塞流

首先要指出的是，由 G_f 所导出的分层图 G'_f 完全符合 4.1 节中流网络的定义（G'_f 与 G_f 不同的地方在于 G'_f 中一定不含有反向边）。此外，不难看出 G'_f 中的任意一条从 s 到 t 的路径都是从 s 到 t 的最短路径。

设 f 是网络 $G = (V, E)$ 上的一个流，$(u,v) \in E$ 是 G 上的一条边；若 $f(u,v) = c(u,v)$ 则称边 (u,v) 是饱和的。又设 f' 是分层图 G'_f 上的一个流，若 G'_f 中的任意一条从 s 到 t 的路径上都至少有一条饱和边，则称 f' 是 G'_f 上的一个阻塞流。注意，G'_f 上阻塞流未必是 G'_f 上的最大流，图 4.1 就是一个例子，图中黑色边是非饱和边，灰色边是饱和边。

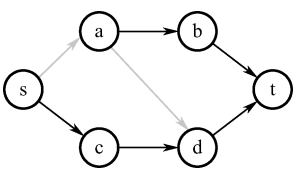


图 4.1　阻塞流非最大流的一个例子

下面考虑如何构造阻塞流。前面已经提到，分层图G'_f是一个流网络，所以一个自然的想法就是在G'_f上不断地进行 DFS 以寻找增广路，并沿着找到的增广路增广。每次增广至少会使一条边饱和，所以至多增广m次，可以在$O(m)$的时间复杂度内在分层图上找到一条从s到t的增广路，所以构造阻塞流的复杂度不超过$O(m^2)$。下面介绍一种称为当前边的优化，可使构造阻塞流的复杂度达到$O(nm)$，另外，还有一个称为多路增广的实现技巧，可进一步优化时间复杂度。

3．当前边优化

对分层图中的每个点u维护一个当前边，初始时u的当前边为u的邻接表中的第一条边。当每次进行 DFS 到点u时，从u的当前边，设为(u,v)，向下递归。若 DFS(v)未找到从v到汇点t的路径，就把u的当前边变为u的邻接表中的下一条边，这相当于将边(u,v)从分层图中删除；若找到了至汇点t的路径就一路回溯。

下面分析这种方法构造阻塞流的复杂度。整个过程的复杂度可以划归为调用 DFS(v)的次数。若 DFS(v)的返回值为“找到了路径”，则这种调用以l个为一组（l为分层图的层数），每次找到一条从s到t的路径至少使一条边饱和，这种调用至多有$lm \leqslant nm$次。若 DFS(v)的返回值为“未找到路径”则有一条边被删除，故这种调用不超过m次。所以构造阻塞流的复杂度为$O(nm)$。

4．多路增广

用$c_f(v)$表示进行 DFS 到v点时从s到v所经过的边的残余容量的最小值，$c_f(s)=\infty$。多路增广是指进行 DFS 到汇点t后不一直向上回溯到源点s，而是一旦回溯到$c_f(v)>0$的点v就从v继续向下递归。这里所说的“一旦回溯到$c_f(v)>0$的点v”也就是找到从s到t的增广路上距离s最近的饱和边的起点。把$c_f(v)$也作为 DFS 的参数，即调用 DFS(v,$c_f(v)$)。当$c_f(v)$变为 0 时就终止 DFS(v,$c_f(v)$)过程。

借助分层图这一概念，可以很直观地理解定理 4.1。Dinic 算法的过程就是不断构造阻塞流，并用阻塞流来增广原来的流f；不难看出，在每次用阻塞流增广f之后，残量网络上的从s到t的的最短路径的长度严格递增，所以 Dinic 算法最多构造$n-1$次阻塞流，因而时间复杂度为$O(n^2m)$。此复杂度上界是比较松的，Dinic 算法的速度能满足大部分实际问题的要求。

4.1.5 ISAP 算法

ISAP 是 Improved Shortest Augmenting Path 的缩写，ISAP 算法来源于 R. K. Ahuja、T. L. Magnanti 和 J. B. Orlin 三人合著的 *Network Flows: Theory, Algorithms and Applications* 一书的 7.4 节 Shortest Augmenting Path Algorithm 中提到的对 SAP 算法的一个优化。在介绍此优化方法之前，先介绍 SAP 算法。顾名思义，SAP 算法也是一种最短增广路算法。在 SAP 算法中

引入了一些新概念，下面一一介绍。

1．距离标号

设 $G(V,E,c,s,t)$ 是一个流网络，f 是 G 上的一个流。在残量网络 $G_f=(V,E_f)$ 上，定义一个距离函数 $d:V\to\mathbb{N}$。若函数 d 满足下面两个条件，则称 d 为合法的距离函数。

$$d(t)=0$$

$$d(u)\leqslant d(v)+1,\quad \forall(u,v)\in E_f$$

$d(u)$ 称为节点 u 的距离标号，上述两条件称为合法条件。

性质 4.1　若距离函数 d 合法，设 p 是残量网络 G_f 上的某条从 u 到 t 的路径，则 p 上的点的距离标号的集合由从零开始的若干连续自然数组成。

性质 4.2　若距离函数 d 合法，则距离标号 $d(u)$ 是残量网络 G_f 上从 u 到 t 的距离的一个下界。

令 $v=v_0\to v_1\to\cdots\to v_{k-1}\to v_k$ 为 G_f 上任意一条从 v 到 t 的长为 k 的路径。合法条件蕴含着：

$$d(v_{k-1})\leqslant d(v_k)+1=d(t)+1=1$$

$$d(v_{k-2})\leqslant d(v_{k-1})+1\leqslant 2$$

$$\vdots$$

$$d(v)=d(v_0)\leqslant d(v_1)+1\leqslant k$$

性质 4.3　若 $d(s)\geqslant n$，则残余网络 G_f 中没有从源点 s 到汇点 t 的路径。

由于 $d(s)$ 是 G_f 上从 s 到 t 的距离的一个下界，又因 $\delta_f(s,t)\leqslant n-1$，所以 $d(s)\geqslant n$ 意味着 G_f 上不存在从 s 到 t 的路径。

若对每个点 $u\in V$，都有 $\delta_f(u,t)=d(u)$，则称距离标记是准确的。

2．允许边和允许路径

若边 $(u,v)\in E_f$ 满足 $d(u)=d(v)+1$，则称 (u,v) 为允许边，否则称为非允许边；若 G_f 的一条从 s 到 t 的路径 p 完全由可行边构成，则称 p 为允许路径，显然有如下性质。

性质 4.4　可行路径是一条最短增广路。

SAP 算法的思想与 Dinic 算法类似[①]。确定一个初始的合法距离标号，SAP 算法重复“DFS 找可行路径”的过程，直到残量网络上不存在从 s 到 t 的路径为止。DFS 由前进（Advance）、后退（Retreat）和重标号（Relabel）三种操作构成，详述如下所述。

从 s 开始，沿着允许边走，试图到达 t；若能到达 t，则沿着找到的最短增广路增广，在新的残量网络上继续寻找允许路径。与 Dinic 算法不同的是，若走到 u 点之后无法继续前进，

① 这里所谓“类似”是指 level(u)与 $d(u)$的意义有相似之处，但仍应注意分层图 G_f' 上的边和残余网络 G_f中的可行边这两个概念的区别：若令 $d(u)$=level(u)，则分层图上的边一定是可行边；反过来则不成立。

即没有以 u 为起点的可行边，则将 $d(u)$ 增大为 $\min\{d(v)+1:(u,v)\in E_f\}$，这一操作称为重标号；然后后退一步继续寻找可行边。上述 DFS 过程采用递归实现比较方便，但顾及效率，下面给出 SAP 算法的迭代实现。在迭代实现中，需要维护：①每个点的“当前边”，点 u 的当前边是指走到 u 时要选择的那条出边；②当前找到的“部分可行路径”。

```
#include <climits>
int d[N], pre[N], cur[N], s, t, n, m;
int augment(){
    int rc = INT_MAX;
    for(int v = u; v != s; v = E[pre[v] ^ 1].v)
    rc = std::min(rc, E[pre[v]].rc);
    for(int v = u; v != s; v = E[pre[v] ^ 1].v)
    E[pre[v]].rc -= rc, E[pre[v] ^ 1].rc += rc;
    return rc;
}
void relabel(int &u){
    int min_d = n - 1;
    for(int i = head[u]; i != -1; i = E[i].next)
    if(E[i].rc > 0)
    min_d = std::min(min_d, d[E[i].v]);
    d[u] = min_d + 1;
    cur[u] = head[u];
    if(u != s) u = E[pre[u] ^ 1].v;
}
using LL = long long;
LL sap(){
    LL ans = 0;
    for(int i = 0; i < n; i++)                    //节点标号从 0 开始
    d[i] = 0, cur[i] = head[i];
    for(int u = s; d[s] < n; ){
        for(int &i = cur[u]; i != -1; i = E[i].next)
        if(d[u] == d[E[i].v] + 1 && E[i].rc > 0)
        break;
        if(cur[u] != -1){
            pre[E[cur[u]].v] = cur[u], u = E[cur[u]].v;
            if(u == t) ans += augment, u = s;
        }
        else relabel(u);
    }
    return ans;
}
```

3．SAP 算法的正确性

不难验证：①重标号操作使一个点的距离标号严格增大；②增广和重标号这两种操作能维持距离标号的合法性。再结合性质 4.2，可以推出：①SAP 算法能在有限次重标号操作之后找到一条最短增广路径；②SAP 算法能在有限次重标号和增广之后求出一个最大流。

4．SAP 算法的时间复杂度

SAP 算法的时间复杂度可分成四部分考虑：检查点的出边是否为允许边；重标号；前进和后退的总次数；增广。

注意到：对点 u 重标号的前提是遍历 u 的出边而未找到允许边；对 u 重标号的时间复杂度即遍历 u 的出边的时间复杂度，所以重标号的总时间复杂度不超过检查点的出边是否为允许边的时间复杂度。

可以得出如下性质。

性质 4.5　若 SAP 算法对每个点重标号不超过 k 次，则“检查点的出边是否为允许边”和“计算新距离标号”的总的时间复杂度为

$$O\left(k\sum_{u\in V}|E(u)|\right)=O(km)$$

式中

$$E(u)=\{(v,w)\in E:v=u \quad 或 w=u\}$$

另外，每次对点 u 重标号之前已经找到了一条完全由允许边组成的从 s 到 u 的路径，从而有 $d(u)\leqslant d(s)<n$；又因为重标号之后 $d(u)$ 至少增加 1，所以 u 经历过至多 n 次重标号。因此①重标号的总的时间复杂度为 $O(n^2)$，后退的总时间复杂度也是 $O(n^2)$；②寻找允许边和计算新距离标号的总的时间复杂度为 $O(nm)$。

再考虑增广的总的时间复杂度。采用定理 4.2 的证明思路，可以证明：

定理 4.3　SAP 算法的增广次数为 $O(nm)$。

因为一次增广的时间复杂度为 $O(n)$，故增广的总的时间复杂度为 $O(n^2m)$，从而前进的总的时间复杂度为 $O(n^2m+n^2)=O(n^2m)$。综上所述，SAP 算法的时间复杂度为 $O(n^2m)$。

SAP 算法终止的条件是 $d(s)\geqslant n$，但实际中往往在 $d(s)\geqslant n$ 达成之前很久就已经算出一个最大流了，在最大流求出之后进行的种种操作显然是多余的。下面我们要介绍的优化能及时检测到当前残量网络上出现了一个最小割，亦即求得了一个最大流。

维护一个长为 n 的数组 num，num[i] 表示当前残量网络上距离标号为 i 的点的数目（$0\leqslant i<n$）。初始的合法距离标号不能任意设置，需要保证 num 数组中的非零项连续，亦即存在某个下标 l 使得 num[0],num[1],…,num[l] 都大于 0，数组中其他项都为 0。要满足这一条件至少有两个选择：①所有距离标号都置为 0；②从 t 点开始反向 BFS，求出准确距离标号。

每次将某点的距离标号从 k_1 提高到 k_2，num[k_1] 减 1，num[k_2] 加 1；若 num[k_1] 变为 0 则终止算法。

证明此优化的正确性。设 num[k_1] 减为 0 时求得的流为 f。令 $S=\{v\in V:d(v)>k_1\}$，$T=\overline{S}=\{v\in V:d(v)<k_1\}$，不难验证 $s\in S$、$t\in T$，即 (S,T) 是流网络 $G=(V,E)$ 的一个割。我们将证明 $|f|=c(S,T)$。考虑点对 $u\in S$ 和 $v\in T$，由 S 和 T 的定义可知 $d(u)>d(v)+1$，所以 $(u,v)\notin E_f$。由 $(u,v)\notin E_f$ 可推出：①若 $(u,v)\in E$ 则必有 $f(u,v)=c(u,v)$；②若 $(v,u)\in E$ 则必有 $f(v,u)=0$。因此我们有

$$\begin{aligned}f(S,T)&=\sum_{u\in S}\sum_{v\in T}f(u,v)-\sum_{v\in T}\sum_{u\in S}f(v,u)\\&=\sum_{u\in S}\sum_{v\in T}c(u,v)-\sum_{v\in T}\sum_{u\in S}0\\&=c(S,T)\end{aligned}$$

所以

$$|f|=f(S,T)=c(S,T)$$

下面给出 ISAP 算法的代码，采用逆向 BFS 求每个点的准确距离标号，其中 augment 函数与 SAP 算法中相同，故略去。

```
#include <climits>
int d[N], pre[N], cur[N], s, t, n, m;
int num[N], que[N];
bool bfs(){
    int beg = 0, end = 0;
    memset(d, -1, sizeof(int[n])); //nodes are 0-indexed
    d[t] = 0, que[end++] = t;
    for(; beg != end; ){
        int u = que[head++];
        for(int i = head[u]; i != -1; i = E[i].next)
        if(E[i^1].rc > 0 && d[E[i].v] == -1)
        d[E[i].v] = d[u] + 1, que[end++] = E[i].v;
    }
    return d[s] != -1;
}
void relabel(int &u){
    int min_d = n - 1;
    for(int i = head[u]; i != -1; i = E[i].next)
    if(E[i].rc > 0)
    min_d = std::min(min_d, d[E[i].v]);
    ++num[d[u] = min_d + 1];
    cur[u] = head[u];
```

```
        if(u != s) u = E[pre[u] ^ 1].v;
    }
    using LL = long long;
    LL sap(){
        LL ans = 0;
        if(bfs()){
            for(int i = 0; i < n; i++) cur[i] = head[i];
            for(int u = s; ; ){
                for(int &i = cur[u]; i != -1; i = E[i].next)
                if(d[u] == d[E[i].v] + 1 && E[i].rc > 0)
                break;
                if(cur[u] != -1){
                    pre[E[cur[u]].v] = cur[u], u = E[cur[u]].v;
                    if(u == t) ans += augment(), u = s;
                }
                else if(--num[d[u]]) relabel();
                else break;
            }
        }
        return ans;
    }
```

4.1.6　网络流的建图

网络流问题包括最大流的费用流两大类，这一小节只考虑最大流的建图（或称建模）。最大流的建图一般从流和割两个角度考虑。

1．从流的角度建图

从流的角度考虑，一般的思路是：将某种操作看成从源点经若干个中间点推送一定数量的流到汇点，这些流量经过边的容量表示对这种操作做出的种种限制。下面给出一个例子。

例 1（Collector's Problem，UVA 10779）　Bob 和朋友们在收集糖果中的贴纸。为了让自己手中的贴纸种类尽量多，他们决定用一张重复的贴纸去和别人交换一张自己所没有的贴纸。Bob 意识到只跟别人交换自己所没有的贴纸并不总是最优的，在某些情况下，换来一张已有的贴纸更划算。

假设 Bob 的朋友们只跟 Bob 交换贴纸（他们之间不交换），并且这些朋友只拿自己重复的贴纸去跟 Bob 交换自己所没有的贴纸。试问 Bob 最多能获得多少种贴纸？

建图方式：用点 a_i 表示第 i 种贴纸，设 Bob 有 c_i 张这种贴纸，从源点 s 向 a_i 连一条容量为 c_i 的边，从 a_i 向汇点 t 连一条容量为1的边。

用点 b_j 表示 Bob 的第 j 个朋友。若朋友 j 没有第 i 种贴纸，就从 a_i 向 b_j 连一条容量为 1

的边；若朋友 j 有 k（$k \geqslant 2$）张第 i 种贴纸，就从 b_j 向 a_i 连一条容量为 $k-1$ 的边。不难证明，最大流的值便是 Bob 所能获得的贴纸种类的最大值。

2．从割的角度建图

通常的思路是用有向边表示一个二元关系，要在满足一组二元关系的条件下求解一个最优化问题。下面以有向图的最大权闭包（Maximum Weight Closure）问题为例，介绍最小割建模，如图 4.2 所示，$\{2,5\}$、$\{1,2,4,5\}$、$\{3,4,5\}$ 都是闭包。

例 2（最大权闭包） 设 $G=(V,E)$ 是一个有向图，若 $V_1 \subseteq V$ 满足：$\forall (u,v) \in E,\ u \in V_1 \Rightarrow v \in V_1$，则称 V_1 是 G 的一个闭包，图 4.2 给出了一个例子。给图 G 每个节点 u 赋一个权值 $w(u) \in \mathbb{Z}$，点集 $X \subseteq V$ 的权值 $w(X)$ 定义为 $w(X)=\sum_{v \in X} w(v)$；最大权闭包问题就是求 G 的一个最大权闭包。

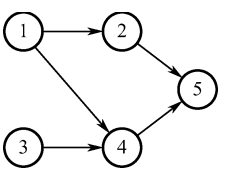


图 4.2 有向图的最大权闭包

可以将最大权闭包问题转化成最小割问题。引入源点 s 和汇点 t，从 s 向每个权值为正的点 u 连一条容量为 $w(u)$ 的边，从每个权值为负的点 u 向 t 连一条容量为 $-w(u)$ 的边，将原图中的边的容量设为 ∞（这里，∞ 是指一个大于 $\sum_{v \in V} |w(v)|$ 的值），将所得流网络记为 $G'=(V',E')$。

设 (S,T) 是网络 G' 的一个割，若 (S,T) 的任意前向边 (u,v) 都满足 $u \in S$ 或 $v \in T$，则称 (S,T) 为简单割（Simple Cut），如图 4.3 所示。不难证明，有向图 G 的闭包和网络 G' 的简单割一一对应。读者可自行验证：①若 V_1 是 G 的一个闭包，令 $S=\{s\} \cup V_1$，则 $(S,\overline{S})$ 是 G' 的一个简单割；②若 $(S,\overline{S})$ 是 G' 的一个简单割，令 $V_1=S-s$，则 V_1 是 G 的一个闭包。

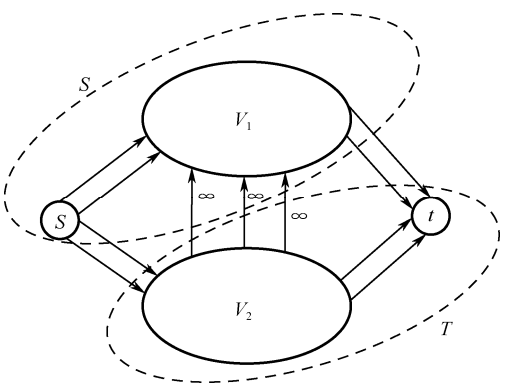


图 4.3 简单割

考虑简单割 $(S,\overline{S})$ 的容量 $c(S,\overline{S})$ 和 G 的闭包 $V_1=S-s$ 的权值 $w(V_1)$ 之间的关系。如图 4.3

所示，设V_1中权值为正的点的集合为V_1^+，权值为负的点的集合为V_1^-，令$V_2=V-V_1$，类似地定义V_2^+、V_2^-，有

$$\begin{aligned} w(V_1) &= w(V_1^+) + w(V_1^-) \\ c(S,T) &= \sum_{u\in V_2^+} c(s,u) + \sum_{u\in V_1^-} c(u,t) \\ &= w(V_2^+) - w(V_1^-) \end{aligned}$$

从而

$$w(V_1) + c(S,T) = w(V_1^+) + w(V_2^+) = w(V^+) = \text{const}$$

式中，V^+表示V中权值为正的点的集合；$w(V_1)$最大即$c(S,T)$最小，又因为图G中的边的容量都为∞，故G'的最小割一定是简单割。这样我们就把最大权闭包问题转化成了最小割问题。

4.1.7　例题讲解

例 4-1　奶牛进餐

Time Limit: 2000/2000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

奶牛很挑食，它们只会吃自己喜欢的某些食物和饮料。

约翰为他的奶牛做好了美味的饭菜，但他忘记了根据奶牛的喜好来制定菜单。虽然他可能无法满足每头奶牛，但他想要尽可能多地让奶牛吃完所有的食物和饮料。

约翰煮了 F（$1\leqslant F\leqslant 100$）种食物，并准备了 D（$1\leqslant D\leqslant 100$）种饮料。他必须为每头奶牛分配一种食物和一种饮料，当且仅当分配到的食物和饮料都是奶牛喜欢的，它才会去进食。（每种食物或饮料只能分配给一头奶牛）。请帮约翰计算出最多有多少头奶牛可以进食。

输入：

第一行：三个整数 N、F、D。

接下来 N 行：每行前两个数为 F_i 和 D_i，接下来的 F_i 个数代表奶牛喜欢的食物种类，D_i 个数代表奶牛喜欢的饮料的种类。

输出：

最多有多少头奶牛可以进食。

样例输入：

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

样例输出：

3

题目来源：POJ 3281。

解题思路：

若只有食物或者饮料的话，这道题很明显就是求二分图最大匹配数的问题。但由于同时存在饮料和食物，那么我们采用最大流的方法，建图方式为源点→食物→奶牛→奶牛→饮料→汇点，共六层网络。食物→奶牛、奶牛→饮料满足了进食关系，对于奶牛进行拆点，连接一条边权为 1 的边满足了每头奶牛只能拥有一种食物和一种饮料。

题目实现：

```
//#include<bits/stdc++.h>
#include<iostream>
#include<cstdio>
#include<cstring>
#include<vector>
#include<queue>

#define FAST ios::sync_with_stdio(false)
#define inf ((int)0x3f3f3f3f)
#define ll long long
#define Mem(a,b) memset(a,b,sizeof(a))
#define FREI freopen("in.txt","r",stdin)
#define FREO freopen("out.txt","w",stdout)
using namespace std;
const int maxn=1e5+100;

const int NodeMAXN=5e2+10;
const int EdgeMAXN=1e5+10;
struct Edge {
    int v,flowid;
};
int flowcnt;
int src,dest,dis[NodeMAXN];
vector<Edge>G[NodeMAXN];
vector<int>capacity;

void inline add(int u,int v,int cap) {
    G[u].push_back({v,flowcnt++});
    capacity.push_back(cap);            //正向容量
    G[v].push_back({u,flowcnt++});
```

```
31          capacity.push_back(0);                  //反向容量
32      }
33      bool bfs() {
34          Mem(dis,-1);
35          queue<int>que;
36          que.push(src);
37          dis[src]=0;
38          while(!que.empty()) {
39              int u=que.front();
40              que.pop();
41              for(int i=0; i<G[u].size(); i++) {
42                  int v=G[u][i].v,flowid=G[u][i].flowid;
43                  if(dis[v]!=-1||capacity[flowid]<=0)
44                      continue;
45                  dis[v]=dis[u]+1;
46                  que.push(v);
47              }
48          }
49          return dis[dest]>=0;
50      }
51      int dfs(int u,int cap) {
52          int ans=0;
53          if(u==dest)
54              return cap;
55          for(int i=0; i<G[u].size(); i++) {
56              int v=G[u][i].v,flowid=G[u][i].flowid;
57              if(dis[v]==dis[u]+1&&capacity[flowid]>0) {
58                  int tmp=dfs(v,min(cap,capacity[flowid]));
59                  capacity[flowid]-=tmp;
60                  capacity[flowid^1]+=tmp;
61                  cap-=tmp;
62                  ans+=tmp;
63              }
64          }
65          if(!ans)
66              dis[u]=-2;
67          return ans;
68      }
69      int maxflow() {
70          int ans=0;
71          while(bfs()) {
```

```
            ans+=dfs(src,inf);
        }
        return ans;
    }
    int N,F,D;
    int main() {
        //  FREI;
        while(scanf("%d%d%d",&N,&F,&D)!=EOF) {
            src=0;
            dest=N*2+F+D+1;
            for(int i=0; i<=dest; i++)
                G[i].clear();
            capacity.clear();
            flowcnt=0;
            int f,d,tmp;
            for(int i=1;i<=N;i++){
                scanf("%d%d",&f,&d);
                for(int j=0;j<f;j++){
                    scanf("%d",&tmp);
                    add(tmp,F+i,1);
                }
                for(int j=0;j<d;j++){
                    scanf("%d",&tmp);
                    add(F+N+i,F+N*2+tmp,1);
    }
                add(F+i,F+N+i,1);
            }
        for(int i=1;i<=F;i++)add(0,i,1);
        for(int i=1;i<=D;i++)add(N*2+F+i,dest,1);
        int ans=maxflow();
        printf("%d\n",ans);
        }
        return 0;
    }
```

例 4-2　Firing

Time Limit: 5000/5000 ms (Java/Others)　　Memory Limit: 131072/131072 KB (Java/Others)

题目描述：

大裁员：公司官僚成风，盘根错节，办实事的“码农”没几个。老板决定大裁员，每开除一个人，同时要将其下属一并开除，如果该下属还有下属，照斩不误。给出每个人的贡献

值和从属关系，求最小裁员数及最大贡献值的和。

输入：

第一行两个整数 n、m，$0<n\leqslant 5000$，$0\leqslant m\leqslant 60000$。

输出：

一行中输出两个整数 a、b，分别代表达到最大利润时所裁员的最少人数和最大利润。

样例输入：

```
5 5
8
-9
-20
12
-10
1 2
2 5
1 4
3 4
4 5
```

样例输出：

```
2 2
```

题目来源：POJ 2987。

解题思路：

不难发现，题目要求的是最大权闭合图。闭合图是指图中每个点的后续都在图中。最大权闭合图是指点的权值之和最大的闭合图，最大权闭合图的求解方法为：

（1）从源点 src 到正权值的点连一条容量为点权值的边。

（2）从负权值的点到汇点 dest 连一条容量为点权值绝对值的边。

（3）将原来的边容量置为无穷大。

（4）求解最小割，最大权=正权值和−最小割权值。

（5）残余网络中点的个数为裁员个数。

题目实现：

```
#include<iostream>
#include<cstring>
#include<algorithm>
#include<cmath>
#include<cstdio>
#include<queue>
```

```
#define ll long long
using namespace std;
const int maxn=3e5+100;
const ll inf=1e16;
int n,m,tot,src,dest;
int point[maxn],v[maxn],next[maxn],last[maxn],mark[maxn];
int deep[maxn],num[maxn],cur[maxn];
ll remain[maxn];
void add(int x,int y,ll z) {
    tot++;
    next[tot]=point[x];
    point[x]=tot;
    v[tot]=y;
    remain[tot]=z;
    tot++;
    next[tot]=point[y];
    point[y]=tot;
    v[tot]=x;
    remain[tot]=0;
}
ll addflow(int s,int t) {
    int now=t;
    ll ans=inf;
    while (now!=s) {
        ans=min(ans,remain[last[now]]);
        now=v[last[now]^1];
    }
    now=t;
    while (now!=s) {
        remain[last[now]]-=ans;
        remain[last[now]^1]+=ans;
        now=v[last[now]^1];
    }
    return ans;
}
void bfs(int s,int t) {
    for (int i=s; i<=t; i++)
        deep[i]=t;
    queue<int> p;
    p.push(t);
    deep[t]=0;
```

```
48          while (!p.empty()) {
49              int now=p.front();
50              p.pop();
51              for (int i=point[now]; i!=-1; i=next[i])
52              if (deep[v[i]]==t&&remain[i^1])
53              deep[v[i]]=deep[now]+1,p.push(v[i]);
54          }
55      }
56      ll isap(int s,int t) {
57          int now=s;
58          ll ans=0;
59          bfs(s,t);
60          for (int i=s; i<=t; i++)
61              num[deep[i]]++;
62          for (int i=s; i<=t; i++)
63              cur[i]=point[i];
64          while (deep[s]<t) {
65              if (now==t) {
66                  ans+=addflow(s,t);
67                  now=s;
68              }
69              bool pd=false;
70              for (int i=cur[now]; i!=-1; i=next[i])
71              if (deep[now]==deep[v[i]]+1&&remain[i]) {
72                  last[v[i]]=i;
73                  cur[now]=i;
74                  pd=true;
75                  now=v[i];
76                  break;
77                  }
78              if (!pd) {
79                  int minn=t;
80                  for (int i=point[now]; i!=-1; i=next[i])
81                  if (remain[i])
82                  minn=min(minn,deep[v[i]]);
83                  if (!--num[deep[now]])
84                  break;
85                  num[deep[now]=minn+1]++;
86                  cur[now]=point[now];
87                  if (now!=s)
88                  now=v[last[now]^1];
```

```
89                  }
90          }
91          return ans;
92  }
93  void dfs(int x) {
94          mark[x]=1;
95          for(int i=point[x]; i!=-1; i=next[i])
96          if (remain[i]&&!mark[v[i]])
97          dfs(v[i]);
98  }
99  int main() {
100         memset(point,-1,sizeof(point));
101         tot=-1;ll sum=0,num=0,x;
102         int u,v;
103         scanf("%d%d",&n,&m);
104         src=1;
105         dest=n+2;
106         ll k=6000;
107         for (int i=1; i<=n; i++){
108                 scanf("%lld",&x);
109                 if(x>0){
110                 add(src,i+1,x*k-1);
111                 sum+=x;
112                 num++;
113                 }
114                 else    add(i+1,dest,-x*k+1);
115         }
116         for (int i=1; i<=m; i++) {
117                 scanf("%d%d",&u,&v);
118                 add(u+1,v+1,inf);
119         }
120         ll ans=isap(src,dest);
121         printf("%lld %lld\n",(ans+num)%k,sum-(ans+num)/k);
122         return 0;
123 }
```

4.2　最小费用流

4.2.1　最小费用流算法

在最大流问题的基础上，本节讨论另一个网络流问题——最小费用流（Minimum Cost Flow）问题。在最小费用流问题中，边 $(u,v)\in E$ 除了有容量 $c(u,v)$，还有费用 $\text{cost}(u,v)\in\mathbb{N}$，表示单位流量流过 (u,v) 的花费。给定流 f，边 (u,v) 上产生的费用为 $\text{cost}(u,v)f(u,v)$，流 f 的费用 $\text{cost}(f)$ 定义为

$$\text{cost}(f)=\sum_{(u,v)\in E}\text{cost}(u,v)f(u,v)$$

最小费用流问题有三种常见的形式：

（1）给定费用流网络 $G=(V,E,s,t,c,\text{cost})$，求费用最小的最大流。

（2）给定费用流网络 $G=(V,E,s,t,c,\text{cost})$，求一个值为 x 且费用最小的流。

（3）最小费用流问题的一般形式。不再区别源点与汇点，每个点 $v\in V$ 都关联了一个值 $b(v)\in\mathbb{Z}$，$b(v)>0$ 代表供给（Supply），$b(v)<0$ 代表需求（Demand）。要求一个流 f 使其满足：边的容量限制、点的供需要求，以及费用最小。可形式化地写成

$$\min\sum_{(u,v)\in E}\text{cost}(u,v)f(u,v) \tag{4.3}$$

约束项为

$$\sum_{v:(u,v)\in E}f(u,v)-\sum_{v:(v,u)\in E}f(v,u)=b(u)\quad\forall u\in V \tag{4.4}$$

$$0\leqslant f(u,v)\leqslant c(u,v)\quad\forall(u,v)\in E \tag{4.5}$$

把满足式（4.4）和式（4.5）的流称为可行流。不难证明，这三种形式是等价的。下面只讨论最小费用最大流（Min-Cost Max-Flow）问题。

仍然采用增广路方法来求解。不难看出，若 $(u,v)\in E$ 且 $(u,v)\in E_f$，那么在残量网络 G_f 上，边 (v,u) 的费用 $\text{cost}(v,u)$ 应为 $-\text{cost}(u,v)$。容易想到一个贪心策略：把边的费用视为距离，在 G_f 上总是沿着最短增广路增广。这个想法是正确的，证明从略。此算法称为连续最短路算法（Successive Shortest Path Algorithm）。由于可能有负权边，不能用 Dijkstra 算法求最短路，而用 Bellman-Ford 算法。下面给出代码，图的表示与前面介绍的最大流算法相比，只多出费用 cost，add_edge 函数也要做相应的修改。

```
1    int n, m, s, t;
2    int d[N], pre[N]; // pre[u]: the edge to u in augmenting path
3    bool inq[N]; // inq[u]: is node u in queue?
4    using LL = long long;
```

```
LL flow, cost;
bool bellman_ford(){
    for(int i = 0; i < n; i++) d[i] = INT_MAX;
    memset(inq, 0, sizeof inq);
    d[s] = 0, inq[s] = true, a[s] = INF;
    std::queue<int> q;
    q.push(s);
    while(q.size()){
        int u = q.front(); q.pop();
        inq[u] = 0;
        for(int i = head[u]; i != -1; i= E[i].next){
            if(E[i].rc > 0 && d[E[i].v] > d[u] + E[i].cost){
                d[E[i].v] = d[u] + E[i].cost;
                pre[E[i].v] = i;
                if(!inq[E[i].v]) q.push(E[i].v), inq[E[i].v] = true;
            }
        }
    }
    if(d[t] == INT_MAX) return false;
    int rc = INT_MAX;
    for(int u = t; u != s; u = E[pre[u] ^ 1].v){
        rc = std::min(rc, E[pre[u]].rc);
    }
    for(int u = t; u != s; u = E[pre[u] ^ 1].v){
        E[pre[u]].rc -= rc;
        E[pre[u] ^ 1].rc += rc;
    }
    flow += rc, cost += LL(rc) * d[t];
    return true;
}

LL mcmf(){
    flow = cost = 0;
    while(bellman_ford());
}
```

4.2.2 例题讲解

例 4-3 回家

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

在地图上有 n 个人和 n 间房子。在每个单位时间内，每个人可以移动一步，到竖直方向或者水平方向的相邻格子，花费为 1。现在任务是每个人都移动到房子里，且每个房子只能容纳一个人。

地图上“H”代表房子，“m”代表人，“.”代表空地（注：每个空地都足够大，可以承受所有人，即所有人可以处于同一点）。

现在求解所有人都可以进入房子的最小费用。

输入：

多组数据。

第一行：两个整数 N 和 M。N 代表地图的行数，M 代表地图的列数，$0<N\leqslant 5000$，$0\leqslant m\leqslant 60000$。

接下来 N 行用于描述地图，每行有 M 个字符，其中“.”代表空地，“m”代表人，“H”代表房间。

输出：

最小的费用。

样例输入：

```
2 2
. m
H .
5 5
HH. .m
. . . . .
. . . . .
. . . . .
mm. .H
7 8
. . .H. . . .
. . .H. . . .
. . .H. . . .
mmmHmmmm
. . .H. . . .
. . .H. . . .
. . .H. . . .
0 0
```

样例输出：

3

题目来源：POJ 2195。

解题思路：

首先建出人和房间之间的联系，即每个人都建立到所有房间的有向边，其中 cost 为 abs(xi−xj)+abs(yi−yj)，边的容量 cap 为 1。为了满足每个人只能进入一个房间，每个房间只能有一个人住，即设立一个源点和汇点，由源点向所有人建立一条 cost=0、cap=1 的边，所有房子向汇点建立一条 cost=0、cap=1 的边，最后求解最小费用流即可。

题目实现：

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<vector>
#include<algorithm>
#include<queue>
#define Mem(a,b) memset(a,b,sizeof(a))
#define inf 0x3f3f3f3f
using namespace std;

const int MAXN=5e3+10;
const int EDGEMAXN=5e4+10;
int n,m,src,dest;
struct Edge {
    int v,cost,flowid;                          //cost 是费用，flowid 是流的 ID
};
struct Point{
    int x,y;
};
int capacity[EDGEMAXN<<1],flowcnt;
vector<Edge>G[MAXN];
pair<int,int> pre[MAXN];                //用 pair 存储路径，first 存储前趋节点，second 存储边的 ID
void add(int u,int v,int cost,int cap) {
    G[u].push_back({v,cost,flowcnt});
    capacity[flowcnt++]=cap;
    G[v].push_back({u,-cost,flowcnt});
    capacity[flowcnt++]=0;
}
bool inqueue[MAXN];                         //spfa 记录数组
int dis[MAXN];                              //距离
```

```
bool spfa() {
    Mem(inqueue,0);
    inqueue[src]=1;
    Mem(dis,inf);
    queue<int>q;
    q.push(src);
    dis[src]=0;
    while(!q.empty()) {
        int u=q.front();
        inqueue[u]=0;
        for(int i=0; i<G[u].size(); i++) {
            int v=G[u][i].v,flowid=G[u][i].flowid,cost=G[u][i].cost;
            if(dis[v]>dis[u]+cost&&capacity[flowid]>0) {
                dis[v]=dis[u]+cost;
                pre[v]= {u,flowid};
                if(!inqueue[v]) {
                    inqueue[v]=1;
                    q.push(v);
                }
            }
        }
    }
    if(dis[dest]<inf)
        return true;
    return false;
}
int updata() {
    int tmp=dest;
    int res=inf;
    while(tmp!=src) {
        int u=pre[tmp].first,flowid=pre[tmp].second;
        res=min(res,capacity[flowid]);
        tmp=u;
    }

    tmp=dest;
    while(tmp!=src) {
        int u=pre[tmp].first,flowid=pre[tmp].second;
        capacity[flowid]-=res;
        capacity[flowid^1]+=res;
        tmp=u;
```

```
72          }
73          return dis[dest];
74      }
75      int mcmf() {
76          int res=0;
77          while(spfa())
78          res+=updata();
79          return res;
80      }
81      Point people[105],house[105];
82      char mp[105];
83      int main(){
84          int n,m;
85          while(scanf("%d%d",&n,&m)!=EOF&&n&&m){
86              int p=0,h=0;
87              for(int i=0;i<n;i++){
88                  scanf("%s",mp);
89                  for(int j=0;j<m;j++){
90                      if(mp[j]=='m'){
91                          people[p].x=i;
92                          people[p++].y=j;
93                      }
94                      else if(mp[j]=='H'){
95                          house[h].x=i;
96                          house[h++].y=j;
97                      }
98                  }
99              }
100         for(int i=0;i<=p+h+1;i++)G[i].clear();
101             for(int i=0;i<p;i++){
102                 for(int j=0;j<h;j++){
103 add(i+1,p+j+1,abs(people[i].x-house[j].x)+abs(people[i].y-house[j].y),1);
104                 }
105             }
106             src=0;
107             dest=p+h+1;
108             for(int i=0;i<p;i++){
109                 add(src,i+1,0,1);
110             }
111             for(int i=0;i<h;i++){
112                 add(p+i+1,dest,0,1);
```

```
            }
            int ans=mcmf();
            printf("%d\n",ans);
        }
        return 0;
    }
```

例 4-4　区间

Time Limit: 5000/5000 ms (Java/Others)　　Memory Limit: 65536/65536 KB (Java/Others)

题目描述：

有 N 个带权的开区间，第 i 个区间覆盖了(a_i,b_i)，且权重为 w_i。想要挑选一些区间，并且最大化总权重。需要满足的条件是，任何点不可以被覆盖超过 K 次。

输入：

第一行输入数据组数。

每组数据第一行输入 N 和 K，$1 \leqslant K \leqslant N \leqslant 200$。

接下来 N 行包含三个整数：a_i、b_i、w_i，$1 \leqslant a_i < b_i \leqslant 100000$，$1 \leqslant w_i \leqslant 100000$。

注：每组数据之前包含一个空白行。

输出：

每组数据的最大总权值。

样例输入：

```
4

3 1
1 2 2
2 3 4
3 4 8

3 1
1 3 2
2 3 4
3 4 8

3 1
1 100000 100000
1 2 3
100 200 300
```

3 2
1 100000 100000
1 150 301
100 200 300

样例输出：

14
12
100000
100301

题目来源：POJ 3680。

解题思路：

离散化所有区间的端点，把每个端点看成一个顶点，建立源点 *S* 和汇点 *T*。

（1）从 *S* 到顶点 1（最左边顶点）连接一条容量为 *K*、费用为 0 的有向边。

（2）从顶点 2*N*（最右边顶点）到 *T* 连接一条容量为 *K*、费用为 0 的有向边。

（3）从顶点 *i* 到顶点 *i*+1（*i*+1<=2*N*）连接一条容量为无穷大、费用为 0 的有向边。

（4）对于每个区间[*a*,*b*]，从 *a* 对应的顶点 *i* 到 *b* 对应的顶点 *j* 连接一条容量为 1、费用为区间长度的有向边。

求最大费用最大流（最大费用流值就是最长 *k* 可重区间集的长度）。

题目实现：

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<vector>
#include<algorithm>
#include<queue>
#define Mem(a,b) memset(a,b,sizeof(a))
#define inf 0x3f3f3f3f
using namespace std;

const int MAXN=5e3+10;
const int EDGEMAXN=5e4+10;
int n,m,src,dest;
struct Edge {
    int v,cost,flowid;                    //cost 是费用，flowid 是流的 ID
};
struct Point{
```

```
18          int x,y;
19      };
20      int capacity[EDGEMAXN<<1],flowcnt;
21      vector<Edge>G[MAXN];
22      pair<int,int> pre[MAXN];                    //用 pair 存储路径，first 存储前趋节点，second 存储边的 ID
23      void add(int u,int v,int cost,int cap) {
24          G[u].push_back({v,cost,flowcnt});
25          capacity[flowcnt++]=cap;
26          G[v].push_back({u,-cost,flowcnt});
27          capacity[flowcnt++]=0;
28      }
29      bool inqueue[MAXN];                         //spfa 记录数组
30      int dis[MAXN];                              //距离
31      bool spfa() {
32          Mem(inqueue,0);
33          inqueue[src]=1;
34          Mem(dis,inf);
35          queue<int>q;
36          q.push(src);
37          dis[src]=0;
38          while(!q.empty()) {
39              int u=q.front();
40              q.pop();
41              inqueue[u]=0;
42              for(int i=0; i<G[u].size(); i++) {
43                  int v=G[u][i].v,flowid=G[u][i].flowid,cost=G[u][i].cost;
44                  if(dis[v]>dis[u]+cost&&capacity[flowid]>0) {
45                      dis[v]=dis[u]+cost;
46                      pre[v]= {u,flowid};
47                      if(!inqueue[v]) {
48                          inqueue[v]=1;
49                          q.push(v);
50                      }
51                  }
52              }
53          }
54          if(dis[dest]<inf)
55              return true;
56          return false;
57      }
58      int updata() {
```

```
59          int tmp=dest;
60          int res=inf;
61          while(tmp!=src) {
62                  int u=pre[tmp].first,flowid=pre[tmp].second;
63                  res=min(res,capacity[flowid]);
64                  tmp=u;
65          }
66
67          tmp=dest;
68          while(tmp!=src) {
69                  int u=pre[tmp].first,flowid=pre[tmp].second;
70                  capacity[flowid]-=res;
71                  capacity[flowid^1]+=res;
72                  tmp=u;
73          }
74          return dis[dest];
75  }
76  int mcmf() {
77          int res=0;
78          while(spfa())
79                  res+=updata();
80        return res;
81  }
82  int x[205],y[205],w[205];
83  int val[205];
84  int t;
85  int point[405];
86  int main(){
87          //freopen("in.txt","r",stdin);
88          scanf("%d",&t);
89          int cnt=0;
90          while(t--){
91                  int n,k;
92                  scanf("%d%d",&n,&k);
93                  cnt=0;
94                  for(int i=0;i<n;i++){
95                          scanf("%d%d%d",&x[i],&y[i],&val[i]);
96                          point[cnt++]=x[i];
97                          point[cnt++]=y[i];
98                  }
99                  sort(point,point+cnt);
```

```
100             int ssize=unique(point,point+cnt)-point;
101             src=0;
102             dest=ssize+2;
103             for(int i=0;i<=ssize;i++)G[i].clear();
104             add(src,1,0,k);
105             add(ssize+1,dest,0,k);
106             for(int i=1;i<=ssize;i++)add(i,i+1,0,inf);
107             for(int i=0;i<n;i++){
108                 int u=lower_bound(point,point+ssize,x[i])-point+1;
109                 int v=lower_bound(point,point+ssize,y[i])-point+1;
110                 add(u,v,-val[i],1);
111             }
112             printf("%d\n",-mcmf());
113         }
114         return 0;
115     }
```

4.3　二分图匹配

4.3.1　二分图的定义

二分图是一类特殊的无向图，其点集 V 可划分为两个不相交的子集 L 和 R，使得 E 中每条边的端点分别属于 L、R。形式化地说，无向图 $G=(V,E)$ 是二分图，当且仅当存在点集 V 的二划分（Bipartition）$\{L,R\}$，使得 $E\subseteq L\times R$。二分图也记为 $G=(L,R;E)$，如图 4.4 所示。

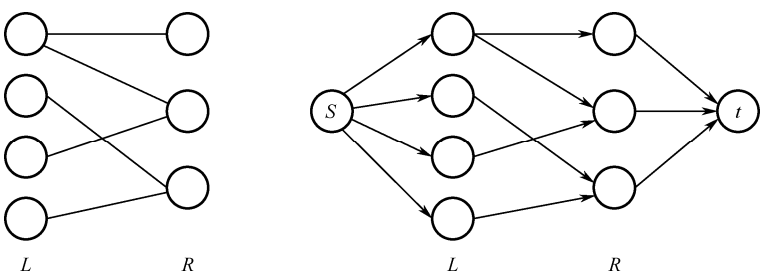


图 4.4　二分图和对应的流网络

4.3.2　二分图的最大匹配

设 $G=(V,E)$ 是一个无向图，若边集 $M\subseteq E$ 满足：对任意节点 $v\in V$，M 中至多有一条边以 v 为端点，则称 M 是 G 上的一个匹配（Matching）。若 $v\in V$ 是 M 中某条边的端点，则称 v

是 M 上的匹配点，否则称为非匹配点。M 中的边称为匹配边，$E-M$ 中的边称为非匹配边。最大匹配即边数最大的一个匹配。若图 G 的所有顶点都是匹配点，则称 M 为完美匹配（Perfect Matching）。显然，完美匹配存在的前提是 $|V|$ 为偶数。

二分图的最大匹配问题可以转化为最大流问题。如图 4.4 所示，给定二分图 $G(V,E)$，$V=L\cup R$ 可以构造一个流网络 $G'=(V',E')$。

$$V'=V\cup\{s,t\},$$

$$E'=\{(s,u):u\in L\}\cup\{(u,v):(u,v)\in E\}\cup\{(v,t):v\in R\}$$

将 E' 中每条边的容量置为 1。注意：E 中的边是无向边，而 E' 中的边是有向边。

不难证明，二分图 $G=(V,E)$ 的最大匹配数等于对应的流网络 $G'=(V',E')$ 上的最大流的值。下面来分析此算法的时间复杂度，已经知道朴素实现的增广路方法的复杂度不超过 $O(|f^*|E)$，其中 $|f^*|$ 表示最大流的值。在二分图的最大匹配中，我们有：①$|f^*|\leqslant \min(L,R)=O(V)$；②没有边与之相连的点不用考虑，故 $|V|\leqslant|E|$，因此 $|E'|=|E|+|V|\leqslant 3|E|$。故而可以在 $O(VE')=O(VE)$ 的时间内求出二分图 $G=(V,E)$ 的最大匹配。

1．匈牙利算法

匈牙利算法（Hungarian Algorithm）也称为 Kuhn-Munkres 算法（简称 KM 算法），可用于求解二分图最大权匹配（Maximum Weighted Bipartite Matching）问题。

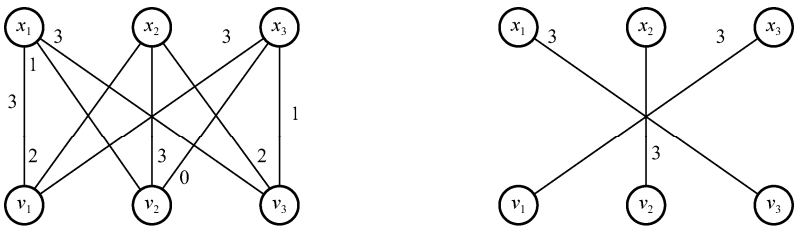
2．二分图的完美匹配

令 $G=(V,E)$ 是一个二分图，$\{A,B\}$ 是其（顶点的）二划分。不失一般性，设 $|A|\leqslant|B|$。设 M 为 G 的最大匹配，若 $|M|=|A|$，则称 M 为 G 上的完美匹配（Perfect Matching）。

3．带权二分图

给二分图 $G=(V,E)$ 的每条边 $(u,v)\in E$ 赋一个权值 $w(u,v)\in\mathbb{N}$，就得到了带权二分图（Weighted Bipartite Graph）。匹配 M 的权值定义为匹配边的权值之和，即 $w(M)=\sum_{(u,v)\in M}w(u,v)$。二分图最大权匹配问题（见图 4.5），即给定带权二分图 $G=(V,E)$，V 的二划分为 $\{X,Y\}$，求一个权值最大的匹配 M。

在二分图最大权匹配问题中，不失一般性，假设：①通过添加权值为 0 的边，可以假定图 $G=(V,E)$ 是完全二分图，即 $\forall u\in L,v\in R,(u,v)\in E$；②通过添加虚拟点，可以假定 $|X|=|Y|$，此时总可以求出一个权值最大的完美匹配。加上了这两个假定之后的二分图最大权匹配问题也称为二分图最大权完美匹配问题或者指派问题（Assignment Problem）。本节将在上述两假定下讨论二分图的最大权完美匹配问题。



（a）带权完全二分图　　（b）带权最大匹配

图 4.5　二分图最大权匹配问题

4．交错路和交错树

令 M 为二分图 $G=(V,E)$ 的一个匹配。设 P 为 G 上一条简单路径[①]，若 P 中的边交替地出现在 M 和 $E-M$ 中，则称 M 为交错路（Alternating Path）。若 P 的两端点都是非匹配点，则称 P 为增广路；不难验证：①增广路上的非匹配边比匹配边多一条；②将 P 上的匹配边和非匹配边对调，得到的边集 M' 仍是 G 上的一个匹配，且 $|M'|=|M|+1$；这正是称 P 为增广路的原因。增广操作可以用对称差（Symmetric Difference，$\oplus$）这一集合运算来表示，设 A、B 为两集合，对称差 $A\oplus B$ 定义为 $(A-B)\cup(B-A)$，即恰属于这两集合中某一个的元素的集合，有 $M'=M\oplus P$。

由 G 和 M 所导出的交错树（Alternating Tree）是一棵有根树，它满足：①以某个非匹配点为根；②从根到树中每个点的路径都是交错路。

5．顶点标号和相等图

设 $G=(V,E)$ 是一个二分图，V 划分为 $\{X,Y\}$，顶点标号是一个函数 $l:V\to\mathbb{R}$，称 l 为可行标号，若它满足

$$l(x)+l(y)\geqslant(x,y),\forall x\in X,y\in Y$$

则相等图（Equity Graph）$G_l=(V,E_l)$ 是由 G 和 l 所导出的一个图，即

$$E_1=\{(x,y)\in E:l(x)+l(y)=w(x,y)\}$$

定理 4.4　若 l 是一个可行标号函数且 $M\subseteq E_1$ 是一个完美匹配，则 M 是一个最大权匹配。

下面给出匈牙利算法的伪代码。

```
Hungarian(G, w)
1.  令 l 为某一可行标号， M 为 G_l 上的某个匹配
2.  while  M 不是完美匹配
3.      if  G_l 中不存在 M 的增广路
4.          计算新标号函数 l' 使得 M ⊂ E_l' 且 G_l' 上存在 M 的增广路
```

① 这里我们将路径视为边的集合，故用大写字母 P 表示。

```
5.            l ← l′
6.        在 G_l 上找一条增广路 P
7.        M ← M ⊕ P
8.    return M
```

初始的可行标号可置为

$$\forall y \in Y, \qquad l(y)=0; \qquad \forall x \in X, \qquad l(x)=\max_{y \in Y}\{w(x,y)\}$$

匈牙利算法的核心是上述伪代码的第 4 行，即优化可行标号使得有新边进入相等图且出现增广路。下面讲述如何优化可行标号，为此，先介绍邻集这一概念。

令 l 是某个可行标号，点 $u \in V$ 在 G_l 上的邻集（Neighbor）定义为 $N_l(u)=\{v \in V:(u,v) \in E_l\}$，类似地，点集 $S \subseteq V$ 的邻集定义为 $N_l(S)=\bigcup_{u \in S} N_l(u)$。

定理 4.5 设 $S \subseteq X$ 满足 $T=N_l(S) \neq Y$。令 $\alpha_l=\min_{x \in S, y \notin T}\{l(x)+l(y)-w(x,y)\}$，又令

$$l'(v)=\begin{cases} l(v)-a_l, & v \in S \\ l(v)+a_l, & v \in T \\ l(v), & \text{其他情况} \end{cases}$$

则 l' 是一个可行标号且满足

（1）对于 $x \in S, y \in T$，若 $(x,y) \in E_l$，则 $(x,y) \in E_{l'}$。

（2）对于 $x \notin S, y \notin T$，若 $(x,y) \in E_l$，则 $(x,y) \in E_{l'}$。

（3）存在边 $(x,y) \in E_{l'}$ 满足 $x \in S, y \notin T$。

此引理的正确性几乎是显然的，读者可自行验证。下面来考虑怎样选择 S 可以使得按定理 4.5 得到 $E_{l'}$ 满足 $M \subset E_l$ 且 $E_{l'}$ 中包含 M 的增广路。为了便于表述，定义一个记号，设 $S \subseteq V$，S 的匹配集 $\text{match}(S)$ 定义为：

$$\text{match}(S)=\{t \in V: \text{存在} s \in S \text{使得} (s,t) \in M\}$$

利用定理 4.5 中的 l' 所满足的三条性质，S 首先要满足 $N_l(S) \neq Y$，这样才能利用性质（3）；因此，①令 $S_0=\{u\}$，u 是 X 中任意一个未匹配点，由于 M 不是完美匹配，u 总是存在的；②从而 $N_l(S_0)$ 中都是匹配点，为了保证 $N_l(S_0)$ 覆盖的匹配边仍在 $E_{l'}$ 中，根据性质（1），再令 $S_l=S_0 \cup \text{match}(N_l(S_0))$，显然有 $S_l \subseteq \{X\}$；如此反复，直到 $\text{match}(N_l(S_k)) \subset S_k$，此 S_k 即所求的 S。不难看出，上述过程是一个不断扩展以 u 为根的交错树的过程。由于 G_l 中不存在增广路，所以对任意 $1 \leqslant i \leqslant k$，$N_l(S_i)$ 中全是匹配点，从而有 $N_l(S_i) \neq Y$，因此 S_k 即所要求的 S。

由上述分析可知 $N_l(S) \subset N_{l'}(S)$，若 $N_{l'}(S)-N_l(S)$ 中不含未匹配点，则继续更新 S；随着 $N_l(S)$ 中点不断增多，迟早会在 $N_{l'}(S)-N_l(S)$ 中遇到未匹配点，此时便得到了一条增广路。

从扩展交错树的角度，将上面给出的匈牙利算法的伪代码的第 3 行到第 7 行加以细化。

```
Augment(G,l)
1.    在 X 中取一未匹配点 u
```

```
2.    S ← {u}
3.    T ← ∅
4.    repeat
5.        if N_l(S) == T
6.            按照定理 4.5 计算 l'
7.            l ← l'
8.        for y∈ N_l(S) − T
9.            if y 是未匹配点
10.               P ← 交错树中的从 u 到 y 的路径
11.               M ← M ⊕ P
12.               return
13.           else
14.               {z} ← match({y})
15.               S ← S ∪ {z}
16.               T ← T ∪ {y}
```

S 和 T 分别是当前交错树的点集与 X 和 Y 的交集。

6．匈牙利算法的时间复杂度

令 $n=|X|=|Y|$，称上述 Augment 过程运行一次为一个阶段（Phase）；每个阶段过后 $|M|$ 都增加1，因此至多有 n 个阶段。下面考虑一个阶段的时间复杂度。

在具体分析之前，先介绍 Augment 过程的实现细节。要点如下：①对于每个 $y\notin T$，维护一个值 $\text{slack}(y)=\min\_(x\in S)\{l(x)+l(y)-w(x,y)\}$；②维护可行标号 l，集合 S、T。显然，维护集合 S、T 的时间复杂度为 $O(n)$。更新一次可行标号的时间复杂度为 $O(n)$，在一个阶段内更新可行标号的次数不超过 n，因为每次更新 l 之后若仍未找到增广路，则 S 中至少增加一个点。维护 $\text{slack}(y)$ 的时间复杂度包括初始化所有 slack 值的时间复杂度 $O(n)$。

第 15 行，每当将点 z 从 $\overline{S}$ 移动到 S 中时，需要更新所有 slack 值，这可在 $O(n)$ 内完成，在一个阶段内最多有 n 个点会被加到 S 中去，所以总的时间复杂度为 $O(n^2)$。

第 6 行，更新 l' 需要先计算 $\alpha_1=\min\limits_{y\in T}\text{slack}(y)$。这可在 $O(n)$ 内完成。计算 l'（更新 l）的时间复杂度也是 $O(n)$ 的，又一个阶段内更新 l 不超过 n 次，故总的时间复杂度为 $O(n^2)$。

综上可得，匈牙利算法的总的时间复杂度为 $O(n^3)$。下面给出匈牙利算法的代码。

```
1    int w[N][N], Lx[N], Ly[N], slack[N], match[N];
2    bool S[N], T[N];
3    int n;
4    bool dfs(int u){
5        S[u]=true;
6        for(int v=1; v<=n; v++){
```

```
            if(T[v]) continue;
            int tmp=Lx[u]+Ly[v]-w[u][v];
            if(tmp==0){
                T[v]=true;
                if(!match[v] || dfs(match[v])){
                    match[v]=u; return true;
                }
            }
            else slack[v] = min(slack[v], tmp);
        }
        return false;
    }
    int KM(){
        memset(match, 0, sizeof match);
        memset(Lx, 0x3f, sizeof Lx);
        memset(Ly, 0x3f, sizeof Ly);
        for(int i = 1; i <= n; i++){        //phase
            for(int j = 1; j <= n; j++)
            slack[j]=INT_MAX;
            for(; ;){
                memset(S, 0, sizeof S);
                memset(T, 0, sizeof T);
                if(dfs（i）) break; int a = INT_MAX;
                for(int j = 1; j <= n; j++) if(!T[j]) a=min(a, slack[j]);
                for(int j = 1; j <= n; j++){
                    if(S[j]) Lx[j] -= a;
                    if(T[j]) Ly[j] += a;
                    else slack[j] -= a;
                }
            }
        }
        int ans = 0;
        for(int i = 1; i <= n; i++) ans += w[match[i]][i];
        return ans;
    }
```

4.3.3 二分图的性质与应用

1．二分图的最小点覆盖

给定无向图 $G=(V,E)$，若点集 $X \subseteq V$ 满足：E 中任意一条边至少有一个端点在 X 中，

则称 X 为 G 的点覆盖。点数最小的点覆盖 X^* 称为最小点覆盖，$|X^*|$ 称为最小点覆盖数。

定理 4.6（König，1931）　二分图的最小点覆盖数等于其最大匹配数。

证明从略。

2．最小路径覆盖

给定有向图 $G=(V,E)$，设 P 是图 G 上若干点不相交（Vertex-Disjoint）的简单路径的集合，若每个点 $v\in V$ 都存在于唯一一条 P 中的路径上，则称 P 是 G 的一个路径覆盖（Path Cover）。路径数目最少的路径覆盖称为最小路径覆盖。用 $\mathrm{MinPC}(G)$ 表示图 G 的最小路径覆盖数。

有向无环图（Directed Acylic Graph，DAG）的最小路径覆盖问题可转化为二分图的最大匹配问题。给定有向无环图 $G=(V,E)$，设 $V=\{1,2,\cdots,n\}$，将点 $i\in V$ 拆成 x_i、y_i 两个点，若 $(i,j)\in E$，则连一条无向边 (x_i,y_j)。这样得到了二分图 $G'=(V',E')$。

$$V'=\{x_1,x_2,\cdots,x_n\}\cup\{y_1,y_2,\cdots,y_n\}$$
$$E'=\{(x_i,y_j):(i,j)\in E\}$$

定理 4.7　设 M' 是 G' 的一个最大匹配，P^* 是 G 的一个最小路径覆盖，则有 $|P^*|=n-|M'|$。

证明：考虑从 $G_0=(V,\varnothing)$ 开始，往图中添边的过程。初始时，每个点 $v\in V$ 自成一条路径，有 $\mathrm{MinPC}(G_0)=n$。希望每次加入的那条边能将两条简单路径合为一条，所以这条边应起始于某条路径的终点，终止于另一条路径的起点，反复如此添边，直到无法操作为止。将添边过程中得到的图依次记为 $G_1,G_2\cdots$，问题就归结为最多能加入多少条这样的边。不难发现，G' 上的匹配 M 和上述加边方案是一一对应的：L 中的某个未匹配点表示某条路径的终点，R 中的某个未匹配点表示某条路径的起点，所以 $\mathrm{MinPC}(G_i)=n-i$，于是 $\mathrm{MinPC}(G)=n-|M'|$。

4.3.4　例题讲解

例 4-5　Muddy Field

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

有一块奶牛进食的矩形区域，有 R 行 C 列（$1\leqslant R\leqslant 50$，$1\leqslant C\leqslant 50$）。天刚刚下过雨，这有利于草的生长，然而奶牛很爱整洁，它们不愿意在吃草的时候弄脏自己的脚。

为了防止这种事情发生，牧场主将会在泥泞的地上摆放一些木板，每块木板的宽度为 1，长度任意，并且摆放的方式为严格平行于边界。

牧场主想要最小化木板的数量，并且木板不能覆盖草地，但可以重叠。

请计算木板的最小数量。

输入：

第一行输入 *R* 和 *C*。

接下来 *R* 行用于描述地图，每行有 *C* 个字符，其中“*”代表空地，“.”代表草地。

输出：

最小的木板数。

样例输入：

```
4 4
*.*.
.***
***.
..*.
```

样例输出：

```
4
```

题目来源：POJ 2226。

解题思路：

进行行列建图，将横着的木板作为二分图中一侧的点，竖着的木板作为另一侧，定义出二分图。对于每一个“*”的点，考虑横着的木板如何覆盖，竖着的木板如何覆盖，以及如何定义横着覆盖它的木板的编号。其实可以把每一个需要覆盖的顶点所在的泥地上的最左端的顶点作为横着木板的编号，所在泥地最上端的顶点作为竖着的，将最左端的顶点和最上端的顶点连边建立二分图。最后求最小的顶点覆盖，就是等于保证每个泥地都被横着的木板或者竖着的木板覆盖了。

题目实现：

```
1   #include<iostream>
2   #include<cstring>
3   #include<cstdio>
4   #include<vector>
5
6   using namespace std;
7
8   const int maxn=55;
9   const int maxv=5005;
10  vector<int> g[maxv];
11
12  int from[maxv],tot;
13  bool used[maxv];
14
```

```
bool match(int x) {
    for(int i=0; i<g[x].size(); i++) {
        int v=g[x][i];
        if(!used[v]) {
            used[v]=1;
            if(from[v]==-1||match(from[v])) {
                from[v]=x;
                return true;
            }
        }
    }
    return false;
}

int hungry() {
    tot=0;
    memset(from,-1,sizeof(from));
    for(int i=0; i<maxv; i++) {
        memset(used,0,sizeof(used));
        if(match(i))
        tot++;
    }
    return tot;
}
char mp[55][55];
int main() {
  //freopen("in.txt","r",stdin);
    int n,m;
    while(scanf("%d%d",&n,&m)!=EOF){
        for(int i=0;i<n;i++){
            scanf("%s",mp[i]);
        }
        for(int i=0;i<maxv;i++)g[i].clear();
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(mp[i][j]=='*'){
                    int y=i,x=j;
                    while(y>0&&mp[y-1][j]=='*')--y;
                    while(x>0&&mp[i][x-1]=='*')--x;
                    g[y * 50 + j].push_back(i * 50 + x + 2500);
                }
```

```
56                      }
57                  }
58                  int ans=hungry();
59                  printf("%d\n",ans);
60              }
61              return 0;
62      }
```

4.4 练习题

习题 4-1

题目来源：Codeforces 546E。

题目类型：最大流。

解题思路：有 n 座城市和 m 条双向道路，每条道路连接两个城市，每座城市驻扎着一支部队，第 i 座城市的部队有 a_i 名士兵。现在士兵需要进行转移，每个士兵可以待在原来所在的城市，也可以转移到某个相邻的城市去。判断在士兵转移之后是否有可能第 i 座城市恰好有 b_i 名士兵。如果可能，输出一种转移方案。在 $n \times n$ 的矩阵 $\boldsymbol{C}$ 中，c_{ij} 表示有多少名士兵从城市 i 转移到城市 j（$i \neq j$），c_{ii} 表示有多少名士兵待在城市 i。

添加一个源点 s，再添加 n 个点代表 n 座城市，从 s 向每个点连一条容量为 a_i 的弧；添加汇点 t，再添加 n 个点代表 n 座城市，从第 i 个点向 t 连一条容量为 b_i 的边；对于前 n 个点中的某个点 i 和后 n 个点中的某个点 j，若 $i = j$ 或存在边 (i, j)，则加入有向边 (i, j) 和 (j,i)，容量都是无穷大。求 s 到 t 的最大流，若最大流等于士兵的总数则存在满足条件的转移方案。其他细节留给读者考虑。

习题 4-2

题目来源：洛谷 1251。

题目类型：网络流。

解题思路：一个餐厅在相继的 N 天里，每天需用的餐巾数不尽相同。假设第 i 天需要 r_i 块餐巾（$i = 1,2,\cdots,N$）。餐厅可以购买 i 块新的餐巾，每块餐巾的费用为 p；或者把旧餐巾送到快洗部，洗一块需 m 天，其费用为 f；或者送到慢洗部，洗一块需 n 天（$n > m$），其费用为 s（$s < f$）。

每天结束时，餐厅必须决定将多少块脏的餐巾送到快洗部，多少块餐巾送到慢洗部，以及多少块保存起来延期送洗。但是每天洗好的餐巾和购买的新餐巾数之和，要满足当天的需求量。

试设计一个算法为餐厅合理地安排好 N 天中餐巾使用计划，使总的花费最小。编程设计一个最佳餐巾使用计划。

把餐巾看成“流”，把使用新餐巾和清洗脏餐巾看成“流”从一点流到另一点；自然地，购买或清洗一块餐巾的费用看成单位流量从边上流过的费用。这样就把问题转化成了一个最小费用最大流问题，建图的细节留给读者考虑。

习题 4-3

题目来源：HDU 6346。

题目类型：整数规划。

解题思路：给定 $n\times n$ 个整数 $a_{i,j}$（$1\leqslant i,j\leqslant n$），要找出 $2n$ 个整数 $x_1,x_2,\cdots,x_n;y_1,y_2,\cdots,y_n$ 在满足 $x_i+y_j\leqslant a_{i,j}$（$1\leqslant i,j\leqslant n$）的约束下，最大化目标函数 $\sum\limits_{1\leqslant i\leqslant n}x_i+\sum\limits_{1\leqslant i\leqslant n}y_i$。需要解决这个整数规划问题，并给出目标函数的最大值。

注意到约束条件与 KM 算法中顶标函数 l 满足的约束 $l(x)+l(y)\geqslant w(x,y)$ 很相似，在 KM 算法中，最大权匹配等于最小顶标和。现在要求最大顶标和，将上式两边乘 -1 即可得到想要的约束形式，从而可以将原问题转化为一个带权二分图的最大完美匹配问题。

习题 4-4

题目来源：NOI 2008。

题目类型：最小费用最大流。

解题思路：申奥成功后，布布经过不懈努力，终于成为奥组委下属公司人力资源部门的主管。布布刚上任就遇到了一个难题：为即将启动的奥运新项目招募一批短期志愿者，经过估算，这个项目需要 N 天才能完成，其中第 i 天至少需要 A_i 个人。

一共有 M 类志愿者可以招募，其中第 i 类可以从第 S_i 天工作到第 T_i 天，招募费用是每人 C_i。希望用尽量少的费用招募足够的志愿者。可将问题转化为最小费用最大流问题，将题目中的约束条件转化成流网络中节点上的流量平衡条件。

习题 4-5

题目来源：Codeforces 808F。

题目类型：网络流最小割。

解题思路：Vova 在玩一个卡牌收集游戏。Vova 的卡包中有 n 张卡，每张卡有 3 个属性：力量 p_i、魔法 c_i、等级 l_i。目前 Vova 的角色在游戏中的等级是 1。Vova 想组一套力量之和不小于 k 的卡组，一套卡组需要满足两个条件：①卡组中任意两张卡牌的法力值之和都不能是质数；②卡组中每张卡牌的等级都不能高于 Vova 的角色的等级。试问 Vova 的角色至少要达

到几级才能组出满足条件的卡组。首先卡组中魔法为 1 的卡至多有一张，在不考虑等级要求的情况下，只有两张卡牌的法力值奇偶性不同的情况下才可能引起冲突。

习题 4-6

题目来源：CodeChef Feb16。

题目类型：最大流。

解题思路：在满足下列条件的前提下，是否可以为每位员工指定工作计划。所有人的工作安排都是按小时计算的：每个人在一个工时内只能干一件事情，中途不能切换到别的事情上去，也不能同时干多件事情。一周内有 D 天是工作日，编号为1～D。编号为 i 的员工每周最多能与顾客通话 L_i 小时。

将每个人的时间安排的约束条件，用流网络中弧的容量来表示。由于开会的安排已知，设第 i 位员工在第 j 天开会的小时数为 $m_{i,j}$，第 i 位员工在第 j 天的午餐时段除了开会还剩 $f_{i,j}$ 小时，第 i 位员工在第 j 天除了午餐时段和开会的时间还剩 $c_{i,j}$ 小时。考虑第 i 位员工的约束条件：先考虑通话时间的限制，①每周通话的时间不超过 L_i 小时→从源点向点 w_i 连一条容量为 L_i 的弧；②每天花在开会或者与客户通电话上的时间不超过 N 小时→从 w_i 向 $d_{i,j}$ 连一条容量为 $N-m_{i,j}$ 的弧；③在午餐时段至少空闲一小时→从 $d_{i,j}$ 向 $l_{i,j}$ 连一条容量为 $f_{i,j}-1$ 的弧，从 $d_{i,j}$ 向 $p_{i,j}$ 连一条容量为 $c_{i,j}$ 的弧；④为每天的每个小时设立一个点 $h_{i,j}$，从 $l_{i,j}$ 和 $p_{i,j}$ 向对应的若干个小时节点各连一条容量为1的弧，再从 $h_{i,j}$ 向汇点连一条容量为 $R_{i,j}$ 的弧。求从源点到汇点的最大流，检查从 $h_{i,j}$ 到汇点的弧上的流是不是满的。

习题 4-7

题目来源：Codeforces 786E。

题目类型：二分图的顶点覆盖。

解题思路：ALT 是 Encore 星系中的一颗行星，ALT 上有 n 座城市，有 $n-1$ 条长度相等的双向道路连接这些城市，使得任意两座城市都可相互到达。ALT 上有 m 位居民，第 i 位居民住在城市 x_i，在另一座城市 y_i 工作。ALT 星上没有狗，居民们都希望放一些小狗到星球上。居民们的要求是或者他自己有一只小狗，或者从他居住的城市到他工作的城市的最路径的每条道路上都有一只小狗，试问至少要在 ALT 星上放多少只小狗才能满足每位居民的要求，并给出分配小狗的方案。

这个问题可转化成二分图的顶点覆盖问题。每位居民和每条道路都用一个顶点表示，对于第 i 位居民，若 x_i 到 y_i 的路径经过第 j 条边，则在对应的两顶点之间连一条边，但是这种方法的时间复杂度太高。正确的解答留给读者考虑。

第 5 章

经典算法问题

5.1 多项式与快速傅里叶变换

快速傅里叶变换是基于数学中离散傅里叶变换的一种算法，在工程中很有价值，被广泛应用在各种信号处理领域。本章节以快速多项式乘法为例，介绍快速傅里叶变换算法在 ACM 竞赛中的应用。

5.1.1 多项式

在数学中，多项式（Polynomial）是指由变量、系数，以及它们之间的加、减、乘、幂运算得到的表达式。一个化到最简的多项式应形如

$$A(x)=\sum_{i=0}^{n}a_i x^i$$

式中，a_i 称为多项式系数，复数域内的多项式要满足 $a_i \in C$ 。如果一个多项式的最高阶非零系数是 a_k，那么称这个多项式为 k 次多项式。

5.1.2 多项式的表示与多项式乘法

多项式最常见的表示方法为系数表示法，即对于 n 次多项式，使用多项式的系数向量 $\boldsymbol{a}=(a_0,a_1,\cdots,a_n)$ 来表示。系数表示法易于理解，并且对于一些运算来说，系数表示法十分方便。例如，多项式求固定点处的值（秦九韶算法），两个多项式求和，都可以在时间复杂度 $O(n)$ 内完成。

但是，在计算多项式乘法时，两多项式逐项相乘的时间复杂度为 $O(n^2)$，在 n 较大时不能满足要求。为了优化多项式乘法的时间复杂度，引入一种新的多项式表示方法——点值表示法。

一个 n 次多项式 A 可以由 n 个点值对组成的集合来表示，称为点值表示，即 $\{(x_0,y_0),(x_1,y_1),\cdots,(x_{n-1},y_{n-1})\}$ 。

对于任意的整数 $k=0,1,\cdots,n-1$，点值对满足 x_k 各不相同，且 $y_k=A(x_k)$。由点值对构成的范德蒙德矩阵的行列式值不为 0 可证得，一个 n 次多项式至少需要 n 个自变量不相同的点值对才可以被唯一表示。一个多项式可以有很多种不同的点值表示，这是因为 n 个点值对有很多种不同的选取方案。

研究点值表示方式下多项式的乘法，不难发现，如果多项式 A和B 的点值表示分别是

$$\{(x_0,A_0),(x_1,A_1),\cdots,(x_{n-1},A_{n-1})\}$$

与

$$\{(x_0,B_0),(x_1,B_1),\cdots,(x_{n-1},B_{n-1})\}$$

那么多项式 $C=AB$ 的点值表示应为

$$\{(x_0,A_0B_0),(x_1,A_1B_1),\cdots,x_{n-1},A_{n-1}B_{n-1}\}$$

不难发现，进行点值表示方式下的多项式乘法，将 n 个点值中的因变量两两相乘，就可以得到乘积多项式的点值表示，时间复杂度为 $O(n)$。可是如何将两个被乘的多项式从常见的系数表示转为点值表示，又如何将乘积多项式转回系数表示呢？

通过任意 n 个点值对直接确定多项式各个系数的值的方法是存在的，利用拉格朗日插值公式可以在时间复杂度 $O(n^2)$ 内得到多项式的系数表示，但这不能达到降低时间复杂度的目的，因此，需要巧妙地选择 n 个点值对的自变量值，这 n 个值即 n 次单位复数根。

n 次单位复数根指的是满足 $\omega^n=1$ 的所有复数 ω，n 次单位复数根刚好有 n 个，它们是 $e^{\frac{2k\pi}{n}i}$，其中 i 是复数单位，$k=0,1,2,\cdots,n-1$。n 次单位复数根记为 $\omega_n=(\omega_n^0,\omega_n^1,\cdots,\omega_n^{n-1})$，在复平面上这 n 个根均匀的分布在半径为 1 的圆上。

n 次单位根满足 3 个重要的引理，这里不加证明地给出：

1．消去引理

对于任何整数 $n\geqslant 0$，$k\geqslant 0$，$d\geqslant 0$，有

$$\omega_{dn}^{dk}=\omega_n^k$$

推论：对任意正偶数 n 有

$$\omega_n^{\frac{n}{2}}=\omega_2=-1$$

2．折半引理

如果 n 是正偶数，那么 n 个 n 次单位复数根的平方的集合，就是 $n/2$ 个 $n/2$ 次单位复数根的集合，即

$$\left(\omega_n^{k+\frac{n}{2}}\right)^2=\omega_n^{2k+n}=\omega_n^{2k}\omega_n^n=(\omega_n^k)^2$$

3．求和引理

对任意整数 $n \geqslant 1$ 和不能被 n 整除的非负整数 k，有

$$\sum_{j=0}^{n-1}(\omega_n^k)^j = 0$$

使用这样的点值表示，可以使用快速傅里叶变换在时间复杂度 $O(n\log n)$ 内完成两种表示方式间的转换。

5.1.3　DFT 和 FFT 的实现

DFT（Discrete Fourier Transformation），即离散傅里叶变换，主要是多项式的系数向量转换成点值表示的过程，其逆变换记为 DFT^{-1}，是多项式的点值表示转换成系数向量的过程。

FFT（Fast Fourier Transformation），即快速傅里叶变换，是离散傅里叶变换的加速算法，可以在时间复杂度 $O(n\log n)$ 里完成 DFT，用相似性也可以在同样的时间复杂度里完成 DFT^{-1}。

在 DFT 中，希望计算多项式 A 在复数根 $\omega_n = (\omega_n^0, \omega_n^1, \cdots, \omega_n^{n-1})$ 处的值，也就是求

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

称向量 $\boldsymbol{y} = (y_0, y_1, ..., y_{n-1})$ 是系数向量 $a = (a_0, a_1, ..., a_{n-1})$ 的离散傅里叶变换，记为 $\boldsymbol{y} = \mathrm{DFT}_n(\boldsymbol{a})$

使用秦九韶算法可直接计算 DFT 的时间复杂度是 $O(n^2)$，而利用复数根的特殊性质的话，可以在时间复杂度 $O(n\log n)$ 内完成，这种方法就是 FFT。在 FFT 中采用分治策略来进行操作，主要利用了消去引理的推论。

在 FFT 中，两个多项式次数需相同且是 2 的整数次幂，不足的话在前面补系数 0，每一步，将当前的多项式 A、次数是 2 的倍数分成两个部分：

$$A^0(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1}$$

$$A^1(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1}$$

则有

$$A(\mathrm{x}) = A^0(x^2) + xA^1(x^2)$$

问题就转化为求出次数是 $\frac{n}{2}$ 的多项式 $A^0(x)$ 和 $A^1(x)$ 在 n 个 n 次单位复数根的平方 $(\omega_n^0)^2, (\omega_n^1)^2, \cdots, (\omega_n^{n-1})^2$ 处的取值。由折半引理可得，这 n 个数其实只有 $\frac{n}{2}$ 个不同的值，多项式由一个变为两个，多项式次数降为原先的一半，需求值的点数减少为原先的一半，故每次递归，计算规模降为原先的一半，如此递归下去，即可在时间复杂度 $O(n\log n)$ 内完成 n 个单

位复数根处的求值操作，进而完成系数向量到点值表示的转换。

在进行 DFT 之后，将多项式的系数表示成功转为点值表示，可以在 $O(n)$ 的时间复杂度内求得乘积多项式的点值表示，那么又如何在 $O(n\log n)$ 的时间复杂度内完成 DFT^{-1}，通过 DFT^{-1} 将点值表示还原为系数表示呢？

用矩阵乘积的形式来表示向量 $\boldsymbol{y}=(y_0,y_1,...,y_{n-1})$ 和系数向量 $\boldsymbol{a}=(a_0,a_1,\cdots,a_{n-1})$ 的关系，再将单位复数根写成的范德蒙德矩阵并求逆，即可推得逆变换，这里不再证明。

根据逆矩阵可以发现，要计算 $\mathrm{DFT}_n^{-1}(y)$ 的话，有关系式

$$a_j=\frac{1}{n}\sum_{k=0}^{n-1}y_k\omega_n^{-kj}$$

所以只需要用 ω_n^{-1} 替换掉 ω_n，最后结果再除以 n 即可，计算 DFT^{-1} 的方法和计算 DFT 相似，都可以在 $O(n\log n)$ 的时间复杂度内解决。

上述过程可以用卷积的形式来表达。对于两个多项式的系数向量 $\boldsymbol{a}=(a_0,a_1,\cdots,a_{n-1})$ 和 $\boldsymbol{b}=(b_0,b_1,\cdots,b_{n-1})$，两个多项式相乘得到的多项式的系数向量 $\boldsymbol{c}=(c_0,c_1,\cdots,c_{2n-2})$ 满足 $c_j=\sum_{k=0}^{j}a_kb_{j-k}$，称系数向量 $\boldsymbol{c}$ 是输入向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的卷积，记为 $\boldsymbol{c}=\boldsymbol{a}*\boldsymbol{b}$。

卷积定理 对任意两个长度为 n 的向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，其中 n 是 2 的幂，有

$$\boldsymbol{a}*\boldsymbol{b}=\mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(\boldsymbol{a})\cdot\mathrm{DFT}_{2n}(\boldsymbol{b}))$$

式中，向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 高次系数用 0 填充，使其长度达到 $2n$，并用“ · ”表示两个 $2n$ 个元素组成向量的点乘。

这个式子实际上就是多项式的系数表示在相乘时进行的卷积运算得到的结果，等同于通过将其系数进行 DFT 变成点值表示之后相乘再换回来的过程。

以一道简单的题目为例，介绍 FFT 的具体实现方法。

5.1.4 例题讲解

例 5-1 多项式乘法

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

给定两个多项式，请输出相乘后的多项式。

输入：

第一行输入两个整数 n 和 m，分别表示两个多项式的次数。

第二行输入 n+1 个整数，分别表示第一个多项式的 0 到 n 次项前的系数。

第三行输入 m+1 个整数，分别表示第一个多项式的 0 到 m 次项前的系数。

输出：

一行，共 n+m+1 个整数，分别表示相乘后的多项式的 0 到 n+m 次项前的系数。

样例输入：

1 2

1 2

1 2 1

样例输出：

1 4 5 2

题目来源：UOJ #34。

解题思路：

多项式乘法问题是 FFT 解决的最基本的问题。FFT 的实现方法有很多，本例题采用一种较为简单、清晰的方法来实现 FFT，并且这一实现在大多数情况下能够满足算法竞赛的需求。

题目实现：

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 2e6+233;
#define DB double
const DB pi = acos(-1);
struct CP {
      DB x, y; CP(){} inline CP(DB a, DB b):x(a),y(b){}
      inline CP operator + (const CP&r) const {
return CP(x + r.x, y + r.y); }
      inline CP operator - (const CP&r) const {
return CP(x - r.x, y - r.y); }
      inline CP operator * (const CP&r) const {
return CP(x*r.x-y*r.y, x*r.y+y*r.x); }
      inline CP conj() { return CP(x, -y); }
} a[maxn], b[maxn], t;

int n, m;
inline void Swap(CP&a, CP&b) { t = a; a = b; b = t; }
const int BUF=8096000,OUT=8000000;

char Buf[BUF],*buf=Buf,Out[OUT],*ou=Out;
int Outn[64],Outcnt;

inline int read() {
       int a;
       for(a=0;*buf<48;buf++);
       while(*buf>47)   a=a*10+*buf++-48;
       return a;
```

```
29  }
30
31  inline void print(int x){
32          if(!x) *ou ++ = 48;
33          else {
34                  for(Outcnt=0;x;x/=10) Outn[++Outcnt]=x%10+48;
35                  while(Outcnt) *ou++=Outn[Outcnt--];
36          }
37          *ou++=' ';
38  }
39
40  void FFT(CP*a, int n, int f) {
41          int i, j, k;
42          for(i = j = 0; i < n; ++ i) {
43                  if(i > j) Swap(a[i], a[j]);
44                  for(k = n>>1; (j ^= k) < k; k >>= 1);
45
46          }
47          for(i = 1; i < n; i <<= 1) {
48                  CP wn(cos(pi/i), f*sin(pi/i));
49                  for(j = 0; j < n; j += i<<1) {
50                          CP w(1, 0);
51                          for(k = 0; k < i; ++ k, w=w*wn) {
52                                  CP x = a[j+k], y = w*a[i+j+k];
53                                  a[j+k] = x+y; a[i+j+k] = x-y;
54                          }
55                  }
56          }
57          if(-1 == f) for(i = 0; i < n; ++ i) a[i].x /= n;
58  }
59
60  int main() {
61          fread(Buf, 1, BUF, stdin);
62          n = read(); m = read();
63          for(int i = 0; i <= n; ++ i) a[i].x = read();
64          for(int i = 0; i <= m; ++ i) a[i].y = read();
65          for(m += n, n = 1; n <= m; n <<= 1);
66          FFT(a, n, 1); CP Q(0, -0.25);
68          for(int i = 0, j; i < n; ++ i)
69  j = (n-i)&(n-1), b[i] = (a[i]*a[i]-(a[j]*a[j]).conj())*Q;
70          FFT(b, n,-1);
```

```
71          for(int i = 0; i <= m; ++ i) print(int(b[i].x+0.2));
72          fwrite(Out, 1, ou-Out, stdout);
73          return 0;
74      }
```

需要注意的是，使用结构体实现的 complex 函数比系统自带的效率要高很多。在算法竞赛中，complex 函数一般使用自定义结构体实现，可防止因常数过大造成的超时。

5.2　NP 完全性

NP 完全性是计算复杂性理论中的一个重要概念，它用于表征某些问题的固有复杂度。一旦确定一类问题具有 NP 完全性时，就可知道这类问题实际上是具有相当复杂程度的问题。本节将对 NP 问题进行简要介绍，并以哈密顿回路问题为例对 NP 问题的求解方法加以分析。

5.2.1　NP 问题简介

想要了解什么是 NP 问题，首先要理解什么是 P 问题。P 问题是指能够在多项式时间复杂度内解决的问题。在之前的学习中，接触到大多数的问题都是 P 问题，这些问题都可以在多项式时间复杂度内解决。而 NP 问题指的是可以在多项式时间复杂度内验证一个解的问题，换句话来说，NP 问题就是可以在多项式时间复杂度内“猜出”一个解的问题。

请注意，NP 问题不是“非 P”问题，根据上述定义，显然所有 P 类问题都是 NP 类问题，即能够在多项式时间复杂度内解决的问题，一定能在多项式时间复杂度内验证一个解。现在想知道的是，是否所有的 NP 类问题也是 P 类问题，如果把 P 和 NP 问题各自看成一个集合，现在想知道，是否 P=NP？。遗憾的是，至今没有人能够证明这一命题成立或不成立。但是大多数人认同 P≠NP。

5.2.2　哈密顿回路

哈密顿回路是指图 G 的一个回路，该回路经过且只经过图的每个节点一次。存在哈密顿回路的图称为哈密顿图。对于一个哈密顿图，可以从一点出发，经过每个点一次后回到起点，回路中允许图中的边未被经过。这与欧拉图的概念恰恰相对，欧拉图实际上讨论的是边的可行遍历问题，与点无关。

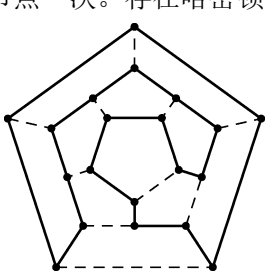


图 5.1　哈密顿回路

如图 5.1 所示，黑色的点代表图的节点，虚线代表图中的边，由实线连起来的回路经过图中的每个节点各一次，即为一条哈密顿回路。

由哈密顿回路引入的最短旅行商（Traveling Salesman Problem，TSP）问题：有 n 个城市，两两之间均有道路直接相连，给出每两个城市 i 和 j 的道路长度 dist(i, j)，求一条哈密顿回路，使得经过的道路

总长度最短，其中 $n \leqslant 15$，城市编号为 0～n−1。

TSP 问题在离散数学中有所涉及，这是一个被证明的 NP 问题，难以在多项式时间复杂度内解决。但本题数据规模较小，常见的解题策略是使用状态压缩动态规划的方式解题。对于本题，起点和终点的选择并不影响计算结果，不妨假设起点和终点的节点编号均为 0。本题具体的策略如下：

（1）状态定义：$d(i,S)$ 表示当前在城市集合 S 中的访问各个城市各一次后回到城市 0 的最短长度。

（2）状态转移方程：$d(i,S)=\min\{d(j,S-\{j\})+\text{dist}(i,j)\mid j\in S\}$。

（3）边界：$d(i,\{\})=\text{dist}(0,i)$。

由上述状态定义可知，最终答案为 $d(0,\{1,2,3,\cdots,n-1\})$，该算法的时间复杂度为 $O(2^n n^2)$

5.2.3 例题讲解

例 5-2 Tour Route

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

某个城市有 N 个景点和 M 条单向边。对于该城市的任意两个景点 A 和 B，存在且仅可能存在以下两种情况：

（1）A 景点能够通过一些单向边到达 B 景点，但 B 景点无法到达 A 景点。

（2）B 景点能够通过一些单向边到达 A 景点，但 A 景点无法到达 B 景点。

求这个城市中的一条回路，使得这条回路从某个景点出发，经过且仅经过全部景点一次，最后又能回到起点。

输入：

输入由多个测试样例组成，并以“0”行结束。

第一行包含一个整数 n（$0<n\leqslant 1000$），代表城市景点的数量。风景名胜区编号为 1 至 N。

接下来 N 行，每行由 n 个整数组成，并构成一个矩阵。如果第 i 行和第 j 列中的元素是 1，则道路的方向是从点 i 到点 j。如果该元素是 0，则道路方向是从 j 到 i。矩阵的主对角线中的数字都是 0。

输出：

对于每一个测试用例，根据一条直线上的路径的移动顺序打印所有的点。如果存在多条路线，只要打印其中一条即可。如果没有这样的路线，打印一个“−1”代替。因为起始点与结束点相同，所以不需要打印结束点。

样例输入：

5

0 0 1 1 1

1 0 1 1 0
0 0 0 1 0
0 0 0 0 1
0 1 1 0 0
2
0 1
0 0
0

样例输出：

1 3 4 5 2
−1

题目来源：HDU 3414。

解题思路：

本题是一个哈密顿回路问题，但是题目中图却是一个特殊的图：其中每对不同的顶点通过单个有向边连接，即这是一个每对顶点之间都有一条边相连的有向图，这样的图称为**竞赛图**。

竞赛图有一个非常重要的性质，即任何有限数量 n 个顶点的竞赛图都包含一个哈密尔顿路径。对于本题而言，由于竞赛图一定存在哈密顿路径，但不一定存在哈密顿回路，所以需要枚举所有起点，构造一个哈密顿路径，然后判断起点和终点是否连通即可解决此问题。

题目实现：

```
#include <iostream>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <algorithm>
#include <queue>
#include <stack>
#include <vector>

using namespace std;
typedef long long LL;
const int maxN = 1000;

while(!(((c=getchar())>='0')&&(c<='9')));a=c-'0';while(((c=getchar())>='0')&&(c<='9'))(a*=10)+=c-'0';}

void Hamilton(int ans[maxN + 7], int map[maxN + 7][maxN + 7], int n, int st) {
```

```
18          int nxt[maxN + 7];
19          memset(nxt, -1, sizeof(nxt));
20          int head = st;
21          for(int i = 1; i <= n; i++) {
22              if(i == st)continue;
23              if(map[i][head]) {
24                  nxt[i] = head;
25                  head = i;
26              }else {
27                  int pre = head, pos = nxt[head];
28                  while(pos != -1 && !map[i][pos]) {
29                      pre = pos;
30                      pos = nxt[pre];
31                  }
32                  nxt[pre] = i;
33                  nxt[i] = pos;
34              }
35          }
36          int cnt = 0;
37          for(int i = head; i != -1; i = nxt[i]) ans[++cnt] = i;
38      }
39
40      int main()
41      {
42          int N;
43          while(~scanf("%d", &N) && N) {
44              int map[maxN + 7][maxN + 7] = {0};
45
46              for(int i = 1; i <= N; i++) {
47                  for(int j = 1; j <= N; j++) {
48                      int u; read(u);
49                      map[i][j] = u;
50                  }
51              }
52              if(N == 1){ printf("1\n");continue; }
53              int ans[maxN + 7] = {0}, i;
54              for(i = 1; i<= N; i++) {
55                  Hamilton(ans, map, N, i);
56                  if(map[ans[N]][ans[1]]) {
57                      for(int j = 1; j <= N; j++) {
58                          printf(j == 1 ? "%d":" %d", ans[j]);
59                      }
```

```
60                          break;
61                      }
62                  }
63                  if(i > N)printf("-1");
64                  printf("\n");
65              }
66              return 0;
68      }
```

例 5-3　Prison Break

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

某一个机器人 Michael#1 要逃出监狱，每走一步消耗一点电量，初始时电量是满的。给定一个大小为 $n \times m(n、m \leqslant 15)$ 的矩阵代表监狱，F 代表起始点，G 代表充电站，每个 G 只能将电量充满一次，Y 代表开关，D 不能经过，S 表示空地。要求打开所有开关，电池的满电量最少是多少。如果不能逃出则输出−1。G 和 Y 的个数和不会超过 15。

输入：

输入包含多个测试用例，以 0×0 结尾。对于每个测试用例，第一行包含两个整数，n 和 m 表示监狱的大小。接下来 n 行由 M 个大写字母组成，代表监狱的描述，可以假设 $1 \leqslant n、m \leqslant 15$，能量池和电源开关之和小于 15。

输出：

对于每个测试用例，在一行中输出一个整数，表示 Michaely#1 需要的电池最小容量。如果 Michaely#1 不能逃脱，输出−1。

样例输入：

```
5 5
GDDSS
SSSFS
SYGYS
SGSYS
SSYSS
0 0
```

样例输出：

```
4
```

题目来源：HDU 3681。

解题思路：

对于本题，答案有这样一个明显的性质：当验证一个满电量可行时，比这个满电量更大

的电量也一定可行，因而考虑采用二分答案的方法处理。对满电量进行二分，则问题转化为判断一个给定的电量是否可行。显然，G 和 Y 应当经过且只经过一次。如果用 BFS 预处理出每两个点之间的距离，则计算耗电量的问题转化为经典的 TSP 问题。

题目实现：

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

struct Point {
    int x, y;
    int d;
    Point(int x, int y, int d) : x(x), y(y), d(d) {}
    Point(int x, int y) : x(x), y(y), d(0) {}
    Point() {}

    bool operator ==(const Point a) const {
        if (x == a.x && y == a.y) return true;
        return false;
    }

} p[50];

int m, n;
char mp[20][20];
int cf;
int dis[20][20];
int cnt = 0;
int ac = 0;
int dir[4][2] = {0, 1, 0, -1, 1, 0, -1, 0};
int vis[20][20];
int dp[1<<17][20];

bool ok(int x, int y)
{
    if (x < n && x >= 0 && y < m && y >= 0 &&
        mp[x][y] != 'D' && !vis[x][y])
        return true;
    return false;
}

int bfs(int a, int b)
```

```
{
    memset(vis, 0, sizeof(vis));
    queue<Point> q;
    q.push(p[a]);
    vis[p[a].x][p[a].y] = 1;
    while (!q.empty()) {
        Point now = q.front();
        q.pop();


        if (now == p[b]) return now.d;
        for (int i = 0; i < 4; ++i) {
            int nx = now.x + dir[i][0];
            int ny = now.y + dir[i][1];
            if (!ok(nx, ny)) continue;
            int nd = now.d + 1;
            q.push(Point(nx, ny, nd));
            vis[nx][ny] = 1;
        }
    }
    return -1;
}

void getDis()
{
    for (int i = 0; i < cnt; ++i) {
        for (int j = i; j < cnt; ++j) {
            if (i == j) dis[i][j] = 0;
            else dis[j][i] = dis[i][j] = bfs(i, j);
        }
    }
}

bool canGo(int en)
{
    memset(dp, -1, sizeof(dp));
    int st = (1 << cnt);
    dp[1 << cf][cf] = en;
    for (int i = 0; i < st; ++i) {
        for (int j = 0; j < cnt; ++j) {
            if ( !((1 << j) & i) || dp[i][j] == -1 ) continue;
```

```
79                  if ((i & ac) == ac) return true;
80                  for (int k = 0; k < cnt; ++k) {
81                      if ( (1 << k) & i || dis[j][k] == -1 || dp[i][j] < dis[j][k] ) continue;
82                      int nt = (1 << k) | i;
83                      dp[nt][k] = max(dp[nt][k], dp[i][j] - dis[j][k]);
84                      if (mp[ p[k].x ][ p[k].y ] == 'G') dp[nt][k] = en;
85                  }
86              }
87          }
88          return false;
89      }
90
91      int main()
92      {
93          while (~scanf("%d%d", &n, &m)) {
94              if (n == 0 && m == 0) break;
95
96              for (int i = 0; i < n; ++i)
97                  scanf("%s", mp[i]);
98
99              ac = cnt = 0;
100             for (int i = 0; i < n; ++i) {
101                 for (int j = 0; j < m; ++j) {
102                     if (mp[i][j] == 'F') {
103                         cf = cnt;
104                         ac += (1 << cnt);
105                         p[cnt++] = Point(i, j);
106                     } else if (mp[i][j] == 'G') {
107                         p[cnt++] = Point(i, j);
108                     } else if (mp[i][j] == 'Y') {
109                         ac += (1 << cnt);
110                         p[cnt++] = Point(i, j);
111                     }
112                 }
113             }
114
115             getDis();
116
117             int l = 0, r = 300;
118             while (l <= r) {
119                 int mid = (l + r) >> 1;
```

```
120                 if (canGo(mid)) r = mid - 1;
121                 else l = mid + 1;
122             }
123             if (l >= 300) l = -1;
124             printf("%d\n", l);
125         }
126         return 0;
127     }
```

5.3　对偶图问题

平面图的对偶图问题是一个经典的算法问题。解决这一类问题主要方法是将平面图转化为对偶图，利用对偶图的有关性质进行解题。

5.3.1　基本概念

平面图（Planar Graph）是指一个可以在平面上画出来的图，并且所有的边只在顶点处相交。平面图的**对偶图**（Dual Graph）是将这个平面的每个区域看成点，原图每一条边所属的两个相邻的区域对应在对偶图中的点有连边。如图 5.2 所示，实线部分为一个平面图，虚线部分即为该平面图的对偶图。

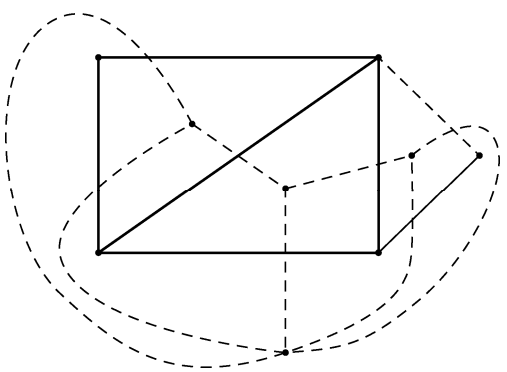


图 5.2　平面图与对偶图（一）

平面图具有一些很好的性质，其中，最广为人知的、应用最多的性质就是欧拉公式，在任何一个平面图上，用 R 表示区域个数，V 表示顶点个数，E 表示边界个数，F 表示为面数，则 $R+V-E=2$，即 $V-E+F=2$。

用下面一道例题来说明欧拉公式的应用。

例 5-4　框架堆叠

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 10000/10000 KB (Java/Others)

题目描述：

有一块椭圆形的地，可以在边界上选 n 个点，并两两连接得到 $n(n-1)/2$ 条线段。它们最多能把土地分成多少个部分？

输入：

输入的第一行包含一个整数 S（$0<S<3500$），表示测试样例的组数。接下来 S 行中每行包含一个整数 n，含义如上所述。

输出：

对于每组测试数据，输出一个整数，表示最多能把土地分割的数量。

样例输入：

4
1
2
3
4

样例输出：

1
2
4
8

题目来源：UVa 10213。

解题思路：

最优方案是不让三条线段交于 1 点。根据欧拉公式可知，$V-E+F=2$，其中 V 是顶点（即所有线段的断点数加上交点数），E 是边数（即 n 段椭圆弧加上这些线段被切成的段数），F 是面数（即土地块数加上椭圆外那个无穷大的面），换句话说，只需求出 V 和 E，答案就是 $E-V+1$；不管是定点还是边，计算时都要枚举一条从固定点出发（所以最后要乘以 n）的所有对角线。假设该对角线左边有 i 个点，右边有 $n-2-i$ 个点，则左右两边的点两两搭配后在这条对角线上形成了 $i(n-2-i)$个交点，得到了 $i(n-2-i)+1$ 条线段。注意：每个交点被重复计算了 4 次，而每条线段被重新计算了 2 次，因为形成每个交点需要由 4 个点两两组成的 2 条线段相交于一点，需要 2 个点形成 1 条被分割的线段，所以可得

$$V = n + n\left[\sum_{i=0}^{n-2} i(n-2-i)\right]$$

$$E = n + n\left\{\sum_{i=0}^{n-2}[i(n-2-i)+1]\right\}/2$$

如果 n=1～5，答案为：1、2、4、8、16，可能就会推出 n=6 时是 32，但不是，而是 31，因此找规律的时候要谨慎！

题目实现：

```
#include<stdio.h>
int main()

{
    int n;
    scanf("%d", &n);
    int v = 0, e = 0;
    for (int i = 0; i <= n-2; i++)
        v += i * (n - 2 - i), e += i * (n - 2 - i) + 1;
    v = v * n / 4 + n;
    e = e * n / 2 + n;
    printf("%d\n", e-v+1);
    return 0;
}
```

5.3.2　平面图转化为对偶图

从例 5-4 可以看出，平面图转化为对偶图后，有一些很好的性质。求解网络流问题需要的时间很多，但是，将平面图转化为对偶图后，最小割问题巧妙地转化为了最短路问题，大大提高了解题的效率。但遗憾的是，并不是所有的平面图都可以像上述的例子一样容易转化为对偶图。在上例中，由于给定的平面图形状确定，且对偶图点和边的分布有明显的规律，实际上做的只是对不同的区域进行重新标号，转化过程其实并没有体现在程序中，而是通过自己的计算得到的。

如果对于一般的平面图，给定顶点的坐标和边的连线，应该怎样去构造平面图的对偶图呢？首先来看一个平面图，以及它的对偶图，如图 5.3 所示。

平面图转对偶图的算法被称为最左转线，算法步骤如下：

步骤 1：把所有的边改成双向边。

步骤 2：对每个点的出边按照极角排序。

步骤 3：找一条没有标记过的边(u,v)，将(u,v)设为当前边。

步骤 3.1：将当前边(u,v)标记。

步骤 3.2：找到 v 的出边中极角序在(v,u)前的最后一条边，设为下一条的当前边。

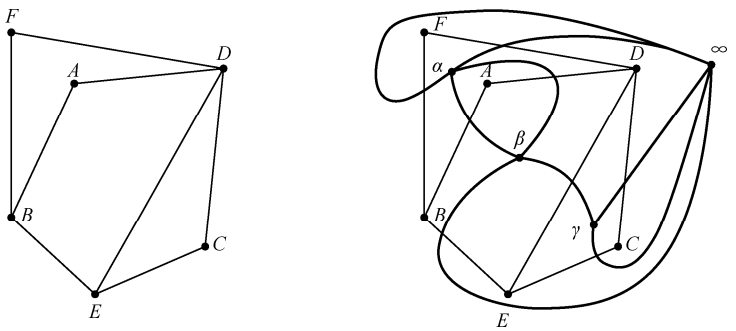


图 5.3　平面图和对偶图（二）

步骤 3.3：重复步骤 3.1 和 3.2，直到找到一条已经被标记过的当前边，这时候就找到了一个区域。

步骤 4：不断重复步骤 3，直到所有边都被标记过。

把逆时针方向围成一个区域的边认为是属于这个区域的，这样一条边就只会属于唯一一个区域。注意到有一个十分特别的区域，它是无界的，在步骤 3 中，计算出这个区域的有向面积，如果是正的，那就说明是有界域，否则就是无界域。

这个算法直观上来讲，就是不断找到一条条边右边最“左转”的边，这样最后就会得到一个个由逆时针方向的边构成的区域（当然无界域看起来是顺时针的）。下面给出程序的实现：

```
struct point_t{
    double x, y;
    point_t(double x = 0, double y = 0) : x(x), y(y) {}
    point_t operator - (const point_t& r) const
    {
        return point_t(x - r.x, y - r.y);
    }

    friend double cross(const point_t& a, const point_t& b)
    {
        return a.x * b.y - a.y * b.x;
    }
} pts[maxn];

struct edge_t{
    int u, v;
    double dist,angle;
    edge_t(int u = 0, int v = 0, double len = 0) : u(u), v(v), dist(len)
    {
        point_t z = pts[v] - pts[u];
```

```
21              angle = atan2(z.y, z.x);
22              if(angle < 0) angle += 2.0 * M_PI;
23          }
24          bool operator<(const edge_t& b) const{
25              return angle<b.angle;
26          }
27      };
28      void find_region(int x, int eid)
29      {
30          if(vis[eid]) return;
31          double area = 0;
32          while(!vis[eid])
33          {
34              area += cross(pts[x], pts[edges_t[eid].v]);
35              vis[eid] = 1;
36              near[eid] = region_tot;
37              x = edges_t[eid].v;
38              if(!rank[eid ^ 1]) eid = et[x].back();
39              else eid = et[x][rank[eid ^ 1] - 1];
40          }
41          if(area < 0) inf_area = region_tot;
42          ++region_tot;
43      }
44
45      void work(){
46          pair<double, int> *tmp = new pair<double, int>[m << 1];
47          for(int i = 0; i != m << 1; ++i)
48              tmp[i] = make_pair(edges_t[i].angle, i);
49          sort(tmp, tmp + (m << 1));
50          for(int i = 0; i != m << 1; ++i)
51          {
52              int eid = tmp[i].second;
53              edge_t e = edges_t[eid];
54              rank[eid] = et[e.u].size();
55              et[e.u].push_back(eid);
56          }
57
58          delete[] tmp;
59
60          for(int i = 1; i <= n; ++i)
61          {
```

```
62                  for(int j = 0; j != et[i].size(); ++j)
63                      find_region(i, et[i][j]);
64              }
65      }
```

需要注意的是，本方法是一个比较通用的解法，能够解决一般平面图的对偶图问题。如果把它具体到求解最小割的题目中，ISAP（Improved Shortest Augment Path）是图论求最大流的算法之一，实际上，它很好地平衡了运行时间和程序复杂度之间的关系，实际上，ISAP 做法的运行时长远低于按照其理论复杂度计算的时长。

5.3.3 对偶图的应用

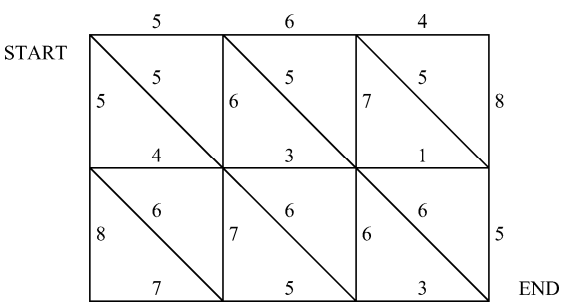
对于一个平面图来说，其对偶图与原图有什么关系呢？换句话说，也就是如何用对偶图解决平面图的问题。下面以一道经典题目为例，详细分析对偶图的在分析平面图问题中的应用。

例 5-5 狼抓兔子

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

现在小朋友们都很喜欢“喜羊羊与灰太狼”，话说灰太狼抓羊抓不到，但抓兔子还是比较在行的，而且现在的兔子还比较笨，它们只有两个窝。现在狼王面对下面这样一个网格的地形：



左上角点为（1,1），右下角点为（N，M）(上图中 N=4，M=5)。有以下三种类型的道路

- $(x,y) \leftrightarrow (x+1,y)$;
- $(x,y) \leftrightarrow (x,y+1)$;
- $(x,y) \leftrightarrow (x+1,y+1)$。

道路上的权值表示这条路上最多能够通过的兔子数，道路是无向的，左上角和右下角为兔子的两个窝。

开始时所有的兔子都聚集在左上角（1,1）的窝里，现在它们要跑到右下角（N,M）的窝中去，狼王开始伏击这些兔子。当然为了保险起见，如果一条道路上最多通过的兔子数为 K，狼王需要安排同样数量的 K 只狼，才能完全封锁这条道路，你需要帮助狼王安排一个伏击方

案，使得在将兔子一网打尽的前提下，参与的狼的数量要最小。

输入：

第一行输入 N、M，表示网格的大小，N、M 均小于或等于 1000。

接下来分三部分描述。

第一部分共 N 行，每行 M–1 个数，表示横向道路的权值。

第二部分共 N–1 行，每行 M 个数，表示纵向道路的权值。

第三部分共 N–1 行，每行 M–1 个数，表示斜向道路的权值。

输入文件保证不超过 10 MB。

输出：

输出一个整数，表示要参与伏击的狼的最小数量。

样例输入：

3 4
5 6 4
4 3 1
7 5 3
5 6 7 8
8 7 6 5
5 5 5
6 6 6

样例输出：

14

题目来源：BZOJ 1001。

解题思路：

本题是一个最小割问题，但点的数量和边的数量都是 1000 量级，朴素做法的时间复杂度较高，不能满足时间要求。对于一般的最小割问题，经典算法的时间复杂度均不能满足本题的要求，那么就理应考察本题的“特性”，以利用“特性”进行求解。

本题与一般的最小割问题的区别在于，本题给定的网络是一个平面图，且是形状规则的平面图，可以利用平面图及其对偶图的性质解题。平面图的对偶图很好构造：将原图中的面变成新图中的点。原图中，每条边必定分割了两个面，在新图中，对应的点之间添加一条边，边权还是原图中边的边权，在原图中一个全局的割就对应了新图中的一个环，也就是说，如果想要求原图中的一个全局最小割，只需要在新图中找一个最小环即可。

怎么求出原图中分割固定点 s 和 t（s 和 t 处于一个无限大的平面边缘）的一个最小割呢？先在原图中添加 s 到 t 的边，给原图增加了一个面。构造原图的对偶图，将由于增边而增加的新面对应的点设为 s，无穷大的平面对应的点设为 t，删掉对偶图中 s 到 t 直接相连的边。求

出 *s* 到 *t* 的最短路即可。因为图为稀疏图，可采用 Dijkstra 算法实现。

题目实现：

```
#include <iostream>
#include <cstring>
#include <cstdio>
#include <queue>

using namespace std;

const int N=2000006, INF=0x3fffffff, E=N*3;

struct ARC {
    int u, val, next;
    inline void init(int a, int b, int c) {
        u=a, val=b, next=c;
    }
} arc[E];
int head[N], tot, S, T, n, m, dis[N];
bool vs[N];

struct data {
    int u, dis;
    data() {}
    data(int a, int b) : u(a), dis(b) {}
    bool operator < (const data &T) const {
        return dis>T.dis;
    }
};

inline void add_arc(int s, int t, int val) {
    arc[tot].init(t, val, head[s]);
    head[s]=tot++;
}

priority_queue <data> Q;
void Dijkstra() {
    fill(dis, dis+T+1, INF);
    fill(vs, vs+T+1, 0);
    while(!Q.empty()) Q.pop();
    dis[S]=0, Q.push(data(S, 0));
```

```
    for(int u; !Q.empty(); ) {
        u=Q.top().u, Q.pop();
        if(vs[u]) continue;
        if(u==T) {
            printf("%d\n", dis[T]);
            break;
        }
        vs[u]=1;
        for(int e=head[u]; e!=-1; e=arc[e].next) {
            int v=arc[e].u;
            if(vs[v] || dis[u]+arc[e].val>=dis[v]) continue;
            dis[v]=dis[u]+arc[e].val;
            Q.push(data(v, dis[v]));
        }
    }
}

void read(int &x) {
    char c;
    while((c=getchar())<'0' || c>'9');
    x=c-'0';
    while((c=getchar())>='0' && c<='9') x=(x<<3)+(x<<1)+c-'0';
}

void Input() {
    for(int i=0, id1, id2, a; i<=n-1; i++)
        for(int j=1; j<=m-1; j++) {
            read(a);
            id1=((i-1)*(m-1)+j)*2-1;
            id2=(i*(m-1)+j)*2;
            if(i==0) id1=T;
            else if(i==n-1) id2=S;
            add_arc(id1, id2, a);
            add_arc(id2, id1, a);
        }

    for(int i=1, id1, id2, a; i<=n-1; i++)
        for(int j=0; j<m; j++) {
            read(a);
            id1=((i-1)*(m-1)+j)*2;
            id2=((i-1)*(m-1)+j+1)*2-1;
```

```
80                  if(j==0) id1=S;
81                  else if(j==m-1) id2=T;
82                  add_arc(id1, id2, a);
83                  add_arc(id2, id1, a);
84              }
85
86          for(int i=1, id1, id2, a; i<=n-1; i++)
87              for(int j=1; j<=m-1; j++) {
88                  read(a);
89                  id1=((i-1)*(m-1)+j)*2;
90                  id2=((i-1)*(m-1)+j)*2-1;
91                  add_arc(id1, id2, a);
92                  add_arc(id2, id1, a);
93              }
94      }
95
96      int main() {
97          read(n), read(m);
98          S=0, T=(n-1)*(m-1)*2+1;
99          fill(head, head+T+1, -1), tot=0;
100         if(n==1 || m==1) {
101             if(n>m) swap(n, m);
102             int ans=INF;
103             for(int i=1, a; i<m; i++) {
104                 read(a);
105                 if(ans>a) ans=a;
106             }
107             printf("%d\n", ans==INF?0:ans);
108         }
109         else Input(), Dijkstra();
110         return 0;
111     }
```

5.4 RMQ 问题

倍增思想是一种十分巧妙的思想，其本质是：每次根据已经得到的信息，将考虑的范围扩大一倍，从而加速操作。在解决信息学问题方面，倍增思想主要有这两个方面的应用：

（1）加速区间操作。

（2）在变化规则相同的情况下加速状态转移。

本节和 5.5 节主要介绍两个经典问题——RMQ 问题和 LCA 问题，并介绍运用倍增算法如何快速高效的解决这两个问题。

RMQ（Range Minimum/Maximum Query）问题是指：对于长度为 n 的数列 A，回答若干询问 RMQ(A,i,j)(i,j≤n)，返回数列 A 中、下标在[i,j]里的最小（大）值，也就是说，RMQ 问题是求区间最值的一类问题 。

5.4.1　RMQ 问题的简单求解方法

求解 RMQ 问题最简单的方法，就是遍历给定区域的数组，用给定区域的数组中每一个值来更新最大（最小值）。

以最大值的求解为例，代码实现如下：

```
1    //求解最大值
2    int l,r;
3    scanf("%d%d",&l,&r);
4    int RMQ_ans=a[l];
5    for(int i=l; i<=r; i++)
6          RMQ_ans=max(RMQ_ans,a[i]);
```

使用这种方法，每次都会遍历一遍给定区域的数组，即每次查询的时间复杂度为 $O(n)$。当查询数很多时，这样的耗时是无法接受的，所以要寻找一种更加高效的求解方法。

5.4.2　ST（Sparse Table）算法

ST 算法是一种更加高效的算法，通过一个时间复杂度为 $O(n\log n)$的预处理求出 ST 数组，换取查询时 $O(1)$的性能。注意，这里的 ST 算法是一种离线的查询方法，不支持修改操作。若需要在查询中修改，需要使用线段树在线算法。

为了优化查询耗时，在查询开始前先预处理出一个 ST 数组。

ST[i][j]表示从第 i 个数起连续 2^j 个数中的 RMQ，ST 数组可以由一个简单的递推方程求出，ST[i][j] = max(ST[i][j−1], ST[i+(1<<(j−1))][j−1])。在递推求解时，先从小到大递推 j，再递推 i，即可确保递推正确性。代码实现如下：

```
1    //求得 ST 数组
2    for(int i=0; i<n; i++) ST[i][0] = a[i];
3    for(int j=1; (1<<j)<=n; j++)
4       for(int i=0; i+(1<<j)-1<n; i++)
5          ST[i][j] = max(ST[i][j-1], ST[i+(1<<(j-1))][j-1] );
```

ST 数组可通过预处理获得，修改非常复杂且耗时，这是 ST 算法不能修改的本质原因。通过前面讲到的 DP，可求出 ST 数组，那么如何通过 ST 数组高效地求出区间的 RMQ 呢？

定义 len = (r − 1 + 1)，并找到最大的满足 2^k ≤len 的 k。

如图 5.4 所示，前 2^k 个数和后 2^k 个数可能有一段交叉（灰色块），这段交叉的最短长度为 0，即前 2^k 个数和后 2^k 个数一定能完整覆盖整个区间。整个区间的 RMQ（以最大值为例）就等于在前 2^k 个数的最大值和后 2^k 个数的最大值之中取一个最大值。因为是 RMQ 问题，绿色块被计算两次不会影响结果。k 可以通过预处理 $O(1)$查到，但是从 0 开始累加找到也不会慢很多。代码如下：

```
1    //查询区间 RMQ
2    int k=0;
3          while(1<<(k+1)<=r-l+1)
4    RMQ_ans_ans = max(ST[l][k], ST[r - (1<<k) + 1][k]);
```

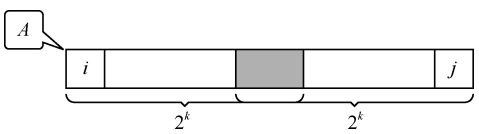


图 5.4　ST 数组求 RMQ 问题

使用 ST 算法时先调用 built_RMQ()函数，原数组数据存放在“a[i]”中，将 ST 数组第二维定义为 log$n$ 大小即可。

```
1    //ST 算法
2    int ST[maxn][30];
3    int a[maxn];
4    int built_RMQ()
5    {
6          for(int i=0; i<n; i++)
7          ST[i][0]=a[i];
8          for(int j=1; (1<<j)<=n; j++)
9          for(int i=0; i+(1<<j)-1<n; i++)
10         ST[i][j]=min(ST[i][j-1],ST[i+(1<<(j-1))][j-1]);
11   }
12   int query_RMQ(int l, int r)
13   {
14         int k=0;
15         while(1<<(k+1)<=r-l+1) k++;
16         return min(ST[l][k], ST[r-(1<<k)+1][k]);
17   }
```

5.4.3　例题讲解

例 5-6　延绵的山峰

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

有一座延绵不断、跌宕起伏的山，最低处海拔为 0，最高处海拔不超过 8848 米，从这座山的一端走到另一端的过程中，每走 1 m 海拔就升高或降低 1 m。有 Q 个登山队计划在这座山的不同区段登山，当他们攀到各自区段的最高峰时，就会插上队旗。写一个程序找出他们插旗的高度。

输入：

第 1 行输出 1 个整数 N（$N \leqslant 10^6$），表示山两端的跨度。接下来 N+1 行，每行一个非负整数 H_i，表示该位置的海拔高度，其中 $H_0=H_n=0$。

接着的一行有一个正整数 Q（$Q \leqslant 7000$），表示登山队的数量。接下来 Q 行，每行有两个数 A_i、B_i，分别表示第 i 个登山队攀爬的区段$[A_i,B_i]$，其中 $0 \leqslant A_i \leqslant B_i \leqslant N$。

输出：

Q 行，每行为一个整数，表示第 i 个登山队插旗的高度。

样例输入：

```
10
0
1
2
3
2
3
4
3
2
1
0
5
0 10
2 4
3 7
7 9
8 8
```

样例输出：

```
4
3
```

4
3
2

题目来源：COGS 58。

解题思路：

本题是寻找区间的最大值，RMQ 问题，可以用 ST 表求解，时间复杂度为 $O(n\log n)$。

题目实现：

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=2000100;
int a[maxn];
int ST[maxn][30];
int build_RMQ(int n)
{
        for(int i=1;i<=n;i++) ST[i][0]=a[i];
        for(int j=1;(1<<j)<=n;j++)
        for(int i=0;i+(1<<j)+1<n;i++)
        ST[i][j]=max(ST[i][j-1],ST[i+(1<<(j-1))][j-1]);
}
int query(int l,int r)
{
        int k=0;
        while((1<<(k+1))<=r-l+1) k++;
        return max(ST[l][k],ST[r-(1<<k)+1][k]);
}
int main()
{
        int n;
        scanf("%d",&n);
        for(int i=0;i<=n;i++) scanf("%d",&a[i]);
        build_RMQ(n+1);
        int q;
        scanf("%d",&q);
        for(int i=1;i<=q;i++)
        {
                int l,r;
                scanf("%d%d",&l,&r);
                l++,r++;
```

```
32                  printf("%d\n",query(l,r));
33          }
34  }
```

例 5-7　均衡队形

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

农夫约翰有 N（$1 \leqslant N \leqslant 50000$）头奶牛，每天挤奶时它们总会按同样的顺序站好。一日，农夫约翰决定为奶牛们举行一个“终极飞盘”比赛。为简化问题，他将从奶牛队列中选出一个连续区间来进行游戏。不过，参加游戏的奶牛要玩得开心的话就不能在身高上差距太大。

农夫约翰指定了 Q（$1 \leqslant Q \leqslant 200000$）个参赛组，给出它们的身高（$1 \leqslant$ 身高 $\leqslant 1000000$）。对于每个参赛组，他需要确定参赛组中最高牛和最低牛的身高差。

输入：

第 1 行输入用两个空格隔开的整数 N 和 Q。

第 2 至 N+1 行：第 i+1 行包含一个整数表示第 i 头牛的身高。

第 N+2 至 N+Q+1 行：每行包含两个整数 A 和 B（$1 \leqslant A \leqslant B \leqslant N$），表示一个从 A 到 B 的参赛组区间。

输出：

第 1 至 Q 行：每行包含一个整数来表示区间上最大身高差。

样例输入：

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

样例输出：

```
6
3
0
```

题目来源：COGS 182。

解题思路：

本题求区间的最大值和最小值的差值，可以分别求区间最大值和最小值，同样是 RMQ 问题，可以用 ST 表求解，时间复杂度为 $O(n\log n)$。

题目实现：

```
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;
int n,a,b,q,ffmin[60000][30],ffmax[60000][30];
int in()
{
    int x=0;
    char ch;
    ch=getchar();
    while(ch<'0'||ch>'9') ch=getchar();
    while(ch>='0'&&ch<='9') x=x*10+ch-'0',ch=getchar();
    return x;
}
int main()
{
    freopen("lineup.in","r",stdin);
    freopen("lineup.out","w",stdout);
    scanf("%d%d",&n,&q);
    for (int i=1;i<=n;i++)
    {
        a=in();
        ffmin[i][0]=a;ffmax[i][0]=a;
    }
    for (int j=1;j<=20;j++)
    for (int i=1;i<=n;i++)
    if (i+(1<<j)-1<=n)
    {
        ffmin[i][j]=min(ffmin[i][j-1],ffmin[i+(1<<(j-1))][j-1]);
        ffmax[i][j]=max(ffmax[i][j-1],ffmax[i+(1<<(j-1))][j-1]);
    }
    else break;
    for (int i=1;i<=q;i++)
    {
        int k,maxx,minn;
        a=in();b=in();
```

```
37              k=(int)(log(b-a+1.0)/log(2.0));
38              maxx=max(ffmax[a][k],ffmax[b-(1<<k)+1][k]);
39              minn=min(ffmin[a][k],ffmin[b-(1<<k)+1][k]);
40              printf("%d\n",maxx-minn);
41          }
42      }
```

5.5　LCA 问题

LCA（Least Common Ancestors）问题，即最近公共祖先问题，是指在有根树中，找出某两个节点 u 和 v 最近的公共祖先。对于有根树 T 的两个节点 u、v，最近公共祖先 LCA(T,u,v) 表示一个节点 x，满足 x 是 u、v 的祖先且 x 的深度尽可能大。另一种理解方式是把 T 理解为一个无向无环图，而 LCA(u,v)是 u 到 v 的最短路（T 是一棵树所以最短路唯一）上深度最小的点。

在图 5.5 的有根树中：

LCA(11,7)=1
LCA(12,7)=4
LCA(12,4)=4

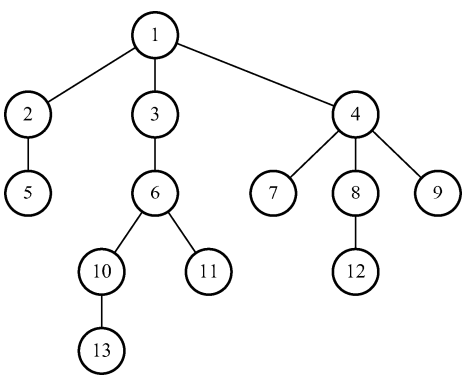


图 5.5　图的公共祖先

5.5.1　LCA 问题的简单求解方法

不考虑效率问题，LCA 的求解方法非常简单。使用双亲表示法保存图，即存储每个节点的父节点 fa，计算 LCA 时先 DFS 一遍，预处理出每个点距离根节点的距离 deep。当计算 LCA(u,v)时，只需通过 fa 把 deep 较大的一个节点“上移”，移至两节点的 deep 相同，如果此

时两节点不重合，再将两个节点一起“上移”，直到两节点重合。重合时的节点即 LCA(u,v)。

每次计算 LCA(u,v)的时间复杂度是 $O(n)$，因为每次计算都需要把整个树遍历一遍。当要计算很多组 LCA 且 n 比较大时，这样的时间复杂度是无法接受的，所以要寻求效率更高的做法。

5.5.2 基于倍增的双亲存储法

分析上一节提出的简单算法，算法效率不高的原因在于每次“上移”的时间复杂度太高，每次“上移”操作都需要遍历一遍整棵树，比如每次都查询两个很深的叶子节点的 LCA。

既然一步一步“上移”很慢，就要考虑能不能在不超过 LCA 的情况下，每次多“上移”几步。为了实现这个操作，不仅要存储每个节点父节点的编号，还要存储每个节点“上移”2^k 个节点后的节点编号，因此计算需要基于倍增的双亲数组 F。

定义 $F[i][j]$为编号 i 的节点“上移”2^k 个点后的节点编号，则 $F[i][0]$代表 i 节点的父节点，可以通过进行一遍 DFS 求得。对于 j 大于 0 的情况，可以通过递推来解决：

$$F[i][j] = F[F[i][j-1]][j-1]$$

即 i 上移 2^j 步与 i 先上移 2^{j-1} 后，再上移 2^{j-1} 步是等效的。通过这个递推式，可以在 $O(n\log n)$ 的时间复杂度内算出数组 F。

5.5.3 高效的 LCA 算法

有了上一节求出的数组 F，可以实现多“上移”几步来减小时间复杂度，那么如何保证减小时间复杂度并且不跳过 LCA 节点呢？

首先，对于高度不同的两个节点，还是先把它们中较深的一个“上移”，使两点同高。若两节点的深度差为 len，即把较深节点“上移”len 步。从 i 开始，从小到大循环 j，若 2^j & len > 0，就把 i 上移到 $F[i][j]$，相当于用二进制表示 len。最多需要“上移”log(len)步。代码如下：

```
1    for(int j=0;(1<<j)<=len;j++)
2    if(len&(1<<j)>0) i=F[i][j];
```

两个节点到达同一高度后，首先要判断两个节点是否重合，若重合，重合点就是所求的 LCA。当两个节点不重合时，要将两个节点一起“上移”。从大到小循环 j，若两个节点都“上移”2^j 步后没有重合，就“上移”两个节点，否则不“上移”。最后，两个节点一定是 LCA 节点的两个子节点，随便取一个节点的父节点，就是所求的 LCA。代码如下：

```
1    for(int j=logn;j>=0;j--)
2    if(f[a][j]!=f[b][j]) a=f[a][j],b=f[b][j];
```

下面的模板中包含三个函数，dfs 用于求出深度数组和每个点的父节点，init 通过递推求出基于倍增的双亲数组，lca 可以直接调用求得任意两个节点的最公共祖先。

```
int deep[maxn];          //深度数组
int f[maxn][30];         //基于倍增的双亲数组
vector<int>a[maxn];   //vector 版的邻接矩阵
int vis[maxn];
int dfs(int x)
{
    for(int i=0;i<a[x].size();i++)
    if(!vis[a[x][i]])
    {
        f[a[x][i]][0]=x;
        deep[a[x][i]]=deep[x]+1;
        vis[a[x][i]]=1;
        dfs(a[x][i]);
    }
}
int init()
{
    for(int i=1;(1<<i)<=n;i++)
    for(int j=2;j<=n;j++)
    {
        f[j][i]=f[f[j][i-1]][i-1];
    }
}
int lca(int l,int r)
{
    if(deep[l]>deep[r]) swap(l,r);
    int len=deep[r]-deep[l];
    for(int i=0;(1<<i)<=len;i++) if((1<<i)&len) r=f[r][i];
    if(l==r) return l;
    for(int i=(int)log2(n);i>=0;i--)
    {
        if(f[l][i]!=f[r][i])
        {
            l=f[l][i];
            r=f[r][i];
        }
    }
    return f[l][0];
}
```

dfs 的时间复杂度为 $O(n)$，init 的时间复杂度为 $O(n\log n)$，lca 的时间复杂度为 $O(\log n)$。

5.5.4 例题讲解

例 5-8 货车运输

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

A 国有 n 座城市，编号为 1 到 n，城市之间有 m 条双向道路。每一条道路对货车都有质量限制，简称限载。现在有 q 辆货车在运输货物，司机们想知道每辆货车在不超过限载的情况下，最多能运多少的货物。

输入：

第 1 行输入 2 个由一个空格隔开的整数 n 和 m，表示 A 国有 n 座城市和 m 条道路。

接下来 m 行中每行有 3 个整数 x、y、z，每两个整数之间用一个空格隔开，表示从 x 号城市到 y 号城市有一条限载为 z 的道路。注意：x 不等于 y，两座城市之间可能有多条道路。

接下来一行有 1 个整数 q，表示有 q 辆货车需要运货。

接下来 q 行中，每行的 2 个整数 x、y 之间用一个空格隔开，表示一辆货车需要从 x 城市运输货物到 y 城市，注意：x 不等于 y。

对于 30%的数据，$0<n<1000$，$0<m<10\,000$，$0<q<1000$；

对于 60%的数据，$0<n<1000$，$0<m<50\,000$，$0<q<1000$；

对于 100%的数据，$0<n<10\,000$，$0<m<50\,000$，$0<q<30\,000$，$0\leqslant z\leqslant 100\,000$。

输出：

输出共有 q 行，每行一个整数，表示对于每一辆货车，它的最大载重是多少。如果货车不能到达目的地，输出−1。

样例输入：

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

样例输出：

```
3
-1
3
```

题目来源：Codevs 2370。

解题思路：

货车可以通过的权值最小的道路一定是在这个图的最大生成树中，所以先用 Kruskal 算法计算最大生成树，将这个最大生成树再存为一个图，起点和重点的连通性可以通过并查集来判断。然后在这个最大生成树上找这两个节点的 lca，找到两个点之间的最小值，就是货车的最大载货量。

题目实现：

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=100010;
const int maxm=100010;
const int root=1;
const int INF=2147483600;
struct line
{
    int l,r,w;
}li[maxm];
int fa[maxn];
int deep[maxn];
int vis[maxn];
int f[maxn][100];
int lim[maxn][100];
vector<int>a[maxn];
vector<int>b[maxn];
int n,m,q;
int cmp(line a,line b)
{
    if(a.w>b.w) return 1;
    return 0;
}
int getfa(int x)
{
    if(fa[x]==x) return x;
    return fa[x]=getfa(fa[x]);
}
int dfs(int x)
{
    for(int i=0;i<a[x].size();i++)
    if(!vis[a[x][i]])
```

```
33          {
34                  f[a[x][i]][0]=x;
35                  deep[a[x][i]]=deep[x]+1;
36                  lim[a[x][i]][0]=b[x][i];
37                  vis[a[x][i]]=1;
38                  dfs(a[x][i]);
39          }
40  }
41  int init()
42  {
43          for(int i=1;(1<<i)<=n;i++)
44          for(int j=2;j<=n;j++)
45          {
46                  f[j][i]=f[f[j][i-1]][i-1];
47                  lim[j][i]=min(lim[j][i-1],lim[f[j][i-1]][i-1]);
48          }
49  }
50  int lca(int l,int r)
51  {
52          int ans=INF;
53          if(deep[l]>deep[r]) swap(l,r);
54          int len=deep[r]-deep[l];
55          for(int i=0;(1<<i)<=len;i++)
56          {
57                  if((1<<i)&len)
58                  {
59                          ans=min(ans,lim[r][i]);
60                          r=f[r][i];
61                  }
62          }
63          if(l==r) return ans;
64          for(int i=(int)log2(n);i>=0;i--)
65          {
66                  if(f[l][i]!=f[r][i])
67                  {
68                          ans=min(ans,lim[l][i]);
69                          ans=min(ans,lim[r][i]);
70                          l=f[l][i];
71                          r=f[r][i];
72                  }
73          }
```

```
74          ans=min(ans,lim[l][0]);
75          ans=min(ans,lim[r][0]);
76          return ans;
77
78  }
79  int main()
80  {
81          scanf("%d%d",&n,&m);
82          for(int i=1;i<=m;i++) scanf("%d%d%d",&li[i].l,&li[i].r,&li[i].w);
83          for(int i=1;i<=n;i++) fa[i]=i;
84          sort(li+1,li+m+1,cmp);
85          for(int i=1;i<=m;i++)
86          {
87                  int ll=getfa(li[i].l);
88                  int rr=getfa(li[i].r);
89                  if(ll!=rr)
90                  {
91                          a[li[i].l].push_back(li[i].r);
92                          b[li[i].l].push_back(li[i].w);
93                          a[li[i].r].push_back(li[i].l);
94                          b[li[i].r].push_back(li[i].w);
95                          fa[ll]=rr;
96                  }
97          }
98          memset(vis,0,sizeof(vis));
99          vis[root]=1;
100         dfs(root);
101         init();
102         scanf("%d",&q);
103         for(int i=1;i<=q;i++)
104         {
105                 int l,r;
106                 scanf("%d%d",&l,&r);
107                 if(getfa(l)!=getfa(r)) printf("-1\n");
108                 else printf("%d\n",lca(l,r));
109         }
110 }
```

5.6 练习题

习题 5-1

题目来源：HDU 1530。

题目类型：NP 完全问题。

解题思路：最大团问题（Maximum Clique Problem，MCP）是图论中一个经典的组合优化问题，也是一类 NP 完全问题，在国际上已有广泛的研究，而国内对 MCP 问题的研究还处于起步阶段，因此，研究最大团问题具有较高的理论价值和现实意义。本题需要加入最优化剪枝，即保证当前的答案减去现在队中的节点要大于可加入节点的数目。

习题 5-2

题目来源：HDU 2458。

题目类型：NP 完全问题。

解题思路：求最大团。最大团=补图的最大独立集，最小覆盖数+最大独立集=顶点数，在二分图中，最小覆盖数=最大匹配数，显然这里补图一定是一个二分图。

习题 5-3

题目来源：HDU 6187。

题目类型：对偶图。

解题思路：首先可以发现题目就是要求一个平面图对偶图的最小生成树。根据欧拉公式可以知道，对于一个有 k 个连通分量的平面图，区域数 $r=E-V+k+1$，那么对偶图生成树的边数为 $r-1=E-V+k$，这些边也就是要删除原图中的边，那么要留下的边数就是 $V-k$，这刚好就是原图每个连通分量生成树的边数之和。考虑保留原图每个连通分量的生成树，显然满足要求。题目要求花费最小，也就是留下的边权值最大，那么直接对每个连通分量求最大生成树即可，删除的边数就是总边数减去生成树的边数和，最小花费就是全部边的花费减去最大生成树的花费。

习题 5-4

题目来源：HDU 3035。

题目类型：对偶图。

解题思路：最小割问题，但是直接建图进行网络流会超时，该题要利用平面图求最小割的方法，把每一块当成一个点，共有边连边，然后每一个路径就是一个割，最短路径就是最

小割了。

习题 5-5

题目来源：UOJ #50。

题目类型：FFT。

解题思路：多项式的维护，由于 $F'(x)=\frac{1}{2}C(x)F^2(x)+1$，所以需要维护一个平方的多项式，表示 1～区间中点 mid 平方的结果，每次用平方数组更新 f，然后重新新计算 mid+1～r，并把它更新到原来那个平方里去。如何更新呢？提示：$(x+a)^2=x^2+2ax+a^2$。

习题 5-6

题目来源：POJ 1345。

题目类型：ST 表。

解题思路：由于最后剩下的一定是原序列中最大的，于是定义 solve(l,r)表示合并 1 到 r 的最小代价，很显然，最后合并区间的最大值更优，而且最后合并的代价一定是最大值。于是 solve(l,r)=solve(l,p−1)+a[p]+solve(p+1,r)+a[p]，显然这个递归是线性的。需要快速寻找区间最大值，即 ST 表。

习题 5-7

题目来源：POJ 3694。

题目类型：LCA。

解题思路：首先运行一次 Tarjan 算法，求出桥和缩点，那么无向图将缩点为一棵树，树边正好是原来的桥。每次连接两点，看看这两点是不是在同一个缩点内，如果是，那么缩点后的树没任何变化；如果两点属于不同的缩点，那么连接起来，然后找出这两个缩点的 LCA。因为从点 u 到 LCA，再到点 v，最后到点 u，将形成环，所以里面的树边都会变成不是桥。计数的时候应注意，有些树边可能之前已经被标记了，这次再经过时不能再标记。

首先按这个思路编写代码，运行时间为 2 s，这是因为显式建树了。不过如果理解好 Tarjan 算法的话，其实发现并不需要显式建树，可以利用 Tarjan 算法留下的 dfn 和 low 的信息找 LCA。

另外，这里用朴素的方法找 LCA，因为这里找 LCA 就是要找到路径，而且途中有些边是被标记了的。朴素的方法就是在树中记录节点的父亲，然后沿着父亲走回到根去。

修改后运行时间为 1 s 多一点。如果用并查集来缩点，运行时间可降低到 200 ms。

习题 5-8

题目来源：POJ 3417。

题目类型：LCA。

解题思路：先给出一棵无根树，再给出 m 条边，把这 m 条边连上，每次能毁掉两条边，规定一条是树边，一条是新边，问有多少种方案能使树断裂。

这 m 条边连上后这棵树必将成环，假设新边为（u，v），那么环为 u→LCA(u,v)→v→u，给这个环上的边计数 1，表示这些边被一个环覆盖了一次。添加了多条新边后，可知树上有些边是会被多次覆盖的，画图很容易发现，如果一个树边被覆盖了 2 次或以上，它就是一条牢固的边，也就是说，毁掉它再毁掉任何一条新边都不会使树断裂，这个结论也是很容易证明的，画图更明显，所以不赘述。

受到启发后，要统计所有的边被覆盖了几次，可分以下几种情况来讨论：

（1）覆盖 0 次：说明这条边不在任何一个环上，这样的边最脆弱，仅仅毁掉它就可以使树断裂了，这时候只要任意选一条新边并毁掉，树就断裂，所以就产生 m 种方案（m 为新边条数）。

（2）覆盖 1 次：说明这条边在一个环上，并且仅在一个环上，那么要使树断裂，就需要毁掉这条树边，并且毁掉和它对应的那条新边（毁其他的新边无效）。这种树边能产生的方案数为 1，一条这样的树边只有唯一解。

（3）覆盖 2 次或以上：无论怎么样都不能使树断裂，产生的方案数为 0。

所以，如果知道所有的树边的覆盖情况，那么统计一次就行了，所以剩下的问题是每条边被覆盖了几次？

这就需要用到树 DP。首先定义 dp[u]的意义为，u 所对应的那条父边（u 和它父亲连接的那条边）被覆盖的次数。对应一条新边（u，v），要计算 LCA(u,v)，则需要计数 dp[u]++、dp[v]++、dp[lca]−=2。为什么要这样计数呢？因为点 u、点 v 和点 lca 都试着沿路径一直回到树根处（注意不是回到 LCA 而是树根），u 的路径中每经过一个点，就将这些点上的值加到 dp[u]；同样，v 的路径上每经过一个点也要将这些点上的值加到 dp[v]。lca 也是这样，lca 回到树根的部分，其实是被抵消掉了，dp 值没有变化。而 u 到 lca，以及 v 到 lca 部分的值都已经分别加到了 dp[u],dp[v]，因此在求完所有 m 对顶点的 LCA 后，每个 u 和 v 都进行一次 dp[u]++、dp[v]++、dp[lca]−=2，然后从树根开始向下遍历一次整棵树，在回溯的时候就执行累加 dp[u]、dp[v]的操作，这其实就是树 DP 的过程。

第 6 章

组合数学

组合数学是离散数学的一部分，在程序设计中，组合数学主要研究的是集合中个体可重复或不可重复的有序或无序的安排问题。这些问题在生活中很普遍，所以研究这些问题的解法十分重要。

本章首先介绍排列组合的相关概念与基本算法，然后介绍母函数、整数划分、Stirling 数和 Catalan 数、容斥原理和反演，以及群论和 Polya 定理的相关概念与算法，并对某些应用给出了具体实现。

6.1 排列组合

从指定元素个数的集合中取出特定数量的元素，并进行排序，得到其方案数的方法称为排列；若不考虑其顺序，则得到方案数的方法称为组合。本节将从计数原则入手，着重介绍排列组合的相关定理及算法，但不详细介绍定理的证明。

6.1.1 基本计数原则

乘积法则 假定一个过程可以被分解成两个任务，如果完成第一个任务有 n 种方式，完成第一个任务后有 m 种方式完成第二个任务，那么完成这个过程就有 nm 种方式。

求和法则 如果完成第一项任务有 n 种方式，完成第二项任务有 m 种方式，并且这些任务不能同时执行，那么完成这个过程有 $n+m$ 种方式。

阶乘 从 1 开始，按照自然数顺序乘到 n，即 $1\times2\times3\times\cdots\times n$，记为 $n!$，其中，0!=1。

6.1.2 排列

有 n 个不同元素，任取 r 个元素的排列数是

$$P_n^r = n(n-1)(n-2)\cdots(n-r+1) = \frac{n!}{(n-r)!}$$

例 字母 ABCDEFGH 的所有排列中有多少种排列包含 ABC？

解 由于 ABC 必须成组出现，只需通过求 ABC、D、E、F、G、H 六个对象的排列数即可得到答案。因此，答案为 6!=720 种。

6.1.3 组合

有 n 个不同元素，从中取出 r 个元素的所有方案数为

$$C_n^r = \frac{n!}{r!(n-r)!}$$

在某些书籍中，也将 C_n^r 记为 $\binom{n}{r}$。为书写方便，以下简记为 $C(n,r)$。

定理 6.1 $C(n,r)=C(n,n-r)$。

定理 6.2 **二项式定理** 设 x 和 y 是变量，n 是非负整数，那么

$$(x+y)^n = \binom{n}{0}x^n + \binom{n}{1}x^{n-1}y + \ldots + \binom{n}{n-1}xy^{n-1} + \binom{n}{n}y^n$$

定理 6.3 **帕斯卡恒等式** $C(n+1,r)=C(n,r-1)+C(n,r)$

帕斯卡三角（又称为杨辉三角）是求组合数的一种简便的方式，其具体构成如下：

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

其中，第二行开始两侧的数为 1，其余的数为在这个数上面的两个数的和。n 行第 m 列的数的值为 $C(n-1,m-1)$。于是，可以得到一种当 n 和 m 较小时求组合数的简便方法，算法时间复杂度为 $O(nm)$，因为仅涉及加法，所以对于取模操作同样适用。实现代码如下：

```
1   int c[15][15];
2   c[0][0]=1;
3   for(int i=1;i<=10;i++){
4       c[i][0]=c[i][i]=1;
5       for(int j=1;j<i;j++)
6       c[i][j]=c[i-1][j-1]+c[i-1][j];
7   }
```

6.1.4　例题讲解

例 6-1　Wall Painting

Time Limit: 10000/5000 ms (Java/Others)　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

方女士很喜欢画画，她每天要画 GFW（有趣的长城）。在每天画画之前，她要将水和一些颜料混合来得到美丽的颜色。在第 K 天，她会将 K 种不同的颜料混合作为当天所使用的颜色。若她将颜色 A 和颜色 B 混合，她将得到颜色 A xor B。

当她将两种相同的颜料混合时，她就会因一些奇怪的原因得到 0。她的丈夫方先生不知道她会在第 K 天选哪 K 种颜色，他想要知道通过 $\binom{n}{K}$ 种不同的方案，方女士能够得到多少种颜色。

例如，假定 n=3、K=2，三个颜料分别为 2、1、2，她可以得到 3、3、0 三个颜色。在这种情况下，方先生可以知道第二天的数是 3+3+0=6。

方先生太忙了，所以他不想把时间花在这上面，你能帮帮他吗？

你需要告诉方先生从第一天到最后一天中每天的答案。

输入：

有多组数据，由 EOF 结尾。对于每组数据，第一行是 N（$1 \leqslant N \leqslant 10^3$）。第二行包含 N 个数，第 i 个数代表第 i 个包中的颜料的颜色。

输出：

对于每组数据，在一行内输出 N 个整数（mod 10^6 +3），代表第一天到最后一天的答案。

样例输入：

```
4
1 2 10 1
```

样例输出：

```
14 36 30 8
```

题目来源：HDU 4810。

解题思路：

给出 n 个数，并且有 n 天，第 k 天从中任取 k 个数异或，并将所有这样的组合相加后输出，共输出 n 个数。这道题由组合与异或两部分组成。因为 $n \leqslant 1000$，所以可以先打表求出组合数；对于第 k 天的方案，若直接求组合会超时，就需要用到异或操作的特性。当两个数或多个数进行异或操作时，任意某个二进制位的异或操作不会对其他位造成影响，因此可以将每个数转换成二进制并将不同数位上的数保存起来。若 k 个数中的某位有 i 个数是 1，则有 $n-i$ 个数的该位为 0。第 k 天某种组合需要从这位中取 j 个 1，则这种组合的所有取法的和为 $C(i,j)$

×$C(n-i,k-j)$× (j%2==0)（若有偶数个 1，则这种组合的所有取法之和自然为 0）。

题目实现：

```
#include<cstdio>
#include<cstring>
#define P 1000003
int n,c[1005][1005],sum[40],a,ans[1005],t;
int main(){
    for(int i=0;i<=1000;++i)
        c[i][0]=c[i][i]=1;
    for(int i=1;i<=1000;++i)
        for(int j=1;j<i;++j)
            c[i][j]=(c[i-1][j]+c[i-1][j-1])%P;
    while(scanf("%d",&n)!=EOF){
    memset(sum,0,sizeof(sum));
    memset(ans,0,sizeof(ans));
    for(int i=1;i<=n;++i){
        scanf("%d",&a);
        for(int j=0;a;++j,a>>=1)if(a%2)sum[j]++;
    }
    for(int k=1;k<=n;k++)
        for(int i=0;i<32;i++)
            for(int j=1;j<=sum[i]&&j<=k;j+=2){
                t=(long long)c[sum[i]][j]*c[n-sum[i]][k-j]%P*(1<<i)%P;
                ans[k]=(ans[k]+t) %P;
            }
    for(int k=1;k<=n;k++)
        printf("%d%c",ans[k],k<n?' ':'\n');
    }
}
```

6.2 母函数

母函数又称为生成函数，是一种表示序列的有效方法，它把序列的项作为幂级数中变量 x 的幂的系数。可以用母函数求解许多类型的计数问题，例如在各种限制下选取或分配不同种类物体的方式数，或用不同面额的硬币换取较大数额纸币的方式数等。本节将主要介绍母函数与计数问题的关系，以及如何利用普通型母函数及指数型母函数求解一些计数问题。

6.2.1　母函数基础

定义 6.1　实数序列 $a_0,a_1,\cdots,a_k,\cdots$的母函数是无穷级数：

$$G(x)=a_0+a_1x+a_2x^2+\cdots+a_kx^k+\cdots=\sum_{k=0}^{\infty}a_kx^k$$

例如，序列$\{a_k\}$（$a_k=3,a_k=k+1,a_k=2^k$）的生成函数分别是

$$\sum_{k=0}^{\infty}3x^k,\qquad \sum_{k=0}^{\infty}(k+1)x^k,\qquad \sum_{k=0}^{\infty}2^kx^k$$

定理 6.4　令 $f(x)=\sum_{k=0}^{\infty}a_kx^k$，$g(x)=\sum_{k=0}^{\infty}b_kx^k$，那么

$$f(x)+g(x)=\sum_{k=0}^{\infty}(a_k+b_k)x^k,\qquad f(x)g(x)=\sum_{k=0}^{\infty}\left(\sum_{j=0}^{k}a_jb_{k-j}\right)x^k$$

定义 6.2　对于实数序列 $a_0,a_1,\cdots,a_k,\cdots$，函数

$$G_e(x)=a_0+\frac{a_1}{1!}x+\frac{a_2}{2!}x^2+\cdots+\frac{a_k}{k!}x^k+\cdots=\sum_{k=0}^{\infty}\frac{a_k}{k!}x^k$$

称为实数序列 $a_0,a_1,\cdots,a_k,\cdots$对应的指数型母函数。

6.2.2　母函数的两类具体应用

先从一道例题谈起。

例 1：有若干个 1 元、2 元、5 元的纸币，若用这些纸币换 1 张 10 元纸币，问有多少种方案。所谓两种方案不同，是指当且仅当两种方案中有不同数量的纸币或至少有一种面额的纸币数量不同。

解：答案是 10。如果学过背包问题的解法，可以看出这道题是简单的完全背包问题。如果没学过背包，因这道题中数值较小，同样可以通过笔算枚举得出。现在换个思路：首先，若只使用面值为 1 元的纸币，那么 0、1、2、3、…、10 元纸币用 1 元纸币兑换各只有 1 种方案；仅用面值为 2 元的纸币，则可取 0、2、4、6、8、10 元；仅用面值为 5 元的纸币，则可取 0、5、10 元。这时已经得到了三个序列，用母函数的定义可以得到三个母函数：

$$f(x)=1+x^1+x^2+\cdots+x^{10}$$
$$g(x)=1+x^2+x^4+\cdots+x^{10}$$
$$h(x)=1+x^5+x^{10}$$

然后利用定理 6.1 中多项式乘法并提取 10 次幂的系数，即可得到答案。

例 1 就是一道基本的利用母函数求解个数的题目。在求解这样的题目时，可以把题目信息转化成形如 $e_1+e_2+\cdots+e_n=C$ 的方程，其中每个 e_i 是可能具有某些约束的非负整数，于是在求 e 的可行解的数目时可以使用母函数。注意到其中一步是多项式乘法，因此在某些极端情

况下甚至会有使用快速傅里叶变换解题的可能。

再看另一道例题。

例 2：假设有 8 个元素，其中 a_1 重复 3 次，a_2 重复 2 次，a_3 重复 3 次。从中取 r 个进行排列，求其排列数。

解：这题需要求解的是排列数，与例 1 求组合数的方法略有不同。首先，利用定义 6.2 对 a_1、a_2、a_3 分别构建指数型母函数：

$$f(x)=1+x^1+x^2/2!+x^3/3!$$
$$g(x)=1+x^1+x^2/2!$$
$$h(x)=1+x^1+x^2/2!+x^3/3!$$

然后利用多项式乘法将它们相乘，得到一个新的式子，即

$$r(x)=1+\frac{3}{1!}x^1+\frac{9}{2!}x^2+\frac{26}{3!}x^3+\frac{70}{4!}x^4+\frac{170}{5!}x^5+\frac{350}{6!}x^6+\frac{560}{7!}x^7+\frac{560}{8!}x^8$$

最后一步，如果题目求 4 个的排列，就将 x^4 的系数乘以 x 的指数的阶乘（即 4!），得到答案是 70。

或许有的人会不明白，为什么在求排列数时使用的是指数型母函数呢？在求有重复元素的排列时，若某个元素在排列中重复 k 次，则需要将原式求出的结果除以 k!，而在这里将原序列转化成指数型母函数的操作就相当于去重操作。最后再乘以取个数的阶乘，则是将这些被当成没有重复元素的物体进行全排列的过程。

以上两题分别介绍了普通型母函数及指数型母函数的解法，下面就从实际入手，熟练掌握这两种解法。

6.2.3 例题讲解

例 6-2 Holding Bin-Laden Captive

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 65536/32768 KB (Java/Others)

题目描述：

给你一些人民币（面值为 1 元、2 元、5 元），并且知道这三种面值的数量为 num_1、num_2 和 num_5，算出你最少无法支付的价值。

作为一个超级 ACM 选手，需要十分容易地解决这道问题，还有不要忘了带走 25 000 000 美元的奖金！

输入格式：

输入包含多组测试数据。每组测试数据包含 3 个不超过 1000 的非负整数 num_1、num_2、num_5。若某组数据的三个数都为 0，则读入结束，这组数据不需要输出。

输出格式：

输出你最少不能支付的价值，每组测试数据一行。

样例输入：

1 1 3

0 0 0

样例输出：

4

题目来源：HDU 1085。

解题思路：

类似 6.2.2 节中例 1 的解法，并且因为此题仅仅是求最小的未出现的价值，所以更简单。若使用背包算法，因为每种人民币数量小于或等于 1000，因此 10000 的数组已经足够。

题目实现：

```
#include<cstdio>
#include<cstring>
int x,y,z,a[10005],b[10005],c[10005],l;
int main(){
    while(scanf("%d%d%d",&x,&y,&z)&&x+y+z){
        memset(a,0,sizeof(a));
        memset(b,0,sizeof(b));
        memset(c,0,sizeof(c));
        for(int i=0;i<=x;i++)a[i]=1;
        for(int i=0;i<=y;i++)b[i*2]=1;
        l=x+y*2;
        for(int i=0;i<=x;i++)
            for(int j=0;j<=y*2;j++)
                c[i+j]+=(a[i]*b[j]);
        for(int i=0;i<=l;i++)a[i]=c[i];
        memset(b,0,sizeof(b));
        memset(c,0,sizeof(c));
        for(int i=0;i<=z;i++)b[i*5]=1;
        for(int i=0;i<=l;i++)
            for(int j=0;j<=z*5;j++)
                c[i+j]+=(a[i]*b[j]);
        l=l+z*5;
        for(int i=0;i<=l+1;i++)
            if(!c[i]){printf("%d\n",i);break;}
    }
}
```

例 6-3　排列组合

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 65536/32768 KB (Java/Others)

题目描述：

有 n 种物品，并且知道每种物品的数量。要求从中选出 m 件物品的排列数。例如，有两种物品 A、B，并且数量都是 1，从中选 2 件物品，则排列有 AB、BA 两种。

输入格式：

每组输入数据有两行，第一行是 2 个数 n、m（$1 \leqslant m$、$n \leqslant 10$），表示物品数；第二行有 n 个数，分别表示这 n 件物品的数量。

输出格式：

对应每组数据输出排列数（任何运算不会超出 2^{31} 的范围）。

样例输入：

2 2

1 1

样例输出：

2

题目来源：HDU 1521。

解题思路：

类似例 2 的解法，在此就不再赘述了。

题目实现：

```
#include<cstdio>
#include<cstring>
int n,m,s[15],q;
double a[15],b[15],c[15];
int main(){
    while(scanf("%d%d",&n,&m)!=EOF){
        memset(a,0,sizeof(a));a[0]=1;
        q=1;
        for(int i=1;i<=m;i++)q=q*i;
        for(int i=1;i<=n;i++){
            scanf("%d",&s[i]);
            memset(b,0,sizeof(b));
            memset(c,0,sizeof(c));
            b[0]=1;
            for(int j=1;j<=s[i];j++){
                b[j]=b[j-1]/j;
            }
            for(int j=0;j<=m;j++)
                for(int k=0;k<=m-j;k++)
                    c[j+k]+=a[j]*b[k];
```

```
21                  for(int j=0;j<=m;j++)a[j]=c[j];
22              }
23              printf("%.0lf\n",a[m]*q);
24          }
25 }
```

6.3　整数划分

定义 6.3　对于某个整数 n，若存在一组正整数序列 $a_1,a_2,a_3,\cdots,a_k$，使得 $a_1+a_2+a_3+\cdots+a_k=n$，则称$\{a_1,a_2,a_3,\cdots,a_k\}$为 n 的一个划分；若 $a_1,a_2,a_3,\cdots,a_k$ 中最大值不超过 m，则称$\{a_1,a_2,a_3,\cdots,a_k\}$为 n 的一个 m 划分，记为 $f(n,m)$。

例如，n=5 时有以下几种划分：

5=5
5=4+1
5=3+2
5=3+1+1
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1

对于某个整数，一般需要知道它的划分数，以及具体的划分方案。对于划分方案，可以使用递归的方法，在这就不详细阐述了；重点需要知道的是划分数的解法。

6.3.1　从动态规划到母函数

求解整数划分的一种有效的方法，就是动态规划。求解动态规划的题目，最重要的是求出题目的状态转移方程。先假设有这么一道题：给定一个整数 n，求出它的 m 划分的方案数。

根据 n 和 m 的关系，考虑以下情况：

当 n=1 时，对于任意 m 划分数都为 1，即 $f(1,m)$=1；

当 m=1 时，对于任意 n 划分数也只有 1 种$\{1,1,1,\cdots1\}$，即 $f(n,1)$=1；

当 $m>n$ 时，$f(n,m)=f(n,n)$，因为若在某个划分中存在大于 n 的数，那么这个划分必存在负数；

当 $m\leqslant n$ 时，可以将划分分成含有 m 以及不含 m 的情况，那么 $f(n,m)$可以表示成 $f(n-m,m)+f(n,m-1)$。因为 $m=n$ 时，含有 m 的情况仅有 1 种划分，所以约定 $f(0,m)$=1。

总结以上四种情况，即可解出 n 的 m 划分的方案数。

再看到以下几种特殊情况：若要求每个划分中不包含相同的数，那么划分数有多少种？若要求每个划分中仅包含奇数，那么划分数又有多少种？

若每个划分中不包含相同的数，即可将 $f(n,m)$的状态转移方程改为 $f(n,m)=f(n-m,m-1)+f(n,m-1)$，表示若取 m，则剩下的数要从 $m-1$ 中取。若每个划分中仅包含奇数，约定 m 为奇数，则 $f(n,m)=f(n-m,m)+f(n,m-2)$。可以看到，动态规划在求解整数划分时很简便，但有些难以理解。

接下来就介绍一下母函数的解法。可以把 n 的 m 划分问题看成如何用 1 到 m 这些整数凑出 n，很显然这是一道母函数的经典模型。令 $f_k(x)=1+x^k+x^{2k}+\cdots+x^{nk}+\cdots$，将 $f_1,f_2,\cdots,f_m$ 用多项式乘法相乘，其中次数为 n 的 x 的系数即 n 的 m 划分的方案数。若要求每个划分中不包含相同的数，则令 $f_k(x)=1+x^k$，再用多项式乘法相乘即可；同样只取奇数的方法也很简单。因为涉及多项式乘法，所以虽然比动态规划要容易理解，但代码量上比起动态规划会更多。

6.3.2 例题讲解

例 6-4 Ignatius and the Princess III

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 65536/32768 KB (Java/Others)

题目描述：

给出一个正整数 N，定义下面的等式：

$$N=a[1]+a[2]+a[3]+\cdots+a[m],\qquad a[i]>0,1\leqslant m\leqslant N$$

问题是：对于给定的 N，能找到多少种不同的等式。

例如，假定 N 是 4，我们可以找到：

4 = 4
4 = 3 + 1
4 = 2 + 2
4 = 2 + 1 + 1
4 = 1 + 1 + 1 + 1

所以当 N=4 时，结果为 5，注意“4 = 3 + 1”和“4 = 1 + 3”在这个问题中是相同的等式。

输入：

输入包含多组测试数据，每组测试数据都包含一个小于或等于 120 的正整数。输入由 EOF 结束。

输出：

对于每组测试数据，需要输出一个整数 P，代表找到的所有不同的等式。

样例输入：

4
10
20

样例输出：

5

42

627

题目来源：HDU 1028。

解题思路：

题目描述就是基本的整数划分，在这将使用动态规划及母函数两种实现方法。

题目实现 1：

```
#include<cstdio>
#include<cstring>
int n,f[125][125];
int main(){
    for(int i=1;i<=120;i++)f[1][i]=f[i][1]=f[0][i]=1;
    for(int i=2;i<=120;i++){
        for(int j=2;j<=i;j++)
            f[i][j]=f[i-j][j]+f[i][j-1];
        for(int j=i+1;j<=120;j++)
            f[i][j]=f[i][i];
    }
    while(scanf("%d",&n)!=EOF){
        printf("%d\n",f[n][n]);
    }
}
```

题目实现 2：

```
#include<cstdio>
#include<cstring>
long long a[300],b[300],c[300];
int n;
int main(){
    for(int i=0;i<=120;i++)
        a[i]=1;
    for(int i=2;i<=120;i++){
        memset(b,0,sizeof(b));
        for(int j=0;j*i<=120;j++)
            b[j*i]=1;
        for(int j=0;j<=120;j++)
            for(int k=0;k<=120;k++)
                c[j+k]+=(a[j]*b[k]);
```

```
15              for(int i=0;i<=120;i++){
16                  a[i]=c[i];
17                  c[i]=0;
18              }
19          }
20          while(scanf("%d",&n)!=EOF){
21              printf("%I64d\n",a[n]);
22          }
23 }
```

6.4 Stirling 数和 Catalan 数

问题 1：将 n 个各不相同的物品排成 k 个非空循环排列，问有多少种方案。

问题 2：将 n 个各不相同的物品放入 m 个无差异的盒子中，问有多少种方案。

问题 3：有一个凸 n 边形，通过连接两两不交叉的对角线（可以交于顶点），将其分成 $n-2$ 个三角形，问有多少种分割的方案。

问题 1 的答案构成的序列被称为第一类 Stiring 数，问题 2 的答案构成的序列被称为第二类 Stiring 数，而问题 3 的答案构成的序列被称为 Catalan 数。以上三种都是由递推得到的序列。下面将从递推出发，得到这三个序列的递推式，并且用它们求解一些具体的题目。

6.4.1 第一类 Stirling 数

n 个各不相同的物品的 k 个非空循环排列的方案数记为 $s(n,k)$，其中，$k \leqslant n$。

记 $s(n,0)=0$；若将 n 个各不相同的物品分成 n 个非空循环排列，则每个物品都是一个排列，方案数则为 1，所以，$s(n,n)=1$；

对于某个小于 n 的 k，$s(n,k)$可以由 $s(n-1,k-1)$及 $s(n-1,k)$转化而来。若前 $n-1$ 个物品构成 $k-1$ 个非空循环排列的方案数为 $s(n-1,k-1)$，则第 n 个物品构成一个排列，方案数为 $s(n-1,k-1)$；若前 $n-1$ 个物品构成 k 个非空循环排列数为 $s(n-1,k)$，则第 n 个物品可以插入前 $n-1$ 个物品的左边，方案数为$(n-1)\times s(n-1,k)$。因此，$s(n,k)=s(n-1,k-1)+(n-1)\times s(n-1,k)$。

这里出现的都是无符号第一类 Stirling 数，它有以下性质：

（1）$s(0,0)=1$。

（2）$s(n,0)=0$（n 不为 0）。

（3）$s(n,n)=1$。

（4）$s(n,1)=(n-1)!$。

（5）$s(n,n-1)=C_n^2$。

（6）$s(n,2)=(n-1)!\sum_{i=1}^{n-1}\frac{1}{i}$。

（7）$s(n,n-2)=2C_n^3+3C_n^4$。

（8）$\sum_{k=1}^{n}s(n,k)=n!$。

6.4.2　第二类 Stirling 数

n 个各不相同的物品放入 m 个无差异盒子的方案数记为 $S(n,m)$，其中，$m\leqslant n$。

记 $S(n,0)=0$；将 n 个各不相同物品放入 n 个无差异的盒子中，则每个盒子里各有 1 个物品，而且方案数只有一种，因此，$S(n,n)=1$。

与第一类 Stirling 数类似，假定已知 $S(n-1,m-1)$与 $S(n-1,m)$。若已知前 $n-1$ 个物品放入 $m-1$ 个盒子，第 n 个物品则可以放入第 m 个盒子；若前 $n-1$ 个物品已放入 m 个盒子，则第 n 个物品可放入 m 个盒子中的任意一个。因此，$S(n,m)=S(n-1,m-1)+m\times S(n-1,m)$。

它同样具有一些性质：

（1）$S(0,0)=1$。

（2）$S(n,0)=0$（n 不为 0）。

（3）$S(n,1)=1$。

（4）$S(n,n)=1$。

（5）$S(n,2)=2^{n-1}-1$。

（6）$S(n,n-1)=C_n^2$。

（7）$S(n,n-2)=C_n^3+3C_n^4$。

（8）$S(n,3)=\frac{1}{2}(3^{n-1}+1)-2^{n-1}$。

对于两类 Stirling 数，它们满足以下关系：

$$\sum_{k=0}^{n}S(n,k)s(k,m)=\sum_{k=0}^{n}s(n,k)S(k,m)$$

6.4.3　Catalan 数

给定一个凸多边形，如图 6.1 所示是一个凸六边形，并选取一条边（图中的粗边）作为基边。显然，这条边将是划分之后某个三角形的一边。选取一个顶点（如 B），将区域分为 P_1、P_2、P_3，可以看出，左边一块有 3 边，右边一块有 4 边。依次选取 A、B、C、D 四个顶点，六边形会被分成不同的区域。记 n 边形分成三角形的方案数为 $t(n)$，则 $t(n)=t(2)\times t(n-1)+t(3)\times t(n-2)+\cdots+t(n-1)\times t(2)$，如图 6.2 所示。

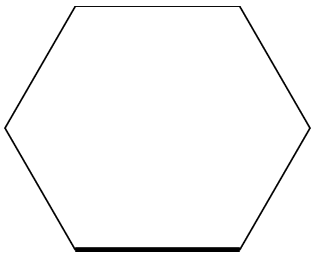


图 6.1　凸六边形

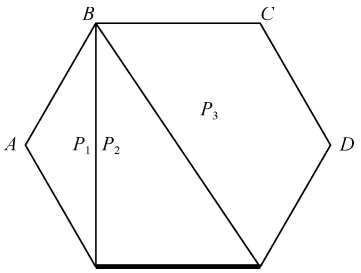


图 6.2　凸六边形的分割

令 $C(n)=t(n+2)$，则 $C(n)=C(0)\times C(n-1)+C(1)\times C(n-2)+\cdots+C(n-1)\times C(0)$。这就是 Catalan 数的递推式。只要满足这个递推式的序列，就是 Catalan 数。同时可以推得 Catalan 数的通项公式为

$$C(n)=\binom{2n}{n}-\binom{2n}{n-1}=\frac{1}{n+1}\binom{2n}{n}$$

6.4.4　例题讲解

例 6-5　Count the Buildings

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB (Java/Others)

题目描述：

由 N 座建筑排列成一列立在城市中，标号分别为 1 到 N，所有的建筑的高度都不相同，并且高度范围在 1 到 N。当站在第一座建筑前边往后看，能看到 F 座建筑，当站在最后一座建筑后边往前看，能看到 B 座建筑。只有在你与一座建筑之间没有比该建筑高的建筑时才能被看见该建筑。

现在，给定 N、F、B，你的任务是找出这些建筑的排列方案有多少种。

输入：

第一行包含一个整数 T（$T\leqslant 100000$），表示接下来会有 T 组测试数据。接下来 T 行，每行包含 3 个整数 N、F、B（$0<N, F, B\leqslant 2000$），描述如上。

输出格式：

对于每组数据，需要输出排列的方法数并对 1e9+7 取模。

样例输入：

2

3 2 2

3 2 1

样例：

2

1

题目来源：HDU 4372。

解题思路：

首先，最高的建筑必然能被看到，也必然是从前或者从后看到的最后一座建筑。从前或者从后看到的建筑必然是单调递增的，不妨将每个能被看到的建筑到下个能被看到的建筑之前的所有建筑看成一组，那么除去最高的建筑，从前面看有 $f-1$ 组，后面有 $b-1$ 组。每组除最高的建筑外，其余建筑可以随意排列，那么每组建筑可以看成一个环。问题就转换成如何从 $n-1$ 个物体里选出 $f-1+b-1$ 个环，也就是第一类 Stirling 数。在选出这些环之后，再取其中的 $f-1$ 个放到最高的建筑的前面，答案为 $s(n-1,f+b-2)\times C(f+b-2,f-1)$。

题目实现：

```
#include<cstdio>
#include<cstring>
long long P=1000000007;
int T,n,f,b;
long long s[2005][2005];
long long c[2005][2005];
int main(){
    s[0][0]=1;
    for(int i=1;i<=2000;i++)s[i][0]=0,s[i][i]=1;
    for(int i=2;i<2000;i++)
        for(int j=1;j<i;j++)
            s[i][j]=(s[i-1][j-1]+s[i-1][j]*(i-1))%P;
    c[0][0]=1;
    for(int i=1;i<=2000;i++)c[i][0]=c[i][i]=1;
    for(int i=2;i<2000;i++)
        for(int j=1;j<i;j++)
            c[i][j]=(c[i-1][j]+c[i-1][j-1])%P;
    scanf("%d",&T);
    while(T--){
        scanf("%d%d%d",&n,&f,&b);
        if(f+b-1>n)printf("0\n");
        else printf("%lld\n",s[n-1][f+b-2]*c[f+b-2][f-1]%P);
    }
}
```

例 6-6　一卡通大冒险

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit:32768/32768 KB (Java/Others)

题目描述：

因为长期钻研算法，无暇顾及个人问题，BUAA ACM/ICPC 训练小组的帅哥们大部分都是单身。某天，他们在机房商量一个绝妙的计划“一卡通大冒险”。这个计划的内容是，把自己的联系方式写在校园一卡通的背面，然后故意将自己的卡“遗失”在某处（如水房、食堂），他们希望能有美女看到他们遗失的卡，能主动跟他们联系，这样就有机会请美女吃饭了。他们决定将自己的一卡通夹在几本相同的书里，然后将书遗失到校园的各个角落。正当大家为这个绝妙的计划叫好时，突然想到一个问题。很明显，如果只有一张一卡通，那么只有一种方法，即将其夹入一本书中；当有两张一卡通时，就有了两种选择，即将两张一卡通夹在一本书里，或者分开夹在不同的书里；当有三张一卡通时，他们就有了 5 种选择，即：

{{A},{B},{C}} , {{A,B},{C}}, {{B,C},{A}}, {{A,C},{B}} ,{{A,B,C}}

于是，他们希望了解，如果 ACM 训练小组里有 n 位帅哥（即有 n 张一卡通），那么要把这些一卡通夹到书里的话，有多少种不同的方法。

输入格式：

包含多组数据，第一行为 n，表示接下来有 n 组数据。以下每行有一个数 x，表示共有 x 张一卡通（$1 \leqslant x \leqslant 2000$）。

输出格式：

对每组数据分别输出一行，表示不同的方法数，因为这个数可能非常大，只显示它除以 1000 的余数。

样例输入：

4
1
2
3
100

样例输出：

1
2
5
751

题目来源：HDU 2512。

解题思路：

有 n 个物体，它们可以放在任意的相同的盒子里，但要保证这些盒子非空。问有多少种方案。很明显是第二类 Stirling 数，只是需要将 1 到 n 个盒子的所有方案全部加起来。这个方案总数也称为贝尔（Bell）数。

题目实现：

```
#include<cstdio>
#include<cstring>
int s[2005][2005],sum[2005];
int T,n;
int main(){
    s[0][0]=1;
    for(int i=1;i<=2000;i++)s[i][0]=0,s[i][i]=1,sum[i]=1;
    for(int i=2;i<=2000;i++)
        for(int j=1;j<i;j++){
            s[i][j]=(s[i-1][j-1]+j*s[i-1][j])%1000;
            sum[i]=(sum[i]+s[i][j])%1000;
        }
    scanf("%d",&T);
    while(T--){
        scanf("%d",&n);
        printf("%d\n",sum[n]);
    }
}
```

例 6-7　How Many Trees

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit:65536/32768 KB (Java/Others)

题目描述：

一个以 k 为根的二叉搜索树（寻找树）是指在根左边的所有节点都比 k 小并且根右边的所有节点都比根大的二叉树。对于一棵节点数为 n 的二叉搜索树来说，这是一种能在平均为 $O(n\log n)$的时间复杂度内找到任意节点 x 的数据结构。

给出一个数 n，你能计算出全部用上这 n 个节点并且只用一次可以构成多少棵结构不同的二叉搜索树吗？

输入：

每行包含一个 i（$1\leqslant i\leqslant 100$），代表树有 i 个节点。

输出：

每行需要输出不同输入的答案。

样例输入：

1

2

3

样例输出：

1

2

5

题目来源：HDU 2512。

解题思路：

已知一棵二叉树的节点个数，问这棵二叉树共有多少种形态。

选某个节点作为根，那么左子树节点个数为 0 到 $n-1$ 个，右子树与左子树节点个数之和为 $n-1$。每棵子树的形态已在之前求出，很明显它是 Catalan 数，因为 n 比较大，所以需要用到大整数高精度运算算法。做法是用推导公式求解，但用组合数求解的话会更简单。

题目实现：

```
#include<cstdio>
#include<cstring>
#define max(a,b) (a>b?a:b)
struct poi{
    int a[105],l;
}f[105];
poi operator *(poi a,poi b){
    poi c;
    c.l=a.l+b.l-1;
    memset(c.a,0,sizeof(c.a));
    for(int i=1;i<=a.l;i++)
        for(int j=1;j<=b.l;j++){
            c.a[i+j-1]=c.a[i+j-1]+a.a[i]*b.a[j]+c.a[i+j-2]/10;
            c.a[i+j-2]%=10;
        }
    if(c.a[c.l]>9)c.a[++c.l]=c.a[c.l-1]/10,c.a[c.l-1]%=10;
    return c;
}
poi operator +(poi a,poi b){
    a.l=max(a.l,b.l);
    for(int i=1;i<=a.l;i++){
        a.a[i]=a.a[i]+b.a[i]+a.a[i-1]/10;
        a.a[i-1]%=10;
    }
    if(a.a[a.l]>9){a.a[++a.l]=a.a[a.l-1]/10;a.a[a.l-1]%=10;}
    return a;
}
```

```
28  void write(poi a){
29      for(int i=a.l;i;i--){
30          printf("%d",a.a[i]);
31      }
32      printf("\n");
33  }
34  int main(){
35      f[0].l=1;f[0].a[1]=1;
36      f[1].l=1;f[1].a[1]=1;
37      for(int i=2;i<=100;i++){
38          f[i].l=0;memset(f[i].a,0,sizeof(f[i].a));
39          for(int j=0;j<=i-1;j++){
40              f[i]=f[i]+f[j]*f[i-1-j];
41          }
42      }
43      int n;
44      while(scanf("%d",&n)!=EOF){
45          write(f[n]);
46      }
47  }
```

6.5　容斥原理与反演

一个班级里有 50 个学生，其中 25 人学习英语，20 人学习俄语，既学习英语又学习俄语的有 10 人，问有多少学生这两门语言都没有学习。

上面就是容斥原理的一个简单应用。接下来将从容斥原理的概念开始，并介绍反演的理论及其应用，以求解更广泛的计数问题。

6.5.1　容斥原理

有集合 A、B，用$|A|$表示集合 A 中元素的个数，那么集合 A 与集合 B 的并集的元素个数为

$$|A \cup B| = |A| + |B| - |A \cap B|$$

现在尝试用这个方法求解上面的题目。令集合 A 为学习英语的人的集合，集合 B 为学习俄语的人的集合，那么选了这两种语言中任意一种的人的集合大小为 25+20−10=35 人，这两种语言都没选的人的集合为 50−35=15 人。

接下来看三集合并的情况。有集合 A、B、C，集合 A 中包含 $A \cap B$、$A \cap C$、$A \cap B \cap C$，B、C 类似，而在 $A \cap B$、$A \cap C$、$B \cap C$ 中都包含了集合 $A \cap B \cap C$，那么 $A \cup B \cup C$ 可以表示成 $(A - A \cap B - A \cap C + A \cap B \cap C) + (B - A \cap B - B \cap C + A \cap B \cap C) + (C - A \cap C - B \cap C + A \cap B \cap C) +$

$(A\cap B-A\cap B\cap C)+(B\cap C-A\cap B\cap C)+(A\cap C-A\cap B\cap C)+A\cap B\cap C$，于是

$$|A\cup B\cup C|=|A|+|B|+|C|-|A\cap B|-|A\cap C|-|B\cap C|+|A\cap B\cap C|$$

即三个集合的并集的大小可以表示成单个集合之和减去两两之交交集加上三个的交集。再进一步推广，可以得到如下定理。

定理 6.5　容斥原理　设 A_1，A_2，A_3，…，A_n 是有穷集，那么

$$|A_1\cup A_2\cup\cdots\cup A_n|=\sum_{1\leqslant i\leqslant n}|A_i|-\sum_{1\leqslant i<j\leqslant n}|A_i\cap A_j|+\sum_{1\leqslant i<j<k\leqslant n}|A_i\cap A_j\cap A_k|-\cdots$$
$$+(-1)^{n+1}|A_1\cap A_2\cap\cdots\cap A_n|$$

例 3　（埃拉托色尼筛）求出不超过 100 的素数的个数。

解　若某个数是素数，先找出不被小于或等于它的平方根整除的素数，对于 100 来说，需要的素数是 2、3、5、7。那么就可以写出式子

$$N=99-\left|\frac{100}{2}\right|-\left|\frac{100}{3}\right|-\left|\frac{100}{5}\right|-\left|\frac{100}{7}\right|+\left|\frac{100}{2\times3}\right|+\left|\frac{100}{2\times5}\right|+\left|\frac{100}{2\times7}\right|+\left|\frac{100}{3\times5}\right|+\left|\frac{100}{3\times7}\right|$$
$$+\left|\frac{100}{5\times7}\right|-\left|\frac{100}{2\times3\times5}\right|-\left|\frac{100}{2\times3\times7}\right|-\left|\frac{100}{2\times5\times7}\right|-\left|\frac{100}{3\times5\times7}\right|+\left|\frac{100}{2\times3\times5\times7}\right|$$
$$=99-50-33-20-14+16+10+7+6+4+2-3-2-1-0+0$$
$$=21$$

因此存在 21+4=25 个不超过 100 的素数的个数。

例 4　有 8 个物体，问所有物体都不在原位置的方案个数。

解　所有不在原位置的方案个数等价于排列数减去至少一个在原位置的方案个数，加上至少 2 个在原位置的方案个数，减去至少 3 个在原位置的方案个数…。于是可以得到：

$$N=8!-C_8^1\times(8-1)!+C_8^2\times(8-2)!-C_8^3\times(8-3)!+\cdots+(-1)^8C_8^8\times0!$$
$$=8!\left(1-\frac{1}{1!}+\frac{1}{2!}-\frac{1}{3!}+\cdots+\frac{1}{8!}\right)=14833$$

因此，共有 14833 种方案可以使所有物体不在原位置。

定理 6.6　错位排列公式　对于 n 个物体来说，使得它们都不在原位置的排列的方案总数为

$$D_n=n!\left(1-\frac{1}{1!}+\frac{1}{2!}-\frac{1}{3!}+\cdots+(-1)^n\frac{1}{n!}\right)$$

6.5.2　反演理论

在代数学和数学分析中，为了确定某个未知元，通常先通过问题要求列出未知元所满足的一系列方程，再通过解方程的方法得到未知元本身。对于未知序列，也是通过问题要求列出序列满足的线性关系，再通过这些线性关系解出序列本身的。某些时候，若一个序列不满足齐次线性关系，那么求解起来就比较困难。而反演，则是求解序列的另外一种方法，它通

过一个序列表示出另外一个序列，并可以通过另外一个序列得到这个序列的表示方法，那么知道其中任意一个序列自然就能解出另外一个序列。

若 $f(n)$、$g(n)$满足

$$(\alpha): g(n)=\sum_{i=1}^{n} a_i f(i) \leftrightarrow (\beta): f(n)=\sum_{i=1}^{n} b_i g(i),$$

则称 (α)、(β) 互为反演公式。

定理 6.7　第一反演公式　设 n 是一个自然数，若多项式序列$\{p_n\}$、$\{q_n\}$满足

$$p_n(x)=\sum_{k=0}^{n} a_{nk} q_k(x), \qquad q_n(x)=\sum_{k=0}^{n} b_{nk} p_k(x)$$

且 a_{ii}、b_{ii} 不为 0，则

$$v_n=\sum_{k=0}^{n} a_{nk} u_k, \qquad u_n=\sum_{k=0}^{n} b_{nk} v_k$$

定理 6.8　二项式反演公式　设有两个序列$\{a_n\}$、$\{b_n\}$，则

$$a_n=\sum_{k=0}^{n} C_n^k b_k \rightarrow b_n=\sum_{k=0}^{n} C_n^k a_k$$

现在用二项式反演公式重新推导错位排列公式。若 n 个物体的错位排列总数为 D_n，则易推得方案总数 $n!$，可以表示成

$$n!=\sum_{k=0}^{n} C_n^k D_{n-k}$$

通过二项式反演公式可以得到

$$D_n=\sum_{k=0}^{n} C_n^k (n-k)!=n!\sum_{k=0}^{n} \frac{(-1)^k}{k!}$$

6.5.3　Mobius 反演

先从一些基本的函数开始，之后引入 Mobius 反演的概念与应用。

欧拉函数　定义所有小于正整数 n 的并且不能整除它的数的个数为正整数 n 的欧拉函数，记为 $\varphi(n)$，规定 $\varphi(1)=1$。

Mobius 函数　定义正整数 n 的 Mobius 函数为 $\mu(n)$，其中

$$\mu(n)=\begin{cases} 1, & n=1 \\ (-1)^k, & n=p_1\times p_2\times\ldots\times p_k，p_i\text{为互不相同的素数} \\ 0, & \text{其他情形} \end{cases}$$

例如，30=2×3×5，则 $\mu(30)=-1$；又如，121=11×11，则 $\mu(121)=0$。

通过 Mobius 函数，可以得到一条引理。

引理 6.1　$\mu(n)$ 满足以下公式：

$$\sum_{d|n}\mu(k)=\begin{cases}1, & n=1\\ 0, & n>1\end{cases}$$

该式等价于

$$\sum_{d|(n/d_1)}\mu(\mathrm{d})=\begin{cases}1, & n=d_1\\ 0, & n>d_1\end{cases}\text{，其中}d_1\mid n$$

于是可以得到 Mobius 反演公式

定理 6.9　Mobius 反演公式　设 $f(n)$、$g(n)$是定义在正整数集合上的两个函数。

$$f(n)=\sum_{d|n}g(d) \tag{6-1}$$

成立。当且仅当

$$g(n)=\sum_{d|n}\mu(d)f\left(\frac{n}{d}\right) \tag{6-2}$$

成立时，称 $f(n)$、$g(n)$互为 Mobius 逆变换，式（6-1）和式（6-2）为 Mobius 反演公式。

此定理较难以理解，因此给出证明。

证明　式（6-1）成立，则有

$$f\left(\frac{n}{d}\right)=\sum_{d_1|(n/d)}g(d_1) \tag{6-3}$$

因 $d_1\mid(n/d)$，所以存在 m 使得 $dd_1m=n$，从而 $d\mid(n/d_1)$，这里的“|”表示除完之后取整数部分之值。

将式（6-3）代入式（6-2）右端，可得

$$\begin{aligned}g(n)&=\sum_{d|n}\mu(d)\sum_{d_1|(n/d)}g(d_1)=\sum_{d|n}\sum_{d_1|(n/d)}\mu(d)g(d_1)=\sum_{d_1|n}\sum_{d|(n/d_1)}\mu(d)g(d_1)\\&=\sum_{d_1|n}\left(\sum_{d|(n/d_1)}\mu(d)g(d_1)\right)=g(n)\end{aligned} \tag{6-4}$$

反之也易推得，可以作为练习加深对于 Mobius 反演的理解。

Mobius 还有另外一种表示方法，即

$$f(n)=\sum_{n|d}g(d)\rightarrow g(n)=\sum_{n|d}\mu\left(\frac{d}{n}\right)f(d) \tag{6-5}$$

接下来介绍 Mobius 反演的一些应用。

例 5　有一个有 m 个不同字母的字母表 T，从中选取字母排列成长度为 n 的圆环，问这样的排列有多少种。规定方向为顺时针，若存在一个断点使得一个序列与另一个断开的序列相同，则这两个圆环相同，即对于 a、b 构成的长度为 4 的环来说，*abab* 与 *baba* 是相同的。

先引入周期的概念。如果由 k 个元素的线排列 L 在圆周上重复若干次得到圆排列 W，则称 L 是圆排列的周期，其中最小者，称为最小周期。

解　对于一个长度为 n 的线排列来说，从 m 个字符中选取来进行排列的方案总数为 m^n 种。对于长度为 4、字符个数为 2 的圆排列来说，理论上排列的方案总数为 $2^4/4=4$ 种。但实际上却有 6 种。题目中提到的 *abab* 构成的圆环，将其断开只能得到 *abab* 和 *baba* 两种不同的线排列，而在错误的答案中，这个圆环断开被看成有 4 种不同的线排列，那么在算圆排列时就会有重复减去的现象，这是因为在有重复元素的圆环中，存在着周期的概念。4 能被 1、2、4 整除，那么对于长度为 4 的环来说，就存在着周期为 1、2、4 三种不同的排列。将周期为 1 的圆排列断开可以得到 1 种线排列，周期为 2 时存在 2 种线排列，周期为 4 时存在 4 种线排列。若周期为 p 的圆排列有 $W(p)$种，那么将其断开可以得到 $pW(p)$种不同的线排列，写出等式

$$\sum_{p|n} pW(p) = m^n$$

即对于所有不同周期的圆排列，所能得到的线排列总数为直接得到的线排列的个数。

应用 Mobius 反演，可得：

$$pW(p) = \sum_{q|p} \mu\left(\frac{p}{q}\right) m^q \rightarrow W(p) = \frac{1}{p}\sum_{q|p} \mu\left(\frac{p}{q}\right) m^q \tag{6-6}$$

令 m 个字母构成长度为 n 的圆排列的方案总数为 $C_m(n)$，则

$$C_m(n) = \sum_{p|n} W(p) = \sum_{p|n} \frac{1}{p}\sum_{q|p} \mu\left(\frac{p}{q}\right) m^q \tag{6-7}$$

在实际应用中，求 Mobius 函数一般需要利用欧拉筛，具体代码如下：

```
int a[100005],p[20005],phi[100005],mu[100005],tot;
void eula(){
    memset(a,0,sizeof(a));
    mu[1]=phi[1]=a[1]=1;tot=0;
    for(int i=2;i<=100000;i++){
        if(!a[i])a[i]=mu[i]=1,p[++tot]=i,phi[i]=i-1;
        for(int j=1;j<=tot&&p[j]*i<=100000;j++){
            a[i*p[j]]=1;
            if(i%p[j]==0){
                mu[i*p[j]]=0;phi[i*p[j]]=phi[i]*p[j];break;
            }
            mu[i*p[j]]=-mu[i];phi[i*p[j]]=phi[i]*phi[p[j]];
        }
    }
}
```

6.5.4 例题讲解

例 6-8 Co-prime

Time Limit: 2000/1000 ms (Java/Others) Memory Limit: 65536/65536 KB (Java/Others)

题目描述：

给定一个数 n，问在区间 A 到 B 中有多少个数与 n 互素。

如果两个数之间除了 1，没有其他素公约数，或者说这两个数最大公约数为 1，则称这两个数互素。1 与任何数互素。

输入格式：

输入的第一行包含一个整数 T（$0 < T \leqslant 100$）表示测试数据的组数，接下来 T 行中，每行包含三个数 A、B、N，$1 \leqslant A \leqslant B \leqslant 10^{15}$，$1 \leqslant n \leqslant 10^{9}$。

输出格式：

对于每组测试数据，输出所有在 A 与 B 之间（包含 A、B 在内）的与 n 互素的数的个数。遵照下面样例的输出格式。

样例输入：

2

1 10 2

3 15 5

样例输出：

Case #1: 5

Case #2: 10

题目来源：HDU 4135。

解题思路：

给定 n 以及区间[A,B]，求在[A,B]中所有与 n 互素的数的个数。

先求出 n 的所有素因数，再利用容斥原理求出小于或等于 $A-1$ 及小于或等于 B 的所有与这些素因数互素的数个数，最后将求出的这两个数相减即可。

题目实现：

```
1  #include<cstdio>
2  #include<cstring>
3  int a[100005],p[20005],tot;
4  int T,getp[105],tot2,n;
5  long long l,r;
6  void div(int t){
7          tot2=0;memset(getp,0,sizeof(getp));
8          for(int i=1;i<=tot&&p[i]*p[i]<=t;i++){
```

```
9                 if(t%p[i])continue;
10                getp[++tot2]=p[i];
11                while(t%p[i]==0)t/=p[i];
12                if(t==1)break;
13        }
14        if(t!=1)getp[++tot2]=t;
15  }
16  void dfs(long long q,long long &b,int k,int r,int t){
17        if(r&1)b=b-q/k;else b=b+q/k;
18       for(int i=t+1;i<=tot2;i++){
19                dfs(q,b,k*getp[i],r+1,i);
20        }
21  }
22  long long geta(long long s){
23        long long t=0;
24        dfs(s,t,1,0,0);
25        return t;
26  }
27  void eula(){
28        memset(a,0,sizeof(a));
29        a[1]=1;tot=0;
30        for(int i=2;i<=100000;i++){
31                if(!a[i])a[i]=1,p[++tot]=i;
32                for(int j=1;j<=tot&&p[j]*i<=100000;j++){
33                        a[i*p[j]]=1;
34                        if(i%p[j]==0)break;
35                }
36        }
37  }
38  int main(){
39        eula();
40        scanf("%d",&T);
41        for(int q=1;q<=T;q++){
42                scanf("%lld%lld%d",&l,&r,&n);
43                div(n);
44                printf("Case #%d: %lld\n",q,geta(r)-geta(l-1));
45        }
46  }
```

例 6-9　GCD

Time Limit: 2000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

给出 5 个整数：a、b、c、d、k，要在区间[a,b]找出 x 和区间[c,d]中找出 y，使得 x、y 的最大公约数为 k。因为数的选择方案可能会特别庞大，因此只需输出有多少组这样不同的数对即可。

请注意，x=5、y=7 和 x=7、y=5 被认为是相同的数对。

可以假定在所有数据中 a=c=1。

输入格式：

输入包含多组数据。第一行是数据的组数，总共有不超过 3000 组输入数据。每组数据包含 5 个整数 a、b、c、d、k，$0 < a \leqslant b \leqslant 100000$，$0 < c \leqslant d \leqslant 100000$，$0 \leqslant k \leqslant 100000$。

输出格式：

对于每组数据，输出选择的方案数。使用样例输出的格式。

样例输入：

2

1 3 1 5 1

1 11014 1 14409 9

样例输出：

Case 1: 9

Case 2: 736427

题目来源：HDU 1695。

解题思路：

给出 a、b、c、d、k，在区间[a,b]及[c,d]中各选一个数使其最大公约数为 k，求出这样的数对的组数。因为默认 a、c 为 1，所以求小于等于 b、d 中有多少数对最小公约数为 k，即求[b/k]和[d/k]中有多少数对是互素的。设 $f(k)$为[1,b]和[1,d]中最大公约数为 k 的数对的个数，$f(k)$=[b/k]×[d/k]。可以得到

$$f(1) = \sum_{i \leqslant \min(b,d)} \mu(i)F(i)$$

题目实现：

```
#include<cstdio>
#include<cstring>
int t,mu[100005],pri[20005],pd[100005],tot,a,b,c,d,k,q;
void eula(){
    pd[1]=1;mu[1]=1;
    for(int i=2;i<=100000;i++){
        if(!pd[i])pd[i]=1,pri[++tot]=i,mu[i]=-1;
        for(int j=1;j<=tot&&pri[j]*i<=100000;j++){
            pd[i*pri[j]]=1;
```

```
                    if(i%pri[j])mu[i*pri[j]]=-mu[i];
                    else {
                        mu[i*pri[j]]=0;
                        break;
                    }
                }
            }
        }
    }
    int min(int a,int b){return a>b?b:a;}
    long long mobius(int x,int y){
        long long f=0;
        for(int i=1;i<=q;i++)
            f+=(long long)(x/i)*(y/i)*mu[i];
        return f;
    }
    int main(){
        eula();
        scanf("%d",&t);
        for(int p=1;p<=t;p++){
            scanf("%d%d%d%d%d",&a,&b,&c,&d,&k);
            printf("Case %d: ",p);
            if(k==0){printf("0\n");continue;}
            b/=k;d/=k;
            q=min(b,d);
            printf("%lld\n",mobius(b,d)-mobius(q,q)/2);
        }
    }
```

6.6　群论与 Polya 定理

作为数学中的一个重要的组成部分，群论的应用十分广泛。本节将从群的基本性质出发，具体介绍 Burnside 引理及 Polya 定理的内容，并且通过具体题目分析其应用。

6.6.1　群的基本性质

二元运算是由两个元素形成第三个元素的一种运算规则，比如算术运算中的加（+）、减（-）、乘（×）、除（/）都是二元运算。

定义 6.4　设 G 是一个非空集合，·是它的一个二元运算，如果满足以下条件：

（1）封闭性：若 $\forall a \in G$、$b \in G$、存在唯一确定的c使得$a \cdot b = c \in G$。

（2）结合性：$\forall a \in G$、$b \in G$、$c \in G$，有 $a \cdot (b \cdot c)=(a \cdot b) \cdot c$。

（3）单位元：存在 $e \in G$，使得 $e \cdot a=a \cdot e=a$，其中，e 称为单位元。

（4）逆元：存在 $b \in G$，使得 $a \cdot b=b \cdot a=e$（e 为单位元），则称 b 与 a 互为逆元素，简称逆元，b 记为 a^{-1}。

则称 G 对·构成一个群。

群中所含元素的个数，称为群的阶。根据群内元素个数是否有限，群可以分为有限群和无限群。群内的元素也有阶的定义，若群内的一个元素 a 最少通过 k 次二元运算操作才能得到单位元，那么可以称 k 为 a 的阶。

群满足消去律，即由 $a \cdot b = a \cdot c$ 可得 $b = c$。

对于加法群来说，$a+b=a+c$ 等价于 $b=c$；对于乘法群来说，$a \times b=a \times c$，当 a 不为 0 时等价于 $b=c$。

6.6.2 置换群

对于从 1 到 n 的某个排列$\{b_n\}$来说，若存在 1 到 n 的另一个排列$\{a_n\}$，使得 a_i 能通过交换顺序得到 b_i，则称$\{b_n\}$是$\{a_n\}$的 n 元置换。例如，1、2、3、4 的一个 4 元置换为

$$\begin{pmatrix} 1, & 2, & 3, & 4 \\ 4, & 1, & 3, & 2 \end{pmatrix}$$

表示 1 被 4 取代，2 被 1 取代，4 被 2 取代。1、2、3、4 经过一次置换变为 4、1、3、2，再经过一次置换变为 2、4、3、1。容易发现，在这个置换中，元素的顺序是可以改变的，即上面的四元置换也可写为

$$\begin{pmatrix} 1, & 2, & 4, & 3 \\ 4, & 1, & 2, & 3 \end{pmatrix}$$

如果两个置换的 n 相同，那么存在连接运算，即

$$\begin{pmatrix} 1, & 2, & 3, \cdots, & n \\ a_1, & a_2, & a_3, \cdots, & a_n \end{pmatrix} \begin{pmatrix} a_1, & a_2, & a_3, \cdots, & a_n \\ b_1, & b_2, & b_3, \cdots, & b_n \end{pmatrix} = \begin{pmatrix} 1, & 2, & 3, \cdots, & n \\ b_1, & b_2, & b_3, \cdots, & b_n \end{pmatrix}$$

显然置换的连接满足结合律，但不满足交换律。

记一个 n 阶循环为

$$(a_1, \ a_2, \ a_3, \cdots, \ a_n) = \begin{pmatrix} a_1, & a_2, & a_3, \cdots, & a_n \\ a_2, & a_3, & a_4, \cdots, & a_1 \end{pmatrix}$$

两个循环不相交是指对于 $a_i (i \in [1,n])$，不存在 $b_j (j \in [1,m])$ 使得 $a_i = b_j$。因此置换可以写为若干个互不相交的置换的乘积。例如上面的例子可写为(1,2,4)×(3)。

介绍完置换的基本概念后，接下来就介绍一下置换群。

置换群必然包含元素与运算。它的元素就是上述的置换，运算就是连接操作。例如，一个 4×4 的棋盘，将它的格子进行编号，那样就能抽象成[1,16]的集合，然后规定 8 种置换操作：

不动、顺时针旋转 90°、逆时针旋转 90°、旋转 180°，以及翻转后这 4 种操作对应的操作。这 8 种置换操作满足封闭性，经过任何两种置换操作后一定会得到其中的另一种，各元素的映射是其自身的置换是单位元 e，并且每种置换都存在逆元，可以构成一个置换群了。

6.6.3　Burnside 定理及 Polya 定理

定理 6.10　Burnside 定理　设 $G=\{a_1,a_2,\cdots,a_g\}$是目标集$[1,n]$上的置换群，每个置换都可写成不相交循环的乘积。$c_1(a_k)$是在置换 a_k 的作用下不动点的个数，也就是长度为 1 的循环的个数。通过上述置换操作后相等的元素属于同一个等价类。若 G 将$[1,n]$划分成 l 个等价类，则等价类个数为

$$l=\frac{1}{|G|}\sum_{i=1}^{|G|}c_1(a_i)$$

例 6：一个 2×2 的正方形用两种颜色染色，共有多少种方案？注：经过转动的算同一种方案。

解：可以知道，如图 6.3 所示，2 个颜色 4 个方块共有 16 种染色的方法。

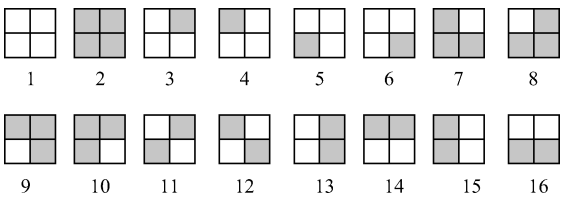


图 6.3　4 个方块的染色方案

对于一个 2×2 的正方形来说，置换操作有 4 种：不动、逆时针旋转 90°、顺时针旋转 90°、旋转 180°。其中，每种置换的循环方式为

不动：　　(1)(2)(3)(4)(5)(6)…(16)

逆时针旋转 90°：　(1)(2)(3 4 5 6)(7 8 9 10)(11 12)(13 14 15 16)

顺时针旋转 90°：　(1)(2)(6 5 4 3)(10 9 8 7)(11 12)(16 15 14 13)

旋转 180°：　　(1)(2)(3 5)(4 6)(7 9)(8 10)(11)(12)(13 15)(14 16)

利用 Burnside 定理可得答案为(16+2+2+4)/4=6 种。

定理 6.11　Polya 定理　设 $G=\{a_1,a_2,\cdots a_{|G|}\}$是目标集$[1,n]$上的置换群，每个置换都可写成不相交循环的乘积。$m(f_i)$是在置换 f_i 的作用下循环节的个数，通过上述置换操作后相等的元素属于同一个等价类。若 G 将$[1,n]$用 k 种颜色分别进行染色，然后划分成 L 个等价类，则染色后的等价类个数 L 为

$$L=\frac{1}{|G|}\sum_{i=1}^{|G|}k^{m(f_i)}$$

再用上面的正方形染色举例，给 4 个方块顺时针标号 1、2、3、4，则

不动：　　(1)(2)(3)(4)

逆时针旋转 90°：　　(1 4 3 2)

顺时针旋转 90°：　　(1 2 3 4)

旋转 180°：　　(1 3)(2 4)

可得循环节个数分别为 4、1、1、2，利用 Polya 定理得到答案为(16+2+2+4)/4=6 种。可以看出，Burnside 定理作用于染色后的置换群中，而 Polya 定理则直接作用于对原来的 *n* 个对象组成的置换群。当颜色较多时，Polya 定理会更加简便。

6.6.4 例题讲解

例 6-10　Necklace of Beads

Time Limit: 3000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

将红、蓝、绿三种珠子连接起来串成有 *n* 个珠子的圆形项链（*n* < 40）。如果通过旋转项链或者通过任一对称轴翻转能得到其他项链，那么认为这两条项链是相同的。问有多少种不同的项链排列方案。

输入：

输入数据包含多行，每一行有一个数 *n*。

最后一行是−1。

输出：

对于每组输入数据，输出不同排列的方案数。

样例输入：

```
4
5
-1
```

样例输出：

```
21
39
```

题目来源：HDU 1817。

解题思路：

一串有 *n* 个珠子的项链，由三种颜色组成。若项链通过旋转或者以某个珠子或两个珠子的中点为轴翻转能得到另一串项链，则认为这两串项链是相同的。问总共有多少种不同的项链。

本题需要用到 Polya 定理。对于长度为 *n*（即 *n* 个珠子）的项链来说，它可以通过不动、

旋转 1 个珠子、旋转 2 个珠子、…、旋转 $n-1$ 个珠子来得到新的项链。对于旋转 i 个珠子的等价类来说，其中的循环节个数就是 i 与 n 的最大公约数，因为其中的某串项链经过 $n/\gcd(n,i)$ 次旋转必然能得到原来的项链，其中 $\gcd(n,i)$表示 n 和 i 的最大公约数。那么旋转的方案数为：

$$S_1 = \sum_{i=0}^{n-1} 3^{\gcd(n,i)}$$

接下来求翻转的方案数。

若有偶数个珠子，那么它可以分成 $n/2$ 种对珠子的翻转方案，以及 $n/2$ 种对两个珠子中点的翻转方案。珠子的翻转方案有$(n+2)/2$ 个循环节，分别选取 2 个珠子以及剩余$(n-2)$个珠子进行两两配对。对于中点的反转方案，则有 $n/2$ 个循环节，也就是对中点两边的珠子进行两两配对。

若有奇数个珠子，那么它有 n 种翻转方案，每种翻转方案有$(n+1)/2$ 个循环节，也就是选定 1 个珠子以及剩余的珠子进行两两配对。那么翻转的方案数为：

$$S_2 = \begin{cases} \dfrac{n}{2} \times 3^{\frac{n+2}{2}} + \dfrac{n}{2} \times 3^{\frac{n}{2}}, & n\text{为偶数} \\ n \times 3^{\frac{n+1}{2}}, & n\text{为奇数} \end{cases}$$

方案总数为上面两者之和除以置换群的方法数，即 $(S_1 + S_2) / (n \times 2)$。

题目实现：

```
#include<cstdio>
#include<cstring>
long long all;
int n;
long long pow(int a,int b){return b?pow(a,b-1)*a:1;}
int gcd(int a,int b){return b?gcd(b,a%b):a;}
int main(){
    while(scanf("%d",&n)&&n!=-1){
        if(n==0){
            printf("0\n");continue;
        }
        all=pow(3,n);
        for(int i=1;i<n;i++)
            all+=pow(3,gcd(n,i));
        if(n%2==0)all+=pow(3,n/2+1)*n/2+pow(3,n/2)*n/2;
        else all+=pow(3,n/2+1)*n;
        all=all/n/2;
        printf("%lld\n",all);
    }
}
```

6.7 练习题

习题 6-1

题目来源：HDU 1027。

题目类型：排列组合。

题目思路：已知 n 及 k，求 1 到 n 的第 k 个排列。在算法库中有 next_permutation 函数，或者可以通过递归来得到排列。

习题 6-2

题目来源：HDU 5698。

题目类型：排列组合。

题目思路：题意很明确。可以把问题看成纵向移 x 步与横向移 x 步有多少种方案，这是简单的排列组合。若有 y 格，相当于把 $y-1$ 个不可区分的物体放入 x 个可区分的盒子里，方案数为 $C(y-2,x-1)$。然后将相同步数的横向与纵向相乘，即可得到当前步数走到右下角的方案数。预处理阶乘的模及阶乘的数论倒数（Number-Theoretic Reciprocal）即可在 $O(1)$时间复杂度内求出组合数。

习题 6-3

题目来源：HDU 2152。

题目类型：母函数。

题目思路：构造一系列母函数，最小次数为最少多少个水果，最高次数为最多多少个水果，再将这些母函数相乘即可得到答案。

习题 6-4

题目来源：HDU 2065。

题目类型：指数型母函数。

题目思路：A、B、C、D 中 A 和 C 出现偶数次，B 和 D 出现任意次，求排列数。可以构造指数型母函数序列，对于 A 和 C 来说，$f(x)=\sum_{i=0}^{\infty}\frac{1}{(2i)!}x^{2i}=(e^{x}+e^{-x})/2$；对于 B 和 D 来说，$f(x)=e^{x}$，那么方案数就是 $e^{2x}\times[(e^{x}+e^{-x})/2]^{2}$ 的第 n 项系数乘以 $n!$，答案为 $2^{n-1}+4^{n-1}$，之后再将答案对 100 取模。

习题 6-5

题目来源：HDU 1134。

题目类型：Catalan 数。

题目思路：有 $2n$ 个点，需要将它们用 n 条线两两连接并且不能相交。固定一个点，若选择相邻的一个点，那么两边就会有 0 个点和 $n-2$ 个点，再依次选下去并保证两边点的个数都是偶数，很显然答案就是 Catalan 数，需要用到高精度。

习题 6-6

题目来源：HDU 6143。

题目类型：第二类 Stirling 数。

题目思路：将 m 个字符放入 2 个长度为 n 的串中，要求这两个串所含的字符不同。首先，若第一个串选取 x 个字符，对于第二个串来说，剩下的 $m-x$ 个字符就有$(m-x)^n$种排列方式。第一个串则有 $C(m,x)\times S(n,x)\times x!$种排列方式，意思是 m 个字符取 x 个字符的方案数乘以 n 个位置放入 x 个集合的方案数，再乘以 x 个字符的排列数。然后将两种排列方式相乘，其中 x 需要从 1 取到 $m-1$。将所有结果加起来就是答案。需要预处理打表。

习题 6-7

题目来源：HDU 5514。

题目类型：容斥原理。

题目思路：对于第 i 个青蛙，求它能跳到最近 $\gcd(a_i,m)$的位置。那么初步解法就是每个青蛙跳到的位置之和，减去每两只青蛙跳到的位置之和，加上每三只青蛙跳到的位置之和，…，但是这样解明显会超时，于是就可以先对 m 分解因数。若某个较小的因数能被 $\gcd(a_i,m)$整除，那么说明它的倍数会被跳到。若另一个因数会被这个因数整除，那么说明这个因数的倍数会被重复计算，因此只需对 m 的因数根据容斥原理进行处理即可。

习题 6-8

题目来源：HDU 5230。

题目类型：整数划分。

题目思路：给定 n、c、l、r，在 1 到 $n-1$ 中取各不相同的数，使得这些数的和在 $l-c$ 到 $r-c$ 内的方案数有多少。如果用原来的 dp，显然空间是不够用的；那么就来换一下 dp 的方程。令 dp[i][j]为选 i 个数相加的和为 j 的方案数，dp[i][$j-i$]就是 i 个数相加的和为 $j-i$ 的方案数。让每个数都加 1 即可得到 dp[i][j]。dp[$i-1$][$j-i$]代表 $i-1$ 个数相加的和为 $j-i$ 的方案数，那么让选择的 $i-1$ 个都加 1 并且再加上 1 个 1 也可转移到 dp[i][j]。可以得到转移方程 dp[i][j]=dp[i][$j-i$]+

dp[i−1][j−i]，因为每个数都不同，所以第二维从 dp[i][i(i+1)/2]开始，边界条件为 dp[0][0]=1。因为转移方程只与 dp[i]和 dp[i−1]有关，因此可以使用滚动数组。

习题 6-9

题目来源：HDU 3923。

题目类型：Polya 定理。

题目思路：n 种颜色的 m 个珠子排列成项链，问不同项链的方案数有多少种。同例 6-10 的做法，只是颜色由 3 种变成了 m 种，在计算时注意将除以 2n 转换为乘 2n 的倒数，即将 x÷2n 转换成 x×1/2n。

第 7 章

计算几何

计算几何的研究对象一般都是非常复杂的图形组合问题，或者步骤繁多的几何操作问题。计算几何研究的典型问题有凸包、图、多边形的三角剖分、划分问题和相交问题。一般情况下给出一个计算几何问题，读者通常会感到无从下手，这就需要将问题一层层剥开，逐步用代码实现，再做判断。

7.1 多边形上的数据结构表示

本节先将计算几何的基本元素，如点线、线段、简单图形等完整地表示出来，以便理清思路、解决问题。由于计算几何问题经常涉及精度，所以需要对一个很小的数进行正负判断。

```
1    //判断一个数的正负
2    int sign(double x)
3    const double eps=1e-8;
4    int sign(double x)
5    {
6        (x)>eps?1:((x)<-eps?2:0);
7    }
```

7.1.1 点

用一个二维点类保存平面上点的 x 坐标和 y 坐标。

```
1    //描述平面上的一个点
2    struct point{double x,y;};
```

同时重载点的加法、减法、乘法运算：

$$(x_1, y_1) \pm (x_1, y_1) = (x_1 \pm x_2, y_1 \pm y_2)$$

$$a \times (x_1, y_1) = (a \times x_1, a \times y_1)$$

$$(x_1, y_1) \times b = (x_1 \times b, y_1 \times b)$$

```
1   //加法运算
2   friend point operator + (const point &a, const point &b)
3   {
4       return point(a.x+b.x+a.y+b.y);
5   }
6   //减法运算
7   friend poinr operator - (const point %a,const point &b)
8   {
9       return point(a.x-b.x,a.y-b.y);
10  }
11  //左乘
12  friend point operator * (const point &a, const double &b)
13  {
14      return point(a.x*b,a.y*b);
15  }
16  //右乘
17  friend point operator * (const double &b, const point &a)
18  {
19      return point(a.x*b,a.y*b);
20  }
```

向量之间乘法的函数为：

$$(x_1, y_1) \cdot (x_2, y_2) = x_1 x_2 + y_1 y_2$$

$$\left|(x_1, y_1) \times (x_2, y_2)\right| = \det\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix}$$

```
1   //叉乘
2   double xmult(const point &a, const point &b)
3   {
4       return a.x*b.y-a.y*b.x;
5   }
6   //点乘
7   double dmult(const point &a，const point &b)
8   {
9       return a.x*b.x+a.y*b.y;
10  }
```

定义向量的旋转为：向量绕原点逆时针旋转 x（弧度）。

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

```
//向量的旋转
double rot(const point &a,double x)
{
    double tx=a.x, ty=a.y;
    return point(tx*cos(x)-ty*sin(x),tx*sin(x)+ty*cos(x));
}
```

7.1.2 线段

线段可以用一个二维线段类保存平面上点 *a* 和点 *b*（有顺序）。描述如下：

```
//线段
struct line
{
    point a,b;
}
```

与线段相关的函数有很多，这里只举几个例子：

（1）求一点 *p* 到线段 *AB* 的距离：如果 *p* 在 *AB* “外部”时，*p* 到 *AB* 距离就是 *p'A* 或者 *p''B*；否则就是垂线段的长度，参见图 7.1 点到线段的距离。

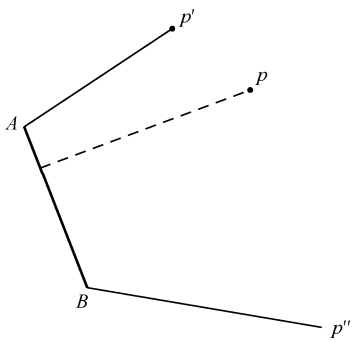


图 7.1　点到线段的距离

```
//点到线段的距离
double displ(const point p,const line l(const point a, const point b))
{
    if (sign(dmult(p-l.a,l.b-l.a))<0)
    return sqrt(sqr(p.x-l.a.x)+sqr(p.y-l.a.y));
    if (sign(dmult(p-l.b,l.a-l.b))<0)
    return sqrt(sqr(p.x-l.b.x)+sqr(p.y-l.b.y));
    return fabs(xmult(l.a-p,l.b-p)/sqrt(sqr(l.a.x-l.b.x)+sqr(l.a.y-l.b.y)));
}
```

（2）判断 p 点是否在线段 AB 上：只需判断 AB 和 Ap 的外积是否为零（共线），以及 pA 和 pB 的内积是否非正（在线段上）。

```
//判断点是否在线段上
double pos(const point p,const line l(const point a,const point b))
{
    return sign(xmult(p-l.a,l.b-l.a))==0 && sign(dmult(p-l.a,p-l.b))<=0;
}
```

（3）判断两条线段是否平行。

```
//判断两条线段 a、b 是否平行
bool parallel(line a,line b)
{
    return !sign(xmult(a.a-a.b,b.a-b.b));
}
```

7.1.3 多边形类

如果想利用计算几何知识解决应用问题，对规则图形则是必不可少的，而多边形恰是规则图形最重要的分支之一。因此，需要实现一个多边形类，更多的关于多边形的算法将在后几节介绍，这里只讨论关于多边形数据结构和简单的计算。

首先，建立多边形类，顺序地保存多边形的每一个顶点。

```
//多边形类
const int maxn=100;
struct polygon
{
    int n;
    point a[maxn];
}
```

计算多边形周长的代码如下：

```
//计算多边形周长
double perimeter(const polygon x(const int n,const point a[]))
{
    double sum=0;
    a[n]=a[0];
    for (int i=0;i<n;i++)
    sum+=sqrt(sqr(a[i+1].x-a[i].x)+sqr(a[i+1].y-a[i].y));
    return sum;
}
```

判断点是否在多边形的内部，在内部为 1，不在为 0，在边界上为 2。如果点 t 在多边形任一有向边的左侧且 y 坐标大至小顺序是 $a[i+1]$、t、$a[i]$的次数，与点 t 在多边形任一有向边的右侧且 y 坐标大至小顺序是 $a[i]$、t、$a[i+1]$的次数相等时，t 就在多边形内部。时间复杂度为 $O(N)$ 。

```
//判断点是否在多边形的内部
double xmult(const point &a, const point &b)
{
    return a.x*b.y-a.y*b.x;
}
//点乘
double dmult(const point &a，const point &b)
{
    return a.x*b.x+a.y*b.y;
}
int pointin(const polygon x(const int n,const point* a), const point t)
{
    int num=0,i,d1,d2,k;
    a[n]=a[0];
    for (i=0;i<n;i++)
    {
        line l;
        l.a=a[i];
        l.b=a[i+1];
        if (pos(t,l)) return 2;
        k=sign(xmult(a[i+1]-a[i],t-a[i]));
        d1=sign(a[i].y-t.y);
        d2=sign(a[i+1].y-t.y);
        if (k>0 && d1<=0 && d2>0) num++;
        if (k<0 && d2<=0 && d1>0) num--;
    }
    return num!=0;
}
```

7.1.4 例题讲解

例 7-1 Lifting the Stone

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/ Others)

题目描述：

找到给定多边形的重心。

输入：

输入由 T 组数据组成，每组数据的第一行给出 T 的数目。每个测试用例（对应每组数据）

以包含一个整数 N（$3 \leqslant N \leqslant 1000000$）的行开始，该行表示形成多边形的点的数目。接下来的 N 行，每行包括 2 个整数 X_i 和 Y_i（$|X_i|$、$|Y_i|$均小于或等于 20000），这些数字是第 i 个点的坐标。当按照给定的顺序连接点时，可得到一个多边形。假设多边形的边从不互相接触（除了相邻的），而且它们从不交叉。多边形的面积从不为零，也就是说它不能折叠成一条直线。

输出：

每一个测试用例打印一行。该行应该包含两个由一个空格隔开的数字，这是重心的坐标。将坐标四舍五入到最接近的数字，小数点后保留两位数（0.005 四舍五入到 0.01）。注意，如果形状不是凸的，重心可能在多边形的外面，如果输入数据中有这样的情况，也会输出重心。

样例输入：

```
2
4
5 0
0 5
-5 0
0 -5
4
1 1
11 1
11 11
1 11
```

样例输出：

```
0.00 0.00
6.00 6.00
```

题目来源：POJ 1385。

思路分析：

这个题目是求给定多边形的重心。经过重心的每条直线将平分多边形的面积，所以重心又可以称为多边形的面积中心。如何去找多边形的重心呢？按照重心定义，有

$$\frac{\sum_{k=1}^{n} m_i p_i}{\sum_{k=1}^{n} m_i}$$

式中，分母是一个定值，分母是关于坐标 p_i 的线性函数。由于线性关系，可以将多边形分成若干不相交的三角形（简单起见，取定一顶点，将其与不相邻顶点相连，将 n 边形变成 $n-2$ 个三角形），先求每个三角形的重心，然后以三角形的面积为权值求平均值。

题目实现：

```
#include<iostream>
#include<cstdio>
#include<cstring>

using namespace std;

const int N=1000000+3;
struct Point{
    double x,y;
    Point(double x=0,double y=0):x(x),y(y) {}
};
typedef Point Vector;
Vector operator-(Point a,Point b){
    return Vector(a.x-b.x,a.y-b.y);
}
double Cross(Vector a,Vector b){
    return a.x*b.y-a.y*b.x;
}
int n;
Point Centre_Mass()
{
    double area=0,sx=0,sy=0;
    Point p0,pi1,pi2;
    cin>>p0.x>>p0.y>>pi2.x>>pi2.y;
    int x,y;
    for(int i=1;i<n-1;i++)
    {
        pi1=pi2;
        scanf("%d%d",&x,&y);
        pi2=(Point){x,y};
        area+=Cross(pi1-p0,pi2-p0);
        sx+=Cross(pi1-p0,pi2-p0)*(pi1.x+pi2.x+p0.x)/3,
        sy+=Cross(pi1-p0,pi2-p0)*(pi1.y+pi2.y+p0.y)/3;
    }
    return Point(sx/area,sy/area);
}
int main()
{
    int T;
```

```
    cin>>T;
    while(T--)
    {
        cin>>n;
        Point cm=Centre_Mass();
        printf("%.2f %.2f\n",cm.x,cm.y);
    }
    return 0;
}
```

7.2 多边形相交问题

多边形是否相交会影响多边形的总面积、总周长等属性。判断多边形是否相交，是解决一部分计算几何问题最基础的步骤。本节从最简单、最基础的线段相交开始讨论，关于多边形的其他相交问题都会引用判断线段是否相交并求其交点的方法。

7.2.1 线段相交

判断线段相交的方法有很多，这里只介绍其中一种方法。两条线段相交，意味着选择任意一条线段 *AB*，另一条线段 *CD* 的两个端点分别位于这条线段的两侧，也就是 *AB* 与 *AC* 的外积和 *AB* 与 *AD* 的外积异号，且 *CD* 与 *CA* 的外积和 *CD* 与 *CB* 的外积也异号。当然首先要判断一下是否共线。

```
//判断线段是否相交
bool intersect(line a,line b)
{
    if ((parallel(a,b)) || xmult(a.a-a.b,a.a-b.a)*mult(a.a-a.b,a.a-b.b)>0
       || xmult(b.a-b.b,b.a-a.a)*mult(b.a-b.b,b.a-a.b)>0)    return false;
    else return true;
{
//求线段的交点
point res(line a,line b)
{
    if intersect(a,b)
    {
        double s1=xmult(a.a-b.a,b.a-b.a);
        double s2=xmult(a.b-b.a,b.b-b.a);
        return (s1*a.b-s2*a.a)/(s1-s2);
      }
```

```
17  }
```

7.2.2 多边形相交问题的讨论

为什么这一节的名字带上“讨论”二字？这是因为多边形相交问题还没有一个相对成熟的算法，能想出来的方法，都带着穷举的思想。归根结底，就是多次判断线段是否相交。严格地说，这样的方法不能称得上是一种算法。但若给一点限定：比如多边形换成凸包，一部分多边形换成半平面，等等，就能找出合适的筛点算法去解决这些问题；如果涉及求面积，也可以用某种替代法将多边形问题简化，这些问题留在以后几节讨论，这里仅根据最一般的多边形相交问题给出几种可行的思路。

1．通过枚举法判断多边形是否相交

根据线段相交的判定，容易想到枚举多边形的边，依次通过判断边是否相交来判断多边形是否相交。这里要注意把多边形相交和相切是否分开的问题。如果有一个 n 边形和 m 边形，判断其是否相交需要 $O(MN)$ 的时间复杂度。给定两个多边形，即 m 边形和 n 边形，它们的边分别记录在两个数组中，代码如下：

```
1   #include<bits/stdc++.h>
2   struct point
3   {
4       double x,y;
5       point(double a,double b)
6       {
7           x=a;
8           y=b;
9       }
10      void input()
11      {
12          scanf("%lf%lf",&x,&y);
13      }
14      friend point operator - (const point &a,const point &b)
15      {
16          return point(a.x-b.x,a.y-b.y);
17      }
18  };
19  struct line
20  {
21      point a,b;
22      line(point x,point y)
```

```
    {
        x=a;
        y=b;
    }
};
double det(point a,point b)
{
    return a.x*b.y-a.y*b.x;
}
const double eps=1e-8;
int cmp(double x)
{
    if (fabs(x)<eps) return 0;
    if (x>0) return 1;
    return -1;
}
bool intersection(line a[],line b[])    //判断是否相交
{
    for (int i=1;i<=m;i++)
    for (int j=1;j<=n;j++)
    {
        if ((cmp(det(a[i].a-b[j].a,a[i].a-a[i].b))==cmp(det(a[i].a-a[i].b,a[i].a-b[j].b)))
            &&(cmp(det(b[j].a-a[i].a,b[j].a-b[j].b))==cmp(det(b[j].a-b[j].b,b[j].a-a[i].b))))
            return true;
    }
    return false;
}
```

于是，有了多边形相交的判断，就能根据 7.3 节的算法算出相交的面积，相应地，也能算出其并的面积（更多的问题将放在 7.3 节讨论）。到此为止，我们已经完成了一系列关于多边形相交问题的讨论，只不过归根结底都是穷举法，后面几节会介绍用半平面去求多边形相交部分的面积。

2．判断一个多边形的顶点是否在另一个多边形的内部

根据这种方法也能判断多边形是否相交，但时间复杂度并没有得到改善。这种方法是不需要判断线段是否相交这一步骤，所以也是另一种思路。

7.2.3 例题讲解

例 7-2　Pick-up sticks

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

在一个坐标系里依次扔 n（n≤100000）根木棒，问最后哪些木棒上面没有被其他木棒覆盖，答案的范围为 m（0<m<1000）。

输入：

输入由若干组数据组成，数据范围为 1≤N≤100000，第一行表示形成多边形的点的数目。下面的 n 行各包含四个数字，这些数字是一个端点的平面坐标。木棍是按照扔的顺序排列的。假设顶部不超过 1000 根，输入以 n=0 的情况结束。

输出：

对于每一个输入样例，打印一行顶部木棍的输出列表，列出示例中给出的格式的顶部分支。顶部的木棒应该按照它们的顺序排列。

图 7.2 说明了输入的第一个例子。

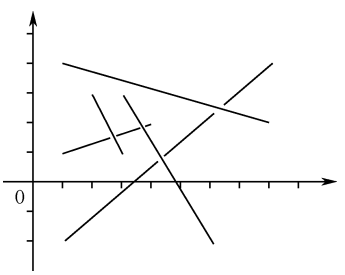


图 7.2　输入样例

样例输入：

5
1 1 4 2
2 3 3 1
1 -2.0 8 4
1 4 8 2
3 3 6 -2.0
3
0 0 1 1
1 0 2 1
2 0 3 1
0

样例输出：

Top sticks: 2, 4, 5.
Top sticks: 1, 2, 3.

题目来源：POJ 2653。

思路分析：

在一个坐标系里依次扔 n（n≤100000）根木棒，问最后哪些木棒上面没有被其他木棒覆盖，也就是求最上层的木棒，答案的个数范围为 m（0<m<1 000）。关键就是解决线段的相交问题，接下来用时间复杂度为 $O(N^2)$ 的某种方法按读入顺序遍历一遍。

如何去遍历呢？一种简单的方法是建一个 m 大小的数组，表示最上层木棒的编号，然后读入一个判断与该数组编号对应木棒的相交情况，不断地更新该数组。但由于可能会出现插

入、删除的操作，所以需要微小的调整；也可以用队列来代替数组。

题目实现：

```
#include<iostream>
#include<cstdio>
#define MAXN 101000
#define eps 1e-10
using namespace std;
struct point{
    double x,y;
    point(){x=y=0;}
    point(double _x,double _y){x=_x; y=_y;}
};
struct line{
    bool is_top;
    point a,b;
    int prev,next;
    line(){
        is_top=0;
        prev=next=0;
    }
}T[MAXN];
int head;
point V(point start,point end){return point(end.x-start.x,end.y-start.y);}
double Cross(point a,point b){return a.x*b.y-b.x*a.y;}
bool cover(line a,line b){
    return (Cross(V(a.a,a.b),V(a.a,b.a))*Cross(V(a.a,a.b),V(a.a,b.b))<-eps)
    &&(Cross(V(b.a,b.b),V(b.a,a.a))*Cross(V(b.a,b.b),V(b.a,a.b))<-eps);
}int n;
int main()
{
    double a,b,c,d;
    while(scanf("%d",&n)&&n){
        head=0;
        for(int i=1;i<=n;i++){
            scanf("%lf%lf%lf%lf",&T[i].a.x,&T[i].a.y,&T[i].b.x,&T[i].b.y);
            T[i].is_top=1; T[i].next=0; T[i].prev=head; T[head].next=i;
            for(int j=head;j;j=T[j].prev){
                if(cover(T[j],T[i])){
                    T[j].is_top=0;
                    T[T[j].prev].next=T[j].next;
```

```
39                              T[T[j].next].prev=T[j].prev;
40                          }head=i;
41                      }
42              printf("Top sticks:");
43              for(int i=1;i<n;i++) if(T[i].is_top) printf(" %d,",i);
44              printf(" %d.\n",n);
45          }
46          return 0;
47      }
```

7.3 多边形求面积

在传统的平面几何上，求多边形的面积是一个很烦琐、复杂的过程。但如果把多边形放在平面直角坐标系中，利用解析几何的知识，它的面积求法会简单很多。

7.3.1 计算多边形的面积

以下介绍两种比较容易理解的计算多边形面积的方法。

1．利用叉乘计算多边形面积

知道多边形的每个顶点坐标，也就是知道了一个多边形类（Polygon）。连接原点和多边形的每个顶点，这些线将 *n* 边形分成了 *n* 个三角形，有向（顶点呈逆时针排列则面积为正，顺时针排列则为负）地叠加这些三角形面积（反向相减），最后得到的就是 *n* 边形的面积。这就用到了外积，两个向量的外积是以这两个向量为邻边的平行四边形面积（有向），除以 2 就是三角形。计算多边形的面积代码如下：

```
1   //计算多边形的面积
2   double area(const polygon x)
3   {
4       double sum=0;
5       a[n]=a[0];
6       for (int i=0;i<n;i++)
7           sum+=xmult(a[i+1],a[i]);
8       return sum/2;
9   };
```

2．利用定积分计算多边形面积

计算多边形面积可以考虑定积分的形式。定积分是有向的，顺次求和，重复部分相互抵

消，最后剩下的总面积的绝对值就是多边形的面积。每一条边和 x 轴形成的梯形面积有向地叠加起来，即可得出最终结果。

```
//计算多边形面积
double ii(const point &p1, const point &p2) {
    return 0.5 * (p2.x - p1.x) * (p2.y + p1.y);
}
double area(const polygon x(int N,point a[]))
{
    if (n<=0) return 0.0;
    double sum = 0.0;
    a[n]=a[0];
    for (int i=0;i<n;++i)
    {
        sum+=(a[i].x-a[i + 1].x)*(a[i].y+a[i + 1].y)/2;
    }
    return sum>=0.0?sum:-sum;
}
```

7.3.2 格点数

7.3.1 节给出了两个计算多边形交、并面积的基础方法，本节介绍一种另类的“面积”，即格点数。

在图论里有 Pick 定理，即

$$S = n + \frac{s}{2} - 1$$

式中，S 是面积，n 为内格点数，s 为边格点数，仅限于顶点均是整点的简单多边形。

本节只讨论顶点在整点上的简单多边形，更一般的情况可以参照求面积的方式，把多边形的总格点数转化成有向梯形区域的格点数叠加。顶点是整点的多边形的边界格点数和内格点数代码如下：

```
//顶点是整点的多边形的边界格点数和内格点数
int gcd(int a,int b)
{
    return b==0?a:gcd(b,a%b);
}
int borderpoint (polygon x)
{
    int num=0;
    a[n]=a[0];
    for (int i=0;i<m;i++)
```

```
11              num+=gcd(abs(int(a[i+1].x-a[i].x)),abs(int(a[i+1].y-a[i].y)));
12              //取边上的格点数，为了不重复，取前不取后，或者取后不取前
13          return num;
14      }
15      int insidepoint(polygon x)
16      {
17          return int(area(x))+1-borderpoint(x)/2;
18      }
```

7.3.3 例题讲解

例 7-3 Triangle

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

用一个有序对(x,y)表示一个格点，其中 x 和 y 都是整数。给定一个三角形的顶点的坐标（恰好是格点），需要计算完全位于三角形内部的格点数（格点在三角形顶点和边上时则不计数）。

输入：

输入包含多组测试数据，每组数据包含 6 个整数 x_1、y_1、x_2、y_2、x_3、y_3,其中(x_1, y_1) 、(x_2, y_2)和(x_3, y_3)是给定三角形顶点的坐标。测试数据中的三角形都是非退化的（即具有大于 0 的面积），数据范围是$-15000 \leqslant x_1$、y_1、x_2、y_2、x_3、$y_3 \leqslant 1500$。测试数据的结尾以 $x_1= y_1= x_2= y_2= x_3=y_3=0$ 来结束，这组数据不进行处理。

输出：

对于每组测试数据，程序需要在一行输出该三角形内部格点数的数量。

样例输入：

0 0 1 0 0 1
0 0 5 0 0 5
0 0 0 0 0 0

样例输出：

0
6

题目来源：POJ 2954。

思路分析：

该题是 Pick 定理的应用，如果一个多边形的每个顶点都是由整点构成的，该多边形的面积为 S，多边形边上的整点为 L，内部的整点为 N，则有 $2S=2N+L-2$。

题目实现：

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

```
using namespace std;
int gcd(int a,int b)
{
    return b==0?a:gcd(b,a%b);
}
int x1,x2,y3,y2,z1,z2;      //由于 y1 在 math.h 中有定义，此处换成 y3
int main()
{
    while (~scanf("%d%d%d%d%d%d",&x1,&x2,&y3,&y2,&z1,&z2))
    {
        if (x1==x2==y3==y2==z1==z2==0)
            break;
        else
    {
        double sum=fabs(x1*y2-x2*y3+y3*z2-z1*y2+z1*x2-x1*z2);
        int n,n1,n2,n3;
        if (x1==y3) n1=fabs(y2-x2); else if (x2==y2) n1=fabs(y3-x1);
            else n1=fabs(gcd(x1-y3,x2-y2));
        if (z1==y3) n2=fabs(y2-z2); else if (z2==y2) n2=fabs(y3-z1);
            else n2=fabs(gcd(z1-y3,z2-y2));
        if (x1==z1) n3=fabs(z2-x2); else if (x2==z2) n3=fabs(z1-x1);
            else n3=fabs(gcd(x1-z1,x2-z2));
        n=n1+n2+n3;
        printf("%d\n",int(sum-double(n)/2+1));
    }
    }
}
```

7.4 凸包

在研究多边形的几何问题时，凸包是非常重要的，尤其是在解析几何、射影几何、离散几何中。为此，先介绍如何用代码实现凸多边形。

7.4.1 凸多边形

定义 7.1 凸包（Convex Hull） 在一个实数向量空间 $\boldsymbol{V}$ 中，对于给定集合 X，所有包含 X 的凸集的交集 S 被称为 X 的凸包。

定义 7.2 凸多边形（Convex Polygon） 可以有以下三种定义：

（1）没有任何一个内角是优角（Reflexive Angle）的多边形。

（2）如果在一个多边形的所有边中，有一条边向两方无限延长成为一直线时，其他各边都在此直线的同旁，那么这个多边形就称为凸多边形。

（3）凸多边形是一个内部为凸集的简单多边形。

简单多边形的下列性质与其凸性等价：

（1）所有内角小于 180°。

（2）任意两个顶点间的线段位于多边形的内部或边上。

（3）多边形内任意两个点，其连线全部在多边形内部或边上。

总之，选定一个点集，凸包就是包含这个点集的最小的凸多边形。那么如何判定凸多边形呢？可以利用定义 7.2.2，即按一个顺序（顺时针或逆时针）对任意相邻两条有向边进行叉积，必须都同号。

判断凸多边形时，顶点按一个顺序（顺时针或逆时针）给出，不允许相邻边共线，代码如下：

```
1   //判断是否凸多边形
2   int isconvex(polygon x)
3   {
4       flag=1;
5       for (i=0;i<n&&flag==0;i++)
6       flag=(sign(xmult(a[(i+1)%n]-a[i],a[(i+2)%n]-a[i])==
7               sign(xmult(a[(i+2)%n]-a[i+1],a[(i+3)%n]-a[i+2]));
8       return flag;
9   }
```

按顺序相邻的两条有向边进行叉积，如果相邻的两个叉积同号，flag 为 1，否则 flag 为 0，跳出循环。而凸包相邻的两个叉积一定同号。如果判断凸多边形时允许相邻边共线，则只需稍加修改代码：

```
1   //判断凸多边形，允许相邻边共线
2   int isconvex(polygon x)
3   {
4       flag=1;
5       for (i=0;i<n&&flag==0;i++)
6
7           flag=((sign(xmult(a[(i+1)%n]-a[i],a[(i+2)%n]-a[i])==sign(xmult(a[(i+2)%
8               n]-a[i+1],a[(i+3)%n]-a[i+2]))|(!xmult(a[(i+1)%n]-a[i],a[(i+2)%n]-a[i]))|(!sign(xmult
                (a[(i+2)%n]-a[i+1],a[(i+3)%n]-a[i+2]))));
9       return flag;
10  }
```

下面介绍判断点是否在凸多边形内的两种算法，在这之前，用代码重新实现一个凸多边

形类，目的是为了和一般的多边形类进行区分。

```
//定义凸多边形类
struct convex
{
    vector <point> P;
    convex(int size=0)
    {
        P.resize(size);          //将向量清空
    }
}
```

第一种方法仍可以用以前介绍的判断点在多边形内的方法。

```
//判断点是否在凸多边形内
#define next(i) ((i+1)%n)
bool contain(const convex &a,const point &b)
{
    int n=a.P.size();
    int _sign=0;
    for (int i=0;i<n;i++)
    {
        int x=sign(xmult(a.P[i]-b,a.P[next(i)]-b));
        if (x)
            if (_sign)
            {
                if (_sign!=x) return false;
            } else sign=x;
    }
}
```

第一种方法的算法时间复杂度为 $O(n)$ 。

第二种方法利用了凸多边形的性质，有一条边向两个方向无限延长成为一直线时，其他各边都在此直线的同侧。二分凸多边形，即二分后仍然保持凸多边形性质，只需要 $O(\log n)$ 的复杂度就能完成判断。

```
//利用凸多边形的性质，判断点是否在凸多边形内
int contain(const convex &a,const point &b)
{
    int n=a.P.size();
    point g=(a.P[0]+a.P[n/3]+a.P[2*n/3])/3.0;
    int l=0; r=n;
    while (l+l<r)
```

```
8           {
9               int mid=(l+r)/2;
10              if (sign(xmult(a.P[l]-g,a.P[mid-g]))>0) r=mid;
11              else l=mid;
12          }
13          else
14          {
15              if (sign(xmult(a.P[l]-g,b-g))<0 && sign(xmult(a.P[mid]-g,b-g))>=0) l=mid;
16              else r=mid;
17          }
18          r%=n;
19          int z=sign(xmult(a.P[r]-b,a.P[l]-b))-1;
20          if (z==-2) return 1;
21          return z;
22      }
```

例 7-4　Shape of HDU

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

创业是需要地盘的，HDU 向高新技术开发区申请一块用地，很快得到了批复，政府划拨的这块用地是一个多边形，为了描述它，用逆时针方向的顶点序列来表示，我们很想了解这块地的基本情况，现在请你编程判断 HDU 的用地是凸多边形还是凹多边形呢。

输入：

输入包含多组测试数据，每组测试数据占 2 行，第一行是一个整数 n，表示多边形顶点的个数；第二行是 $2\times n$ 个整数，表示逆时针顺序的 n 个顶点的坐标（x_i,y_i），n 为 0 的时候结束输入。

输出：

对于每组测试数据，如果地块的形状为凸多边形，请输出“convex”，否则输出“concave”，每组测试数据的输出占一行。

样例输入：

4 0 0 1 0 1 1 0 1 0

样例输出：

Convex

题目来源：HDU 2108。

思路分析：

应用叉积判断点是否在多边形内部，根据点输入的顺序，每次判断一条边和一个点的关系，若叉积>0，则表示存在大于 180° 的内角，即凹多边形。

题目实现：

```
#include<iostream>
#include<cstdio>
using namespace std;
#define MAXN 100005
const double eps=1e-8;
struct point
{
    double x,y;
    point() {}
    point(double a,double b)
    {
        x=a;
        y=b;
    }
    void input()
    {
        scanf("%lf%lf",&x,&y);
    }
    friend point operator - (const point &a,const point &b)
    {
        return point(a.x-b.x,a.y-b.y);
    }
};
double det(point a,point b)
{
    return a.x*b.y-a.y*b.x;
}
int cmp(double a)
{
    if (fabs(a)<eps) return 0;
    if (a>0) return 1;
    return -1;
}
int main()
{
    point a[MAXN];
    int n;
    bool flag=true;
    scanf("%d",&n);
```

```
40          for (int i=1;i<=n;i++)
41              a[i].input();
42          a[n+1]=a[1];
43          a[n+2]=a[2];
44          for (int i=3;i<=n+2;i++)
45          {
46              if (cmp(det(a[i-1]-a[i-2],a[i]-a[i-1]))<0) {flag=false;break;}
47          }
48          if (flag==false) printf("concave\n");
49          else printf("convex\n");
50      }
```

7.4.2 凸多边形的性质

本节介绍一些关于凸多边形的性质。

性质 7-1 如果两点属于该凸多边形，那么连接这两个点的整条线段都属于该凸多边形。这个性质是由凸包的定义直接得出的。

定理 7-1 Helly 定理 假设 F 是 E^2 中的一组凸多边形，如果其中的任意 3 个凸多边形都有公共点，那么 F 中任意多个凸多边形都有非空的交。

与 Helly 定理等价的还有 Caratheodory 定理和 Radon 定理。如果推广到 n 维可以发现，这三个定理从不同角度叙述了凸体和空间维数的潜在关系。感兴趣的读者可以去找一找相关的论文，由于涉及比较多的篇幅，这里不给出证明。

定时 7-2 John 定理 对每一个 E^2 凸多边形 K，都有一个相应的椭圆 E 满足

$$E \subseteq K \subseteq nE$$

其中，nE表示E中的每个点的坐标都乘n后得到的新的坐标点在nE中 。

对每一个中心对称的 E^2 凸多边形 C 都有一个相应的椭球 E^* 满足

$$E^* \subseteq C \subseteq nE^*$$

证明不太难，但有些技巧在其中，这里不做证明，有兴趣的读者可以参考 Banach 空间几何理论的相关资料。

7.4.3 构造凸包

凸包是包含给定点集的最小凸多边形。例如，三角形就是它的三个顶点的凸包，圆盘则是它的边界圆周的凸包。下面介绍构造一个点集的凸包的两种方法。

1．卷包裹算法（Gift Wrapping Algorithm）

卷包裹算法的复杂度是 $O(n^2)$，核心思想就是枚举法。取定第一个点，可以找 x 坐标最小的、y 坐标最小的或者极角最小的点，分别对应水平序和极角序。找第二个点作为凸包的顶点，

根据凸包的性质，它首先是个凸体，所以需要找“最外围”的点，把这个点作为第二个点，再这样继续进行下去，如图 7.3 所示，当又一次找到的点刚好是第一个点作为“最外围”的点时结束。

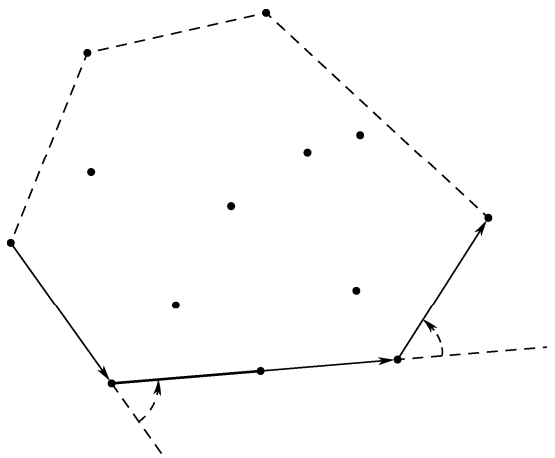


图 7.3　卷包裹算法

最坏情况每一次都要遍历所有剩余的点才能找到“最外围”的点，所以时间复杂度是 $O(n^2)$，当然最好情况的时间复杂度是 $O(n)$。

现在的问题是如何判断“最外围”的点，利用叉积的性质，不断找最左或者最右的点即可。通过两重循环可以很好地实现（以逆时针为例），代码如下：

```
#include<bits/stdc++.h>
#define MAXN 1000005
const double eps=1e-8;
using namespace std;
struct point
{
    double x,y;
    point() {}
    point(double a,double b)
    {
        x=a;
        y=b;
    }
    friend point operator - (const point &a,const point &b)
    {
        return point(a.x-b.x,a.y-b.y);
```

```
17          }
18      };
19      point p[MAXN];
20      vector<point> a;   //现有的点存储在数组 a 中
21      double det(point a,point b)
22      {
23          return a.x*b.y-a.y*b.x;
24      }
25      int cmp(double a)
26      {
27          if (fabs(a)<eps) return 0;
28          if (a>0) return 1;
29          return -1;
30      }
31      void gift()
32      {
33          p[0]=a[0];
34          p[1]=a[1];
35          int l=1;
36          int j=1;
37          while (l!=0)
38          {
39              point k=a[0]-p[j];
40              int l=0;
41              for (int i=1;i<=a.size();i++)
42                  if (det(p[j]-p[j-1],a[i]-a[j])>0)
43                  {
44                      if (det(k,a[i]-p[j])<0)
45                      {
46                          k=a[i]-p[j];
47                          l=i;
48                      }
49                  }
50                  p[j++]=a[l];     //凸包的顶点存储在数组 p 中
51          }
52      }
```

下面介绍一种效率更高的算法。

2．Graham 扫描算法（Graham Scan Algorithm）

Graham 扫描算法的核心是维护一个凸壳，通过不断在凸壳中加入新的点和去除影响凸性

的点，最后形成凸包，如图 7.4 所示。而要维护这个性质，就需要对数组进行排序，然后不断维护这个顺序的性质。现在有两种排序方法：极角序和水平序。

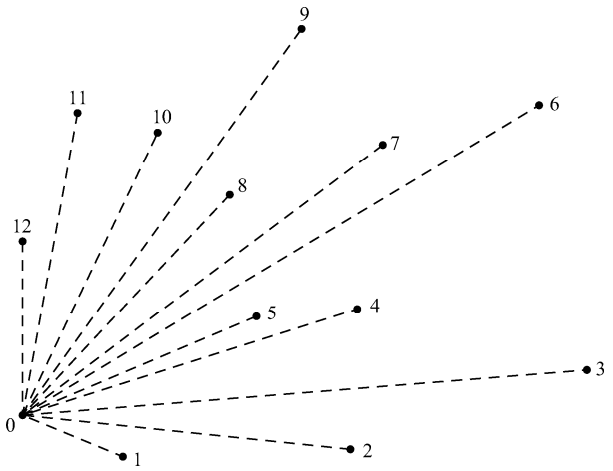


图 7.4　Graham 扫描算法示意图

极角序排起来比较方便，代码也容易理解。由于涉及除法运算，可能对结果的精度造成影响，而且极角序对于斜率相等的点的排序比较复杂，因此这里用没有精度损失的方法，即水平序。水平序需要正向和反向各扫一遍，得到左链和右链，再把两个链合起来。代码如下：

```
1   //Graham 扫描算法
2   convexhull(vector<point> a)
3   {
4       convex res(2*a.size()+5);
5       sort(a,begin(),a.end(),a.end());
6       int m=0;
7       for (int i=0;i<a.size();++i)
8       {
9           while (m>1 && sign(xmult(res.P[m-1]-res.P[m-2],a[i]-res.P[m-2]))<=0)
10              m--;
11          res.P[m++]=a[i]
12      }
13      int k=m;
14      for (int i=int(a.size())-2;i>=0;--i)
15      {
16          while (m>k && sign(xmult(res.P[m-1]-res.P[m-2],a[i]-res.P[m-2]))<=0)
17              --m;
18          res.P[m++]=a[i];
```

```
19          }
20          res.P.resize(m);
21          if (a.size()>1) res.P.resize(m-1);
22          return res;
23      }
```

介绍一下算法实现的过程，如图 7.5 所示的极角序，先选定一个点，用快速排序将其他点按 x 坐标（或者 y 坐标）从小到大进行排序，时间复杂度是 $O(n\log n)$。接下来开始由小到大地扫点，如果每次保证下一个点在这个点左侧（或者右侧），该点进入 vector 数组，否则弹出该点，直到保证下一个点在 vector 最后一个点的左侧（或者右侧）为止。由于每个点只能弹出或者进入一次，所以时间复杂度是 $O(n)$，因此总体时间复杂度为 $O(n\log n)$。

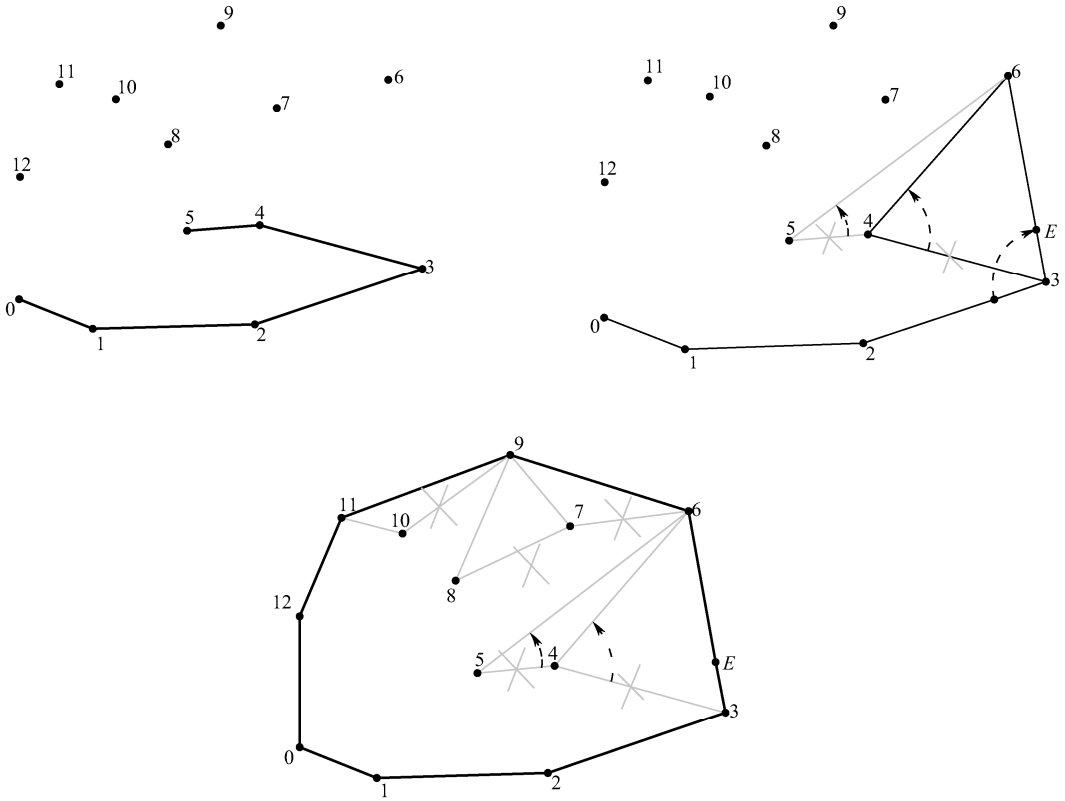


图 7.5　极角序

7.4.4　例题讲解

例 7-5　Muddy Field

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

有一块奶牛进食的矩形区域，有 R 行 C 列（$1 \leqslant R \leqslant 50$，$1 \leqslant C \leqslant 50$）。天刚刚下过雨，这有利于草的生长，然而奶牛很爱整洁，它们不愿意在吃草的时候弄脏自己的脚。

为了防止这种事情发生，牧场主将会在泥泞的地上摆放一些木板，每块木板的宽度为 1，长度任意，并且摆放的方式为严格平行于边界。

牧场主想要最小化木板的数量，并且木板不能覆盖草地，但可以重叠。

请计算木板的最小数量。

输入：

第一行输入两个数，即 R 和 C。

接下来 R 行用于描述地图，每行 C 个字符，其中“*”代表空地，“.”代表草地。

输出：

最小的木板数。

样例输入：

4 4

..

.***

***.

..*.

样例输出：

4

题目来源：POJ 2226。

解题思路：

先进行行列建图，将横着的木板作为二分图中一侧的点，竖着的木板作为另一侧，定义出二分图。对于每一个“*”的点，考虑横着的木板如何覆盖它，竖着的木板如何覆盖，如何定义横着覆盖它的木板的编号。其实可以把每一个需要覆盖的顶点所在的泥地上的最左端的顶点作为横着木板的编号，所在泥地最上端的顶点的木板是竖着的，将最左端的顶点和最上端的顶点连边建立二分图，最后求最小的顶点覆盖，这就可以保证每个泥地都被横着的木板或者竖着的木板覆盖。

题目实现：

```
1    #include<iostream>
2    #include<cstring>
3    #include<cstdio>
4    #include<vector>
5
6    using namespace std;
```

```
7
8   const int maxn=55;
9   const int maxv=5005;
10  vector<int> g[maxv];
11
12  int from[maxv],tot;
13  bool used[maxv];
14
15  bool match(int x) {
16      for(int i=0; i<g[x].size(); i++) {
17          int v=g[x][i];
18          if(!used[v]) {
19              used[v]=1;
20              if(from[v]==-1||match(from[v])) {
21                  from[v]=x;
22                  return true;
23              }
24          }
25      }
26      return false;
27  }
28
29  int hungry() {
30      tot=0;
31      memset(from,-1,sizeof(from));
32      for(int i=0; i<maxv; i++) {
33          memset(used,0,sizeof(used));
34          if(match(i))
35              tot++;
36      }
37      return tot;
38  }
39  char mp[55][55];
40  int main() {
41      //freopen("in.txt","r",stdin);
42      int n,m;
43      while(scanf("%d%d",&n,&m)!=EOF){
44          for(int i=0;i<n;i++){
45              scanf("%s",mp[i]);
46          }
47          for(int i=0;i<maxv;i++)g[i].clear();
```

```
48              for(int i=0;i<n;i++){
49                  for(int j=0;j<m;j++){
50                      if(mp[i][j]=='*'){
51                          int y=i,x=j;
52                          while(y>0&&mp[y-1][j]=='*')--y;
53                          while(x>0&&mp[i][x-1]=='*')--x;
54                          g[y * 50 + j].push_back(i * 50 + x + 2500);
55                      }
56                  }
57              }
58              int ans=hungry();
59              printf("%d\n",ans);
60          }
61          return 0;
62      }
63                          while(x>0&&mp[i][x-1]=='*')--x;
64                          g[y * 50 + j].push_back(i * 50 + x + 2500);
65                      }
66                  }
67              }
68              int ans=hungry();
69              printf("%d\n",ans);
70          }
71          return 0;
72      }
```

例 7-6　Grandpa's Estate

Time Limit: 5000/5000 ms (Java/Others)　　Memory Limit: 65536/65536 KB (Java/Others)

题目描述：

作为祖父唯一活着的后代，卡姆兰继承了祖父所有的物品，其中最具有价值的就是祖父曾经生活的村子留下的一个凸多边形农场。祖父的农场通过一条粗大的绳子挂在一些钉子上形成的一个凸多边形，从而与附近的农场分隔开。但是当卡姆兰去查看农场时，他发现绳子和一些钉子都丢失了。你的任务就是编写一个程序去帮助卡姆兰判断农场的边界是不是可以只由这些剩下的钉子来确定。

输入：

输入的第一行只包含一个整数 t（$1 \leqslant t \leqslant 10$），代表有多少组测试数据。接下来是每组的测试数据，测试数据每组的第一行包含一个整数 n（$1 \leqslant n \leqslant 1000$），表示有多少个剩下的钉子。接下来的 n 行，每一行输入一个整数对(x,y)，表示每个钉子的 x 轴坐标和 y 轴坐标。

输出：

每组测试数据输出一行，其中内容是 YES 或 NO，具体取决于农场的边界是否可以根据输入的钉子坐标唯一确定。

样例输入：

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

样例输出：

NO

题目来源：POJ 1228。

解题思路：

给定若干个点，问这些点作为顶点构成的凸包是否稳定。所谓稳定，是指不能找到至少一个点加入这些顶点中，这些点为新的凸包顶点，而新的凸包包含原来的凸包。

可以发现，如果一个凸包稳定，那么每条边上至少有 3 个点存在，如果凸包上存在一条边上的点只有 2 个点，这个凸包是不稳定的，一定能找到一个点作为新加入的点，使凸包变大，因为如果还有一个点为新加入的点，那么凸包就变成凹多边形了。因此只需要做出凸包，然后判断是否一条边上有 3 个点，是的话就输出 YES，否则输出 NO。

题目实现：

```
#include <iostream>
#include <algorithm>
#include <stdio.h>
#include <math.h>
#define MAXN 1005
using namespace std;
const double eps=1e-8;
int cmp(double a)
{
    if(fabs(a)<0) return 0;
    if (a>0) return 1;
    return -1;
}
```

```
14  struct point
15  {
16      double x,y;
17      point() {}
18      point(double a,double b)
19      {
20          x=a;
21          y=b;
22      }
23      void input()
24      {
25          scanf("%lf%lf",&x,&y);
26      }
27      friend point operator - (const point &a,const point &b)
28      {
29          return point(a.x-b.x,a.y-b.y);
30      }
31  };
32  point a[MAXN],b[MAXN];
33  int s;    //记录凸包顶点数
34  int det(point a,point b)
35  {
36      return a.x*b.y-a.y*b.x;
37  }
38  int comp(point a,point b)
39  {
40      return cmp(a.x-b.x)<0 || (cmp(a.x-b.x)==0 && cmp(a.y-b.y)<0);
41  }
42  void hull(int n)
43  {
44      sort(a+1,a+n+1,comp);
45      int m=1;
46      for (int i=3;i<=n;i++)
47      {
48          while(m>2 && cmp(det(b[m-1]-b[m-2],a[i]-b[m-1]))<0) --m;
49          b[m++]=a[i];
50      }
51      int k=m;
52      for (int i=n-1;i>0;i--)
53      {
54          while(m>k && cmp(det(b[m-1]-b[m-2],a[i]-b[m-1]))<0) --m;
```

```
55              b[m++]=a[i];
56          }
57          s=m;
58          if (n>1) s=m-1;
59      }
60      bool cross(point a,point b,point c)
61      {
62          if (cmp(det(a-b,a-c))==0) return true;
63          return false;
64      }
65      bool judge()                    //判断凸包的每条边上是否至少有 3 个点
66      {
67          b[0]=b[s];
68          b[s+1]=b[1];
69          b[s+2]=b[2];
70          for(int i=1;i<=s;i++)
71          {
72
73              if((cross(b[i-1],b[i+1],b[i]))!=0&&(cross(b[i],b[i+2],b[i+1]))!=0)
74                  return false;
75          }
76          return true;
77      }
78      int main()
79      {
80          int t;
81          scanf("%d",&t);
82          while (t--)
83          {
84              s=0;
85              int n;
86              memset(a,0,sizeof(a));
87              memset(b,0,sizeof(b));
88              scanf("%d",&n);
89              for (int i=1;i<=n;i++)
90                  a[i].input();
91              hull(n);
92              if (judge()) printf("YES\n");
93              else printf("NO\n");
94
95          }
```

```
96  }
```

例 7-7　SCUD Busters

Time Limit: 5000/5000 ms (Java/Others)　　Memory Limit: 65536/65536 KB (Java/Others)

题目描述：

在一个 500 km×500 km 的平坦世界，这个平坦世界中由几个交战王国组成。每个王国都被设计为保护王国和隔离它的高墙围绕（但厚度可忽略）。为了避免争夺权力，每个王国都有自己的发电厂。

当无法抑制过大的战斗欲望时，王国的人们往往向其他王国发射导弹。落在王国城墙内的每颗导弹将破坏该王国的发电厂（不会造成生物的死亡）。给定几个王国的坐标（包括王国中房屋和发电厂的位置）以及导弹轰炸点的坐标，请编写一个程序，确定在导弹爆炸后所有没有电力的王国的总面积。

输入：

先输入一系列王国的数据，然后输入一系列导弹轰炸位置。

每个王国的输入数据中，第一行的单个整数 N（$3 \leqslant N \leqslant 100$）表示该王国的建筑数量（发电厂和房屋）。下一行有一个数对，表示发电厂的 x 和 y 坐标；随后的 $N-1$ 行，每一行有一个数对，表示该发电厂服务的家庭的 x 和 y 坐标，如果 N 的值为−1 表示后面不再有王国。输入的数据中至少有一个王国。

紧接着是一次或多次导弹攻击的坐标，表明导弹着陆的位置。每个导弹的位置占一行。你要处理导弹攻击，直到文件末尾。

每个位置都限定在 500 km×500 km 的网格中，所有的坐标都取自 0 到 500 之间的整数。坐标被指定为由一行上的空格分隔的一个整数对。

输入文件最多包含 20 个王国，以及任意数量的导弹。

输出：

输出一个数字，表示所有导弹爆炸后没有电力的王国的总面积，保留两位小数。

样例输入：

```
12
3 3
4 6
4 11
4 8
10 6
5 7
6 6
6 3
```

7 9
10 4
10 9
1 7
5 20
20 20
40 40
20 40
40 30
30 3
10 10
21 10
21 13
-1
5 5
20 12

样例输出：

70.50

题目来源：UVA 109。

解题思路：

本题十分长，也十分的复杂，是计算几何的综合问题。这个问题分成三个子问题，构造出凸包、计算出多边形面积、判断点是否在其中。

题目实现：

```
#include<bits/stdc++.h>
#define MAXN 105
using namespace    std;
const double eps=1e-8;
struct point
{
        double x,y;
        point() {}
        point(double a,double b)
        {
                x=a;
                y=b;
        }
```

```
14          void input()
15          {
16              scanf("%lf%lf",&x,&y);
17          }
18          friend point operator - (point a,point b)
19          {
20              return point(a.x-b.x,a.y-b.y);
21          }
22      };
23      point a[MAXN],b[MAXN][MAXN],c[MAXN];
24      int d[MAXN],e[MAXN];
25      int cmp(double a)
26      {
27          if(fabs(a)<0) return 0;
28          if (a<0) return -1;
29          return 1;
30      }
31      double det(point a,point b)
32      {
33          return a.x*b.y-a.y*b.x;
34      }
35      bool comp(const point &a,const point &b)
36      {
37          return cmp(a.x-b.x)<0 || (cmp(a.x-b.x)==0 && cmp(a.y-b.y)<0);
38      }
39      void    hull(int n)
40      {
41          sort(a,a+d[n],comp);
42          int k=0;
43          b[n][0]=a[0];
44          for (int j=1;j<d[n];j++)
45          {
46              while (k>0 && cmp(det(b[n][k]-b[n][k-1],a[j]-b[n][k-1]))<=0) --k;
47              b[n][++k]=a[j];
48          }
49          int l=k;
50          for (int j=d[n]-2;j>=0;j--)
51          {
52              while (k>l && cmp(det(b[n][k]-b[n][k-1],a[j]-b[n][k-1]))<=0) --k;
53              b[n][++k]=a[j];
54          }
```

```
55          d[n]=k;
56      }
57      double area(int n)
58      {
59          double sum=0;
60          for (int j=0;j<d[n];j++)
61              sum+=det(b[n][j],b[n][j+1]);
62              return sum/2;
63      }
64      bool contain(int n,point p)
65      {
66          #define next(i) ((i+1)%n)
67          int sign=0;
68          for (int i=0;i<d[n];i++)
69          {
70              int q=cmp(det(b[n][i]-p,b[n][next(i)]-p));
71              if (q)
72              {
73                  if (sign)
74                  {
75                      if (sign!=q) return false;
76                  }else sign=q;
77              }
78          }
79          return true;
80      }
81
82      int main()
83      {
84          int N;
85          int t1=1;
86          int t2=1;
87          while (scanf("%d",&N)!=-1)
88          {
89              for (int i=0;i<N;i++)
90                  a[i].input();
91                  d[t1]=N;
92                  hull(t1);
93                  t1++;
94          }
95          while (~scanf("%lf%lf",&c[t2].x,&c[t2].y))
```

```
96              t2++;
97          for (int i=1;i<t2;i++)
98          {
99              for (int j=1;j<t1;j++)
101                 if (!e[j] && contain(j,c[i]))    e[j]=1;
102         }
103         double ans=0;
104         for (int j=1;j<t1;j++)
105             if (e[j]) ans+=area(j);
106         printf("%.2lf",ans);
107 }
```

7.5 相交问题

在 7.3 节中讨论多边形的相交问题时，多边形是一般多边形，所使用的算法核心思想就是枚举法，但当对多边形进行特殊化后，就能得到新的结论和算法。

7.5.1 半平面交

根据解析几何的知识可知，一个半平面可以用一个一次不等式来表示，即 $ax+by+c\leqslant 0$，因此可以构造一个半平面类。

```
1   //构造半平面类
2   struct halfplane
3   {
4       double a,b,c;
5       halfplane(point p,point q)
6       {
7           a=p.y-q.y;
8           b=q.x-p.x;
9           c=xmult(p,q);
10      }
11      halfplane(double aa,double bb,double cc)
12      {
13          a=aa;
14          b=bb;
15          c=cc;
16      }
17  }
```

半平面的参数是 a、b、c，将一个点 (x,y) 代入上述关系中，可通过通过值的正负性来判断点和半平面的位置关系。

```
//判断点与半平面的位置关系
double cal(halfplane &L,point &a)
{
    return a.x*L.a+a.y*L.b+L.c;
}
```

接下来得到两点连线和半平面的交点。

```
//两点连线和半平面的交点
point Labi(point &a,point &b,halfplane %)
{
    point res;
    double t1=Labi(L,a), t2=Labi(L,b);
    res.x=(t2*a.x,t1*b.x)/(t2-t1);
    res.y=(t2*a.y-t1*b.y)/(t2-t1);
    return res;
}
```

现在，半平面起作用了，如果给定一个多边形，用一个半平面去截它，如何求它们的交，即新的多边形呢？类似的问题曾在 7.3 节提到过，当时仅仅描述了思路，将多边形换成半平面后，思路就清晰了许多。求半平面和多边形的交的代码如下：

```
//半平面和多边形的交
convex cut(convex %a,halfplane &L)
{
    int n=a.P.size()
    convex res;
    for (int i=0;i<n;i+)
    {
        if (cal(L,a.P[i])<-eps)
            res.P.push_back(a,P[i]);
        else
        {
            int j;
            j=i-1;
            if (j<0)
                j=n-1;
            if (cal(L,a.P[j])<-eps)
                res.P.push_back(Labi(a.P[j],a.P[i],L));
            j=i+1;
```

```
19                      if (j==n)
20                          j=0;
21                      if (cal(L,a.P[j])<-eps)
22                          res.P.push_back(Labi(a.P[i],a.P[j],L));
23                  }
24              }
25              return res;
26      }
```

Vector 数组在这种未知数组个数的情况下的使用十分方便，但缺点是不容易让别人快速看懂自己写的代码。

7.5.2 凸多边形交

定义 7.3 凸多边形核 多边形的核是多边形内部的一个点集，该点集中任意一点与多边形边界上一点的连线都处于这个多边形内部。

回到 7.3 节中最后问题，对于两个凸多边形，只要将每个多边形分解为一组半平面，就能套用半平面交的算法求出凸多边形的交，时间复杂度为 $O(n\log n)$。但是还存在一个问题，拆分多边形的时间复杂度显然是线性的，而半平面交的时间复杂度最高的部分就是排序，时间复杂度为 $O(n\log n)$，如果能够换一种更优化的排序方法，是不是就能得出更低的时间复杂度的算法，答案是肯定的。因为凸多边形的有序边的斜率在同余的意义下是单调的，所以可以用线性归并的排序方法排序，最终的时间复杂度仅为 $O(n)$。

7.5.3 例题讲解

例 7-8 Uyuw's Concert

Time Limit: 1000/1000 ms (Java/Others) Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

王子 Remmarguts 成功地解决了一局棋的难题。作为嘉奖，Uyuw 计划在一个巨大的以伟大设计师 Ihsnayish 命名的广场上举办音乐会 。

这个广场坐落在自由联合州上，自由联合州是一个[0, 10000]×[0, 10000]的矩形。有一些长椅已经伫立在上面很多年了，但是一塌糊涂。

在图 7.6 有 3 条长椅，而且长椅的方向用箭头标出。这些长椅已经年代久远很难被移动了。王子 Remmarguts 告诉这个广场的拥有者 UW 先生，他需要在广场中要建一个大的舞台。这个舞台必须尽可能大，但是他需要每个位置的每个

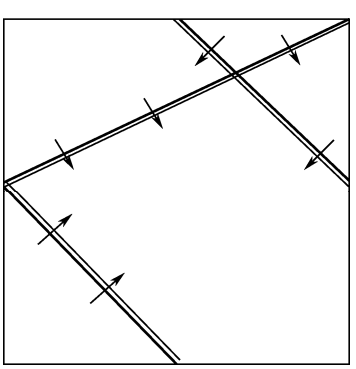


图 7.6 广场的场椅

长椅上的观众能够在不转身的前提下看到舞台，也就是说舞台只能建在每个观众面朝的那个方向。

为了简单起见，舞台可以设置得足够高，以确保即使数以千计的长椅在你面前，只要你面对舞台，就能看到 Uyuw。

作为一个疯狂的偶像崇拜者，你能告诉他们舞台的最大的尺寸吗？

输入：

在第一行中，有一个非负整数 N（N≤20000），表示长椅的数量。以下每一行包含四个浮点数 x_1、y_1、x_2、y_2，这意味着(x_1,y_1)–(x_2,y_2)的线段上有一个长椅，并且面向它的左侧，即一个点(x,y)在该段的左侧意味着$(x-x_1)\times(y-y_2)-(x-x_2)\times(y-y_1)>=0$。

输出：

输出一个浮点数，保留一位小数，该数字表示舞台的最大面积。

样例输入：

4 4

3

10000 10000 0 5000

10000 5000 5000 10000

0 5000 5000 0

样例输出：

54166666.7

题目来源：POJ 2451。

解题思路：

题目的意思是给定 n 个半平面，求它们的交。显然我们不能暴力枚举相交点，可以用分治法，因为保证了半平面的交一定为凸的。这里给出了排序增量法，通过维护某个结构来得到结果。

用一个向量的左侧（或右侧）来描述一个半平面，将半平面按极角排序，这里最好用计算叉积的方法排序，避免因为使用反三角函数丢失精度（当然大多数丢失精度是在所难免的）。对于极角相同的平面（即平行平面），我们只留最左边的（平行平面的公共部分）那个平面。

接下来要维持一个双端队列。首先，将两个平面加入队列，按顺序插入每一个半平面，以这种方式来维护这个双端队列：在插入 p_i 半平面前，判断双端队列队尾的两个半平面的交是否在 p_i 中，不在则删除底端的半平面；再判断双端队列对首的两个半平面的交是否在 p_i 中，不在则删除顶端（顶部）的半平面。当所有的半平面插入完毕，再做一次处理，判断顶端两个半平面的交是否在底端的半平面中，不在则删除顶端半平面；判断底端两个半平面的交是否在顶端的半平面中，不在则删除底端半平面。当处理好了这些，队列中剩下的半平面即要求交的半平面，再依次按顺序求交点，这些交点生成一个凸多边形，该凸多边形即所求的答案。在以上过

程中，如果队列中只剩下一个元素，则立刻停止循环。算法的时间复杂度为$O(n\log n)$。

题目实现：

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<vector>

using namespace std;

typedef complex<double> point;
typedef pair<point, point> halfplane;
const double eps=1e-10;
const double inf=1000;
int sgn(double n)
{
        return fabs(n)<eps ? 0 : (n<0?-1:1);
}
double cross(point a,point b)
{
        return (conj(a)*b).imag();
}
double dot(point a, point b)
{
        return (conj(a)*b).real();
}
double satisfy(point a, halfplane p)
{
        return sgn(cross(a-p.first,p.secod-p.first))<=0;
}
point crosspoint(const halfplane &a, const halfplane &b)
{
        double k=cross(b.first-b.second,a.first-b.second);
        k=k/(k-cross(b.first-b.second,a.second-b.second));
        return a.first+(a.second-a.first)*k;
}
bool cmp(const halfplane &a,const halfplane &b)
{
        int res=sgn(arg(a.second-a.frist)-arg(b.second-b.frist));
        return res==0 ? satisfuy(a.first,b) : res<0;
}
```

```
vector<point> halfplaneintersection(vector<halfplane> v)
{
    sort(v.begin(),v.end(),cmp);
    deque<halfplane> q;
    dequepoint< ans;
    q.push_back(v[0]);
    for (int i=1;i<int(v.size());i++)
    {
        if (sgn(arg(v[i].second-v[i].first)-arg(v[i-1].second-v[i-1].first))==0)
        {
            continue;
        }
        while (ans.size()>0 & !satisfy(ans.back(),v[i]))
        {
            ans.pop_back();
            q.pop_back;
        }
        while (ans.size()>0 & !satisfy(ans.front(),v[i]))
        {
            ans.pop_front();
            q.pop_front();
        }
        ans.push_back(crosspoint(q.back(),v[i]));
        q.push.back(v[i]);
    }
    while (ans.size()>0 && !satisfy(ans.back(),q.front()))
    {
        ans.pop_back();
        q.pop_back();
    }
        while (ans.size()>0 && !satisfy(ans.front(),q.back()))
    {
        ans.pop_front();
        q.pop_front();
    }
    ans.push_back(crosspoint(q.back(),q.front()));
    return vector<point>(ans.begin(),ans.end());
}
```

例 7-9　How I Mathematician Wonder What You Are

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

艾萨克在童年时经常数天空中无穷无尽的星星，现在作为一个天文学家和数学家，借助一个巨大的天文望远镜使他的思绪融入太空，思考着如何写一个程序，来计算数不尽的星星。程序中最难的部分是如何判断天空中闪着亮光的物体是否真正的星星。作为一个严谨的数学家，唯一的方法就是用星形的数学定义来判断是否真正的星星。

星形的数学定义是：一个平面图形 F 是星形的，当且仅当如果存在一点 C，满足任意一点 $P \in F$，使得 C 和 P 连成的线段都含于 F，就称 C 是 F 的中心。再来看下面的星形例子，如图 7.7 所示。

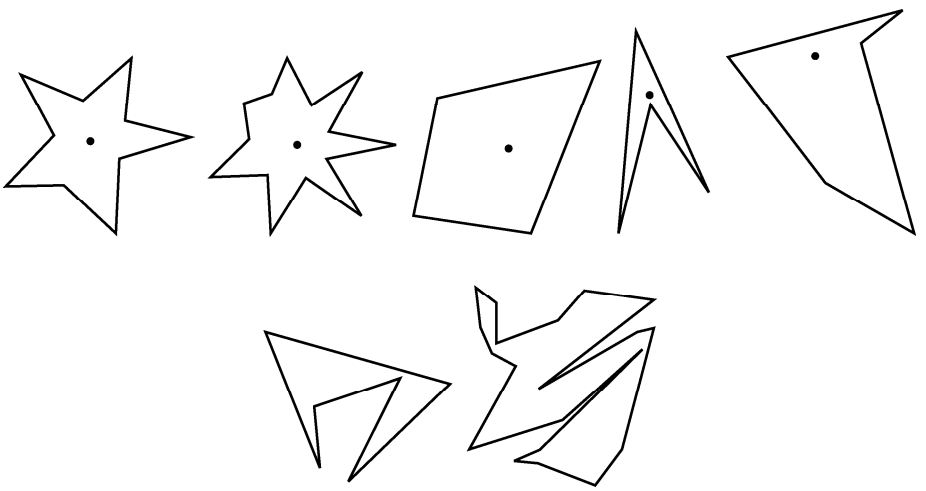


图 7.7　星形

图 7.7 中前两个是通常所说的星形。但是，根据上述定义，第一排的所有形状都属于星形，第二排中的两个则不是。对于每个星形，中心用黑点标出。请注意，星形通常具有无限多的中心。例如，对于第三个四边形形状，其中的所有点都是中心。你的工作是编写一个程序，判断一个给定的多边形是否星形。

输入：

输入一系列的数据，最后一行只有一个数字 0。每组数据都给定了一个多边形，格式如下。

n

x_1　y_1

x_2　y_2

…

x_n　y_n

每组数据中，第一行是顶点的数目 n（$4 \leqslant n \leqslant 50$），随后的 n 行是 n 个顶点的 x 和 y 坐标，它们是整数且 $0 \leqslant x_i \leqslant 10000$ 和 $0 \leqslant y_i \leqslant 10000$（$i=1,\cdots,n$）。沿逆时针方向，线段$(x_i, y_i)$–$(x_i+1, y_i+1)$（$i=1,\cdots, n-1$）和线段$(x_n, y_n)$–$(x_1,y_1)$构成多边形的边界。也就是说，这些线段在其方向左侧是多边形的内部。可以假设多边形是简单的，也就是说，它的边界永远不会穿过或接触到它。不妨假设，即使在无限延伸的情况下，多边形的任意三条边都不会交于同一个点。

输出：

对于每组数据，如果多边形是星形，输出 1，否则输出 0。每个数字必须在一个单独的行中，并且该行不应包含任何其他字符。

样例输入：

```
6
66 13
96 61
76 98
13 94
4 0
45 68
8
27 21
55 14
93 12
56 95
15 48
38 46
51 65
64 31
0
```

样例输出：

```
1
0
```

题目来源：POJ 3130。

解题思路：

多边形的核就是一个集合（即点集），该集合上的每个点与多边形的顶点相连的线段与多边形的边不相交。可以换个角度考虑，多边形核内的每个点都在多边形上每条边朝向多边形内侧的那一侧，也就是多边形的核可以看成每条边所表示的半平面相交的区域。

题目实现：

```
#include<bits/stdc++.h>
using namespace std;
typedef complex<double> point;
typedef pair<point,point> halfplane;
const double eps=1e-10;
const int inf=10005;
int sign(double n)
{
    return fabs(n)<eps?0:(n<0?-1:1);
}
double det(point a,point b)
{
    return (conj(a)*b).imag();
}
double dot(point a,point b)
{
    return (conj(a)*b).real();
}
int satisfy(point a,halfplane p)
{
    return sign(det(a-p.first,p.second-p.first))<=0;
}
point crosspoint(const halfplane &a,const halfplane &b)
{
    double k=det(b.first-b.second,a.first-b.second);
    k=k/(k-det(b.first-b.second,a.second-b.second));
    return a.first+(a.second-a.first)*k;
}
bool cmp(const halfplane &a,const halfplane &b)
{
    int res=sign(arg(a.second-a.first)-arg(b.second-b.first));
    return res==0 ? satisfy(a.first,b) : res<0;
}
vector<point> halfplaneintersection(vector<halfplane> v)
{
    sort(v.begin(),v.end(),cmp);
    deque<halfplane> q;
    deque<point> ans;
    q.push_back(v[0]);
```

```
40
41          for (int i=1;i<int(v.size());i++)
42          {
43              if (sign(arg(v[i].second-v[i].first)-arg(v[i-1].second-v[i-1].first))==0)
44              {
45                  continue;
46              }
47              while (ans.size()>0 && !satisfy(ans.back(),v[i]))
48              {
49                  ans.pop_back();
50                  q.pop_back();
51              }
52              while (ans.size()>0 && !satisfy(ans.front(),v[i]))
53              {
54                  ans.pop_front();
55                  q.pop_front();
56              }
57              ans.push_back(crosspoint(q.back(),v[i]));
58              q.push_back(v[i]);
59          }
60          while (ans.size()>0 && !satisfy(ans.back(),q.front()))
61          {
62              ans.pop_back();
63              q.pop_back();
64          }
65          while (ans.size()>0 && !satisfy(ans.front(),q.back()))
66          {
67              ans.pop_front();
68              q.pop_front();
69          }
70          ans.push_back(crosspoint(q.back(),q.front()));
71          return vector<point> (ans.begin(),ans.end());
72      }
73      int n;
74      int main()
75      {
76          vector<halfplane> h;
77          vector<point> r;
78          while (~scanf("%d",&n))
79          {
80              point s[inf];
```

```
81              double a[inf],b[inf];
82              if (n>=4)
83              {
84                  for (int i=0;i<=n-1;i++)
85                  {
86                      scanf("%lf%lf",&a[i],&b[i]);
87                      s[i]=point(a[i],b[i]);
88                  }
89                  for (int i=0;i<=n-2;i++)
90                      h.push_back(halfplane(s[i],s[i+1]));
91                  h.push_back(halfplane(s[n-1],s[0]));
92                  r=halfplaneintersection(h);
93                  if (r.size()>1) printf("1\n");
94                  else printf("0\n");
95              }
96          }
97          return 0;
98      }
```

7.6 圆

涉及圆的计算几何问题，既可以是最简单的问题，又可以是最复杂的问题，这是由圆本身的性质所决定的。圆仅仅由两个参量便能确定，圆心 o 和半径 r，但同时它又是正多边形的无穷逼近，所以涉及圆计算和精度的问题，通常不能用一般的求解多边形的思路去考虑（除非精度要求不高）。因此涉及圆的问题和算法层出不穷，本节仅简单地介绍其中一部分问题。

7.6.1 圆与线段的交

涉及圆与线段的交的问题，就不能再用多边形的方法了，因为用线段逼近圆必定会牺牲一部分的精度。将线段 $\overline{AB}$ 写成参数方程 $\vec{P}=\vec{A}+t\times\left(\vec{A}-\vec{B}\right)$，代入圆的方程，解出 t（如果是实数），就得到线段所在的直线与圆的交点。如果 $0\leqslant t\leqslant 1$ 则说明点在线段上，否则不在。

```
1   //圆与线段的交
2   void ccl(point a,point b,point o,double r,point ret[],int &num)
3   {
4       double x0=o.x, y0=o.y;
5       double x1=a.x, y1=a.y;
6       double x2=b.x, yw=b.y;
```

```
7           double dx=x2-x1,dy=y2-y1;
8           double A=dx*dx+dy*dy;
9           double B=2*dx*(x1-x0)+2*dy*(y1-y0);
10          double C=sqr(x1-x0)+sqr(y1-y0)-sqr(r);
11          double delta=B*B-4*A*C;
12          num=0;
13          if (dcmp(delta)>=0)
14          {
15                  double t1=(-B-sqrt(delta))/(2*A);
16                  double t2=(-B+sqrt(delta))/(2*A);
17                  if (dcmp(t1-1)<=0 && dcmp(t1)>=0)
18                  {
19                          ret[num++]=point(x1+t1*dx,y1+t1*dy);
20                  }
21                  if (dcmp(t2-1)<=0 && dcmp(t2)>=0)
22                  {
23                          ret[num++]=point(x1+t2*dx,y1+t2*dy);
24                  }
25          }
26  }
```

将 t_1 和 t_2 代入 $\vec{P}=\vec{A}+t\times\left(\vec{A}-\vec{B}\right)$ 即得到圆与直线的交点，如果 t_1 和 t_2 存在（实数），表示有两个相同或者不同的实交点。

7.6.2 圆与多边形的交的面积

首先需要明确这个面积到底如何来求。前面已经说过了，肯定不能用极限的方法去求精确的值，否则一定会有精度丢失。我们知道，圆的面积就是扇形（圆也是特殊的扇形），因此可以将圆与多边形交的面积看成扇形和其他多边形的面积和的形式。

以原点为中心将多边形进行三角剖分，这样只需要求三角形和圆的面积就可以了。根据不同的多边形的边 *AB* 和圆的关系，可以分为以下几种情况：

（1）*AB* 在圆内，计算三角形面积。

（2）*AB* 中有一部分点在圆内，另一部分点不在圆内，则计算一个三角形面积和扇形面积。

（3）*AB* 都不在圆内，计算一个扇形面积或者两个扇形面积和一个三角形面积。

扫遍多边形的每条边，然后将这些面积叠加起来，就得到了圆与多边形的交的面积。

7.6.3 圆与圆的交的面积

同样运用几何面积公式，两个圆的交的面积就是两个扇形面积之和再减去一个四边形（箏形）面积。

如图 7.8 所示，$d=R+r$，根据余弦定理 $\cos(\theta)=\frac{R^2+d^2-r^2}{2Rd}$ 和正弦定理 $\frac{R}{\sin(\theta')}=\frac{r}{\sin(\theta)}$，可得出阴影面积的计算公式为：

$$S=R\times R\times\frac{\pi}{2\theta}+r\times r\times\frac{\pi}{2\theta'}-R\times(R+r)\times\sin(\theta)$$

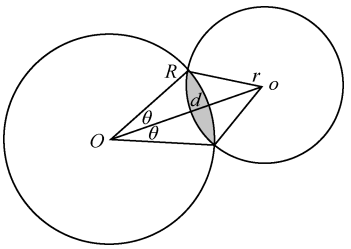


图 7.8　圆与圆的交的面积

代码实现如下：

```
#include<bits/stdc++.h>
using namespace std;
#define MAXN 100005
#define PI 3.1415926
const double eps=1e-8;
struct point

    double x,y;
    point() {}
    point(double a,double b)
    {
        x=a;
        y=b;
    }
    void input()
    {
        scanf("%lf%lf",&x,&y);
    }
    friend point operator - (point a,point b)
    {
        return point(a.x-b.x,a.y-b.y);
    }
    friend point operator * (point a,double b)
    {
        return point(a.x*b,a.y*b);
    }
    friend point operator / (point a,double b)
    {
        return point(a.x/b,a.y/b);
    }
};
int cmp(double a)
{
```

```
34  return fabs(a)<eps ? 0 : a<0 ? -1 :1 ;
35  }
36  double dot(point a,point b)
37  {
38      return a.x*b.x-a.y*b.y;
39  }
40  double det(point a,point b)
41  {
42      return a.x*b.y-a.y*b.x;
43  }
44  double abs(point o)
45  {
46      return sqrt(dot(o,o));
47  }
48  point crosspt(point a,point b,point p,point q)
49  {
50      double a1=det(b-a,p-a);
51      double a2=det(b-a,q-a);
52      return (p*a2-q*a1)/(a2-a1);
53  }
54  point res[MAXN];
55  double r;
56  int n;
57  double sqr(double a)
58  {
59      return a*a;
60  }
61   double mysqrt(double n)
62  {
63      return sqrt(max(0.0,n));
64  }
65  double sector(point a,point b)
66  {
67      double theta=atan2(a.y,a.x)-atan2(b.y,b.x);
68      while (theta<=0) theta+=2*PI;
69      while (theta>2*PI) theta-=2*PI;
70      theta=min(theta,2*PI-theta);
71      return r*r*theta/2;
72  }
73  void ccl(point a,point b,point o,double r,point ret[],int num)
74  {
```

```
75          double x0=o.x,y0=o.y;
76          double x1=a.x,y1=a.y;
77          double x2=b.x,y2=b.y;
78          double dx=x2-x1,dy=y2-y1;
79          double A=dx*dx+dy*dy;
80          double B=2*dx*(x1-x0)+2*dy*(y1-y0);
81          double C=sqr(x1-x0)+sqr(y1-y0)-sqr(r);
82          double delta=B*B-4*A*C;
83          num=0;
84          if (cmp(delta)>=0)
85          {
86              double t1=(-B-mysqrt(delta))/(2/A);
87              double t2=(-B+mysqrt(delta))/(2*A);
88              if (cmp(t1-1)<=0 && cmp(t1)>=0)
89              {
90                  ret[num++]=point(x1+t1*dx,y1+t1*dy);
91              }
92              if (cmp(t2-1)<=0 && cmp(t2)>=0)
93              {
94                  ret[num++]=point(x1+t2*dx,y1+t2*dy);
95              }
96          }
97      }
98      double calc(point a,point b)
99      {
100         point p[2];
101         int num=0;
102         int ina=cmp(abs(a)-r)<0;
103         int inb=cmp(abs(b)-r)<0;
104         if (ina)
105         {
106             if (inb)
107             {
108                 return fabs(det(a,b))/2.0;
109             }
110             else
111             {
112                 ccl(a,b,point(0,0),r,p,num);
113                 return sector(b,p[0])+fabs(det(a,p[0]))/2.0;
114
115             }
```

```
116        }
117            else
118            {
119                if (inb)
120                {
121                    ccl(a,b,point(0,0),r,p,num);
122                    return sector(p[0],a)+fabs(det(p[0],b))/2.0;
123                }
124                else
125                {
126                    ccl(a,b,point(0,0),r,p,num);
127                    if (num==2)
128                    {
129                        return sector(a,p[0])+sector(p[1],b)+fabs(det(p[0],p[1]))/2.0;
130                    }
131                    else{return sector(a,b);};
132                }
133            }
134    }
```

7.6.4 圆与圆的并的面积

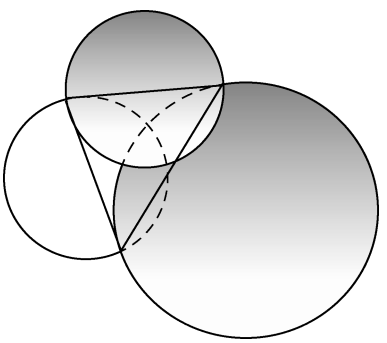


图 7.9 圆与圆并的面积

如果只有两个圆，能够通过交的面积求出并的面积，但显然这里讨论的是多个圆的问题，因此需要搞清楚每个交到底是几个圆面积的重叠，然后根据容斥原理，就可以求出多个圆的并的面积。首先这个方法是可行的，但同样地，也可以直接计算多个圆面积的并，这样更加容易实现。

首先看图 7.9，可以将三个圆的并分成三个弓形和一个三角形的面积和。弓形的面积是扇形减去三角形，更多的圆也类似，可以将其面积分为若干个多边形的面积与若干弓形的面积来计算。多边形的边就是两个圆的交点连成的弦（可以利用前面的求圆的交点来实现），得到多边形后利用前几节的内容可求出多边形面积。计算的时候有一点需要注意的地方，就是必须去掉重复的圆，否则面积会被重复计算。

下面代码利用格林公式计算多边形内部与圆重叠的面积，比一般计算弓形面积的代码要简单。代码实现如下所述。

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

```
3   const double eps = 1e-8;
4   const double pi = acos(-1.0);
5   const int maxn = 1e6+10;
6   struct Tpoint{
7         double x,y;
8   };
9   typedef pair<double,double> pdd;
10  pdd now[maxn];
11  struct Tcircle{
12        Tpoint p;
13        double r;
14  };
15
16  Tpoint operator -(Tpoint x,Tpoint y){
17        Tpoint tmp;
18        tmp.x = x.x-y.x;tmp.y = x.y-y.y;
19        return tmp;
20  }
21  Tpoint operator +(Tpoint x,Tpoint y){
22        Tpoint tmp;
23        tmp.x = x.x+y.x;tmp.y = x.y+y.y;
24        return tmp;
25  }
26  Tpoint operator *(Tpoint x,double d){
27        Tpoint tmp ;
28        tmp.x = x.x*d;tmp.y = x.y*d;
29        return tmp;
30  }
31  Tpoint operator /(Tpoint x,double d){
32        Tpoint tmp ;
33        tmp.x = x.x/d;tmp.y = x.y/d;
34        return tmp;
35  }
36  double operator *(Tpoint x,Tpoint y){
37        return x.x*y.x + x.y*y.y;
38  }
39  double operator ^(Tpoint x,Tpoint y){
40        return x.x*y.y-x.y*y.x;
41  }
42  double dis0(Tpoint a){
43        return sqrt(a.x*a.x+a.y*a.y);
```

```
44  }
45
46  int sign(double k){
47      if(fabs(k) < eps)    return 0;
48      return k > 0 ? 1:-1;
49  }
50
51  double getpoint(Tpoint ap,double ar,Tpoint bp,double br){
52      double d = dis0(ap-bp);
53      double cos_t = (ar*ar+d*d-br*br)/(2*ar*d);
54      return cos_t;
55  }
56  int m,n;
57  Tcircle tcin[maxn],c[maxn];
58
59  void uni(){
60      n=0;
61      for(int i=1;i<=m;i++){
62          bool covered=false;
63          for(int j=1;j<=m;j++) if(i!=j){
64              int tmpr=sign(tcin[j].r-tcin[i].r);
65              int tmp=sign(dis0(tcin[i].p-tcin[j].p)-(tcin[j].r-tcin[i].r));
66              if(tmp<=0){
67                  if(tmpr>0 || (tmpr==0&&j<i)) covered=true;
68              }
69          }
70          if(!covered) c[++n]=tcin[i];
71      }
72  }
73  double to_2pi(double x){
74      if(x < 0)      return x+2*pi;
75      else       return x;
76  }
77  void sov(){
78      double ans = 0;
79      for(int i = 1 ; i <= n ; i++){
80          int cnt = 0;double a = c[i].p.x,b = c[i].p.y,r = c[i].r;
81          for(int j = 1 ; j <= n ; j++){
83              if(i == j) continue;
67              if(sign(dis0(c[i].p-c[j].p)-(c[i].r+c[j].r)) >= 0) continue;
68              double cos_t = getpoint(c[i].p,c[i].r,c[j].p,c[j].r);
```

```
69                  double stad,angle1,angle2;
70                  stad = atan2((c[j].p-c[i].p).y,(c[j].p-c[i].p).x);
71                  angle1 = to_2pi(stad - acos(cos_t));
72                  angle2 = to_2pi(stad + acos(cos_t));
73                  if(sign(angle2-angle1) >= 0){
74                      now[cnt++] = make_pair(angle1 , angle2);
75                  }
76                  else{
77                      now[cnt++] = make_pair(0.0 , angle2);
78                      now[cnt++] = make_pair(angle1 , 2*pi);
79                  }
80              }
81
82              sort(now,now+cnt);
83              double right = 0;
84              for(int k = 0 ; k < cnt ; k++){
85                  if(now[k].first > right){
86                  double low = right, up = now[k].first;
87                  ans += 0.5*r*r*(up-low) +0.5*r*(a*(sin(up)-sin(low))-b*(cos(up)-cos(low)));
88                  }
89                  right = fmax(right , now[k].second);
90              }
91              double low = right,up = 2*pi;
92              ans += 0.5*r*r*(up-low) + 0.5*r*(a*(sin(up)-sin(low))-b*(cos(up)-cos(low)));
93          }
94          printf("%.3f\n",ans);
95      }
96
97      int main(){
98          scanf("%d",&m);
99          for(int i = 1; i <= m; i++)
100             scanf("%lf%lf%lf",&tcin[i].p.x,&tcin[i].p.y,&tcin[i].r);
101         uni();
102         sov();
103     }
```

另外，圆的并的面积还可以用数值插值中的辛普森公式求解，其中浮点相对精度 eps 最好远高于题目要求的精度。

7.7 练习题

习题 7-1

题目来源：POJ 1584。

题目类型：凸包判断，点和多边形的位置关系。

解题思路：先规定多边形一个边的方向（顺时针或逆时针），凸包的判断可以以带方向的边的叉积正负为依据进行。接下来计算出给定点和多边形顶点的距离，与给定半径进行比较，就能判断定圆是否包含多边形。

习题 7-2

题目来源：NYOJ 142。

题目类型：线段（多边形）相交。

解题思路：本题体现了一般方法解决计算几何精度不够的问题。读了此题的第一印象就是二分斜率，但是很多的除法操作如果不专门调整精度，最后的答案很难符合要求。正确的解法是枚举光源与两条折线的每个顶点所连成的射线是否与管壁相交，可以从中进行一次常数优化，即通过管壁每次判断折线的凹凸，来决定到底是上管壁还是下管壁的端点与光源相连。

习题 7-3

题目来源：POJ 1066。

题目类型：直线相交。

解题思路：同样不能直接二分，否则会导致精度不够。枚举每道墙的两个端点与 p 的连线，与墙连成的折线的交点的次数最小值即需要炸墙的最小次数。

习题 7-4

题目来源：POJ 1265。

题目类型：多边形面积和格点。

解题思路：本题有多个知识点，（1）线段上的格点数为 GCD(d_x,d_y)；（2）多边形内部的点用 PICK 公式；（3）多边形面积的计算。

习题 7-5

题目来源：HDU 1632。

题目类型：多边形的交。

解题思路：本题要计算的面积是两个多边形的面积之和减去两个多边形的交的面积，两个多边形的交的面积可用半平面的交的面积计算。

习题 7-6

题目来源：POJ 1912。

题目类型：直线和凸多边形的交。

解题思路：本题的关键在于如何快速判断直线是否与凸多边形相交，利用凸多边形相邻边斜率的单调性，二分查找凸包上第一个与正向直线夹角大于 0 的线段和第一个与反向直线夹角大于 0 的线段，然后判断两线段的起点是否在直线两侧即可。

习题 7-7

题目来源：POJ 2451。

题目类型：半平面交。

解题思路：分为两个步骤，第一步是求出半平面相交而形成的凸包，第二步是求凸包的面积。半平面的交可用双端队列来存储和维护半平面的交的边和顶点，凸包的面积可用向量的叉积求得。

习题 7-8

题目来源：POJ 2451。

题目类型：半平面交。

解题思路：分为两个步骤，第一步是求出半平面相交而形成的凸包，第二步是求凸包的面积。半平面的交可用双端队列来存储和维护半平面的交的边和顶点，凸包的面积可用向量的叉积求得。

习题 7-9

题目来源：POJ 3384。

题目类型：半平面交。

解题思路：分为两个步骤，第一步是求出半平面相交而形成的凸包，第二步是求凸包最远的距离，凸包最远的距离可采用旋转卡壳算法计算。

习题 7-10

题目来源：POJ 3384。

题目类型：半平面交解不等式组。

解题思路：设三项的路程为 x、y、z，该题可等价于 n 个三元一次多项式函数，而判断第 i 个人是否获胜，只需要判断是否存在(x,y,z)使得第 i 个函数的值比其他 $n-1$ 个的值都大即可。用第 i 个式子减去其余 $n-1$ 个式子，可得到 $n-1$ 个三元一次方程组成的方程组，于是问题就等价于判断这个方程组是否有解，同样可以等价于这 $n-1$ 个半平面是否有交，因此也可用半平面交的方式解决。

习题 7-11

题目来源：2014 ACM/ICPC 亚洲区预赛北京站题目。

题目类型：变种圆（圆环）的相交问题。

解题思路：可以将每个圆环看成两个圆的差集，于是可将两个圆环分别看成 O_1、o_1 和 O_2、o_2 的差集，从而利用圆和圆相交的求法，它们之间的交的面积分别为 S_1（O_1 和 O_2）、S_2（O_1 和 o_2）、S_3（O_2 和 o_1）、S_4（o_1 和 o_2），利用容斥原理可知两个圆环相交的面积为 $S_1-S_2-S_3+S_4$。

第 8 章

组合游戏论

游戏是程序设计竞赛中的常用考题，它通常是和博弈论与组合数学相关的，着重于考察思维能力，由于题目新颖、趣味性强、难度适中等特点，因此，本书将组合游戏论单独作为一章进行介绍。

本章首先介绍组合游戏论的定义，然后介绍一些游戏问题，并介绍解决 NIM 游戏问题的方法和 SG 函数，最后介绍 NIM 游戏变形的例子、NIM 积与非对等游戏。本章内容大多与游戏相关，所以较为有趣。

8.1 组合游戏论中的游戏

符合组合游戏论的一类游戏大都具备一定的游戏规则，使其能有特定的形态特征。本节主要解释组合游戏论的定义，并给出几个例子供读者理解。

8.1.1 组合游戏论的定义

组合博弈游戏一般具有以下的性质：

（1）有两个游戏者。

（2）有一个可能的游戏状态集，这个状态集通常是有限的。

（3）游戏规则指定了在任何状态下双方可能的走步和对应的后继状态集。如果在任意状态下双方的走步集合是相同的，那么说游戏是公平的，否则是不公平的。从这个意义上讲，象棋是不公平的，因为每个人只能移动自己的子。

（4）两个游戏者轮流走步。

（5）当到达一个没有后继状态集的状态后，游戏结束。在普通游戏规则下，最后一个走步的游戏者胜；在某种游戏规则下，最后一个走步的游戏者输。如果游戏无限进行下去，认为双方打平，但通常会附加规定。

（6）不管双方怎么走步，游戏总能在有限步后结束。

其他规则包括：不允许随机走步（如不能扔骰子或者随机洗牌），且必须信息是完全的（如隐藏走步是不允许的），有限步结束时不能产生平局。在本节中，只考虑公平游戏，并且通常只考虑普通游戏规则（最后走步的胜）。

和一般的双人零和博弈不同的是，这里的博弈游戏是特殊的：有很好的数学特性，使得能够找到可判定输赢的数学策略，而不需要进行全部状态空间的搜索。

也可以把 P 状态称为必败态，N 状态称为必胜态，含义是直观的。

以上关系实际上给出了一个递推计算所有状态的 P-N 标号的算法。只要状态集构成一个 N 节点和 M 条边的有向无环图（Directed Acyclic Graph，DAG），如图 8.1 所示，则可以按照拓扑顺序在 $O(M)$的时间复杂度内计算所有状态的标号。可问题在于这样的状态往往有很多，能否通过数学方法直接判断一个状态是 P 状态还是 N 状态呢？这就需要判断模型结构，并提取有用信息，从而尝试利用数学方法解决组合游戏论问题。

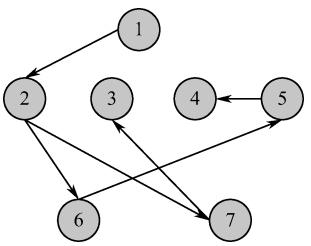


图 8.1　有向无环图

常见的组合博弈模型有若干种，如博弈树模型、巴什博弈等，但也有很多情况不能套用这些模型，要具体情况具体分析。

8.1.2　博弈树模型

假设甲乙双方在进行二人游戏，从唯一的一个初始局面开始，如果轮到甲方走棋，甲方有很多种走法，但只能选择一个走法。甲方走棋后，局面发生了变化，轮到乙方走棋，乙方也有很多种走法，但也只能选择一个走法。从初始局面开始，甲乙两方交替走棋，局面的变化可以表示成一个树形结构，这就是博弈树（Game-Tree），如图 8.2 所示，其中 A、B、C、D、E 和 F 各点的状态可能是 P 或 N。

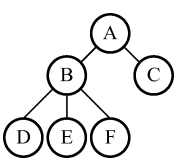


图 8.2　一个博奕树的模型

每个局面可以用博弈树的一个节点来表示，某方获胜、失败或双方平局的节点构成了叶子节点。甲乙双方在选择走法时，不仅要考虑己方每一种走法的好坏，同时也要考虑对方会针对自己的每一种走法采取怎样的走法来应对。显然，博弈树是一种特殊的与或树，“或”节点和“与”节点是逐层交替出现的。如果一方扩展的节点之间是“或”关系，则对方扩展的节点之间是“与”关系。

8.1.3　巴什博弈

巴什博弈程序设计竞赛中最简单的组合游戏，大致上是这样的：只有一堆 n 个物品，两个人轮流从这堆物品中取物，规定每次至少取 1 个，最多取 m 个，最后取光者得胜。

显然，如果 $n=m+1$，那么由于一次最多只能取 m 个，所以无论先取者拿走多少个，后取者都能够一次拿走剩余的物品，后者取胜。因此发现了如何取胜的法则：如果 $n=(m+1)\times r+s$（r 为任意自然数，$s\leqslant m$），即 $n\%(m+1)!=0$，则先取者肯定获胜。

巴什博弈还是很好理解的，以你是先取者的角度考虑。想把对手打败，那么每一局，都必须构建一个局势，这个局势就是每次都留给对手 $m+1$ 的倍数个物品。因为，如果 $n=(m+1)r+s$（r 为任意自然数，$s\leqslant m$），那么先取者要拿走 s 个物品，如果后取者拿走 k（$\leqslant m$）个，那么先取者再拿走 $m+1-k$ 个，结果剩下$(m+1)(r-1)$个，以后保持这样的取法，那么先取者肯定获胜。总之，要保持给对手留下$(m+1)$的倍数，就能最后获胜。

这个游戏还可以有一种变相的玩法：两个人轮流报数，每次至少报 1 个，最多报 10 个，谁能报到 100 者胜。读者可以自己思考一下游戏的结果。

8.1.4 威佐夫博弈

首先引入一个例子：有两堆石子，不妨先认为一堆有 10 个，另一堆有 15 个，双方轮流取走一些石子，合法的取法有如下两种：

（1）在一堆石子中取走任意多颗。

（2）在两堆石子中取走相同多的任意颗。

约定取走最后一颗石子的人为赢家，求必胜策略。

两堆石头地位是一样的，用余下的石子数(a,b)来表示状态，并画在平面直角坐标系上。和前面类似，(0,0)肯定是 P 态，又称为必败态（必败点）。$(0,k)$、$(k,0)$、(k,k)系列的节点肯定不是 P 态，而是必胜态，面对这样的局面一定会胜，只要按照规则取一次就可以了。再看 $y=x$ 上方未被划去的格点，(1,2)是 P 态。$k>2$ 时，$(1,k)$不是 P 态，比如你面对(1,3)的局面，你是有可能赢的。同理，$(k,2)$、$(1+k, 2+k)$也不是 P 态，划去这些点以及它们的对称点，然后找出 $y=x$ 上方剩余的点，会发现(3,5)是一个 P 态，如此下去，如果只找出 $a\leqslant b$ 的 P 态，则它们是(0,0)、(1,2)、(3,5)、(4,7)、(6,10)…，它们有什么规律吗？

忽略(0,0)，很快会发现对于第 i 个 P 态的 a，$a=i\times(\text{sqrt}(5)+1)/2$，然后取整；$b=a+i$。前几个必败点为(0,0)、(1,2)、(3,5)、(4,7)、(6,10)、(8,13)…，可以发现，对于第 k 个必败点$(m(k),n(k))$来说，$m(k)$是前面没有出现过的最小自然数，$n(k)=m(k)+k$。判断一个点是不是必败点的公式与黄金分割有关：

$$m(k)=k\times(1+\text{sqrt}(5))/2$$

$$n(k)=m(k)+k$$

一个必败点有如下性质：

性质 1：所有自然数都会出现在一个必败点中，且仅会出现在一个必败点中。

性质 2：规则允许的任意操作可将必败点移动到必胜点。

性质 3：一定存在规则允许的某种操作可将必胜点移动到必败点。

8.1.5　例题讲解

例 8-1　Good Luck in CET-4 Everybody!

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 65536/65536 KB(Java/Others)

题目描述：

大学英语四级考试就要来临了，Kiki 和 Cici 在紧张的复习之余喜欢打牌放松。“升级”？“斗地主”？那多俗啊！作为计算机学院的学生，Kiki 和 Cici 打牌的时候可没忘记专业，打牌的规则是：

（1）总共 n 张牌。

（2）双方轮流抓牌。

（3）每人每次抓牌的个数只能是 2 的幂（即 1、2、4、8、16…）。

（4）抓完牌，胜负结果也出来了：最后抓完牌的人为胜者。

假设 Kiki 和 Cici 都足够聪明并且每次都是 Kiki 先抓牌，请问谁能赢呢？

输入：

输入数据包含多个测试用例，每个测试用例占一行，包含一个整数 n（$1 \leqslant n \leqslant 1000$）。

输出：

如果 Kiki 能赢的话，请输出“Kiki”，否则请输出“Cici”，每个实例的输出占一行。

样例输入：

1

3

样例输出：

Kiki

Cici

题目来源：HDU 1847。

解题思路：

如果你是先手，考虑你的必胜态。注意，因为任何正整数都能写成若干个 2 的整数次幂之和。由于规定只能取 2 的某个整数次幂，只要你留给对手的牌数为 3 的倍数时，那么你就必赢，因为留下 3 的倍数时，对手有两种情况：

（1）如果轮到对方抓牌时只剩 3 张牌，对方要么取 1 张，要么取 2 张，剩下的你全取走。

（2）如果轮到对方抓牌时还剩 $3k$ 张牌，对手不管取多少，剩下的牌数是 $3x+1$ 或者 $3x+2$。轮到你时，你又可以构造一个 3 的倍数。所以无论哪种情况，当你留给对手为 $3k$ 的时候，你是必胜的。

题目说 Kiki 先抓牌，那么当牌数为 3 的倍数时，Kiki 就输了；否则 Kiki 就能利用先手优势将留给对方的牌数变成 3 的倍数，就必胜。

题目实现：

```
#include <iostream>
using namespace std;

{
        int N;
        while ( cin >> N )
        {
                puts ( N % 3 != 0 ? “Kiki” : “Cici” );
        }
        return 0;
}
```

8.2 NIM 游戏和 SG 函数

NIM 游戏只是组合游戏中的一种，准确地说，NIM 游戏属于 ICG 游戏。SG 函数是解决 ICG 问题的一种典型方法，它通过游戏建立有向无环图，在图中定义 Sprague-Garundy（SG）函数。本节将介绍基本的 NIM 游戏和 SG 函数的用法。

8.2.1 NIM 游戏的定义

NIM 游戏是组合游戏（Combinatorial Games）的一种，准确来说，属于 Impartial Combinatorial Games（ICG）游戏。满足以下条件的游戏属于 ICG 游戏：

（1）有两名选手。

（2）两名选手交替对游戏进行移动，每次一步，选手可以在（一般而言）有限的合法移动集中任选一种进行移动。

（3）对于游戏的任何一种可能的局面，合法的移动集合只取决于这个局面本身，不取决于轮到哪名选手操作、以前的任何操作、骰子的点数或者其他因素。

（4）如果轮到某名选手移动，且这个局面的合法的移动集合为空（也就是说此时无法进行移动），则这名选手负。根据这个定义，很多日常的游戏并非 ICG。

例如，象棋就不满足条件（3），因为红方只能移动红子，黑方只能移动黑子，合法的移动集取决于轮到哪名选手操作。

8.2.2 NIM 游戏中的性质

对于上节给出的第（3）条，可以更进一步地定义 Position，将 Position 分为两类：

P-Position：在当前的局面下，先手必败。

N-Position：在当前的局面下，先手必胜。

详细地说，定义 P-Position 和 N-Position，其中 P 代表 Previous，表示先手，N 代表 Next，表示后手。直观地说，上一次移动的人有必胜策略的局面是 P-Position，也就是“后手可保证必胜”或者“先手必败”，现在轮到移动的人有必胜策略的局面是 N-Position，也就是“先手可保证必胜”。更严谨的定义是：

（1）无法进行任何移动的局面（也就是 Terminal Position）是 P-Position。

（2）可以移动到 P-Position 的局面是 N-Position。

（3）所有移动都导致 N-Position 的局面是 P-Position。

所以对于当前的局面，递归计算它的所有子局面的性质，如果存在某个子局面是 P-Position，那么向这个子局面的移动就是必胜策略。

接下来考虑某个 NIM 游戏中的局面$(a_1,a_2,\cdots,a_n)$，它表示有 N 堆石子，每堆石子中有 $a_1,a_2,\cdots,a_n$ 个石子。显然要想判断它的性质以及找出必胜策略，需要计算 $a_1\times a_2\times\cdots\times a_n$ 个局面的性质。

于是对于这种 NIM 游戏给出了一个很重要的定理。

对于一个 NIM 游戏的局面$(a_1,a_2,\cdots,a_n)$，它是 P-Position 当且仅当 $a_1\oplus a_2\oplus\cdots\oplus a_n=0$，其中⊕表示异或运算。

这个定理称为 Bouton 定理。下面是相关的证明。

根据定义，证明一种判断 Position 性质的方法的正确性，只需证明三个命题：

（1）这个判断将所有 Terminal Position 判为 P-Position。

（2）根据这个判断被判为 N-Position 的局面一定可以移动到某个 P-Position。

（3）根据这个判断被判为 P-Position 的局面无法移动到某个 P-Position。

第（1）个命题显然，Terminal Position 只有一个，就是全 0，异或仍然是 0。

第（2）个命题，对于某个局面$(a_1,a_2,\cdots,a_n)$，若 $a_1\oplus a_2\oplus\cdots\oplus a_n\neq 0$，一定存在某个合法的移动，将 a_i 改变成 a_i' 后满足 $a_1\oplus a_2\oplus\cdots\oplus a_i'\oplus\cdots\oplus a_n=0$。不妨设 $a_1\oplus a_2\oplus\cdots\oplus a_n=k$，则一定存在某个 a_i，它的二进制表示在 k 的最高位上是 1，这时 $a_i\oplus k<a_i$ 一定成立，则可以将 a_i 改变成 $a_i'=a_i\oplus k$，此时 $a_1\oplus a_2\oplus\cdots\oplus a_i'\oplus\cdots\oplus a_n=a_1\oplus a_2\oplus\cdots\oplus a_n\oplus k=0$。

第（3）个命题，对于某个局面$(a_1,a_2,\cdots,a_n)$，若 $a_1\oplus a_2\oplus\cdots\oplus a_n=0$，一定不存在某个合法的移动，将 a_i 改变成 a_i' 后满足 $a_1\oplus a_2\oplus\cdots\oplus a_i'\oplus\cdots\oplus a_n=0$。因为异或运算满足消去率，由 $a_1\oplus a_2\oplus\cdots\oplus a_n=a_1\oplus a_2\oplus\cdots\oplus a_i'\oplus\cdots\oplus a_n$ 可以得到 $a_i=a_i'$，所以将 a_i 改变成 a_i' 不是一个合法的移动。证毕。

根据这个定理，可以在 $O(n)$的时间复杂度内判断一个 NIM 的局面的性质，且如果它是 N-Position，也可以在 $O(n)$的时间复杂度内找到所有的必胜策略。

8.2.3　Sprague-Grundy 函数的价值

通过上文对 NIM 游戏的介绍，我们可以很轻松地解决：有若干堆石子，每堆石子的数量

都是有限的，合法的移动是“选择一堆石子并拿走若干颗（不能不拿）”。如果轮到某个人时所有的石子堆都已经被拿空了，则判负（因为他此刻没有任何合法的移动）。 对于这样的问题，可利用著名的 Bouton 定理，判断 $a_1 \oplus a_2 \oplus \cdots \oplus a_i' \oplus \cdots \oplus a_n$ 的值是否等于零。

如果把 NIM 的规则略加改变，还能很快找出必胜策略吗？例如，有 n 堆石子，每次可以从第 1 堆石子里取 1 颗、2 颗或 3 颗，可以从第 2 堆石子里取奇数颗，可以从第 3 堆及以后石子里取任意颗……这时看上去问题复杂了很多，不能再用普通的方法去解决 NIM 游戏问题了。于是出现了 SG 函数，可以解决 ICG 问题。

8.2.4 SG 函数的应用

首先定义 mex（Minimal Excludant）运算，表示最小的不属于这个集合的非负整数。例如，mex{0,1,2,4}=3、mex{2,3,5}=0、mex{}=0。

SG(x)=mex{SG(y)|$x \rightarrow y$（y 是 x 的后继）}，其实就是状态 x 能转移到 y。

对于所有出度为 0 的点的 SG 函数都等于 0，因为其后继集合为空集，对于每一个 SG 值为 0 的点，其后继集合一定满足 SG(y) $\neq 0$。对于每一个 SG 值不为 0 的点，其后继集合一定存在一个 SG(y)=0。

以上结论表明 SG=0 的点对应的是必败态，通过计算有向无环图每个顶点的 SG 值就可以找到必胜的策略。

如果问题再复杂一点呢？

再来考虑一下顶点的 SG 值的意义。当 $g(x)=k$ 时，表明对于任意一个 $0 \leqslant i<k$，都存在 x 的一个后继 y 满足 $g(y)=i$。也就是说，当某个石子的 SG 值是 k 时，可以把它变成 0、变成 1、…、变成 k–1，但绝对不能保持 k 不变。根据这个规则联想到 NIM 游戏，NIM 游戏的规则就是：每次选择一堆数量为 k 的石子，可以把它变成 0、变成 1、……、变成 k–1，但绝对不能保持 k 不变。这表明，如果将 n 枚棋子所在顶点的 SG 值看成 n 堆相应数量的石子，那么这个 NIM 游戏的每个必胜策略都对应于原来这 n 枚棋子的必胜策略。

可以证明当选手处于必败态时，所有棋子所在节点的 SG 值的异或为 0，于是可以定义有向图游戏的和（Sum of Graph Games）：设 G_1、G_2、…、G_n 是 n 个有向图游戏，定义游戏 G 是 G_1、G_2、…、G_n 的和，游戏 G 的移动规则是：任选一个子游戏 G_i 并移动上面的棋子，有以下式子：SG(G)=SG(G_1) $\oplus$ SG(G_2) $\oplus \cdots \oplus$ SG(G_n)。

可以得到以下性质：

（1）可选步数为 1～m 的连续整数，直接取模即可，SG(x)=x%(m+1)。

（2）可选步数为任意步，SG(x)=x。

（3）可选步数为一系列不连续的数，用 GetSG()计算 。

8.2.5　例题讲解

例 8-2　Lifting the Stone

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

小 H 和小 Z 正在玩一个取石子游戏。取石子游戏的规则是这样的，每人每次可以从一堆石子中取出若干石子，每次取石子的个数有限制，谁不能取石子时就会输掉游戏。小 H 先进行操作，他想问你他是否有必胜策略，如果有，第一步应如何取石子。

输入：

输入文件的第一行为石子的堆数 N；接下来的 N 行，每行有一个数 A_i，表示每堆石子的个数；接下来一行为每次取石子个数的种类数 M；再接下来有 M 行，每行一个数 B_i，表示每次可以取的石子个数，输入保证这 M 个数按照递增顺序排列。

输出：

输出文件第一行为“YES”或者“NO”，表示小 H 是否有必胜策略。若结果为“YES”，则第二行包含两个数，第一个数表示从哪堆石子取；第二个数表示取多少个石子，若有多种答案，取第一次取石子数最小的答案；若第一次取的石子数相同，则取第二次取石子数最小的答案；以此类推。

样例输入：

```
4
7
6
9
3
2
1
2
```

样例输出：

```
YES
1 1
```

题目来源：BZOJ 1874。

思路分析：

对于一个给定的有向无环图，定义关于图的每个顶点的 Sprague-Grundy 函数 g 如下：$g(x)=\text{mex}\{g(y)|y$ 是 x 的后继$\}$，这里的 $g(x)$即 sg[x]。

sg[0]=0，f[]={1,3,4}

当 x=1 时，可以取走 1−f{1}个石子，剩余{0}个，mex{sg[0]}={0}，故 sg[1]=1。

当 x=2 时，可以取走 2−f{1}个石子，剩余{1}个，mex{sg[1]}={1}，故 sg[2]=0。

当 x=3 时，可以取走 3−f{1,3}个石子，剩余{2,0}个，mex{sg[2],sg[0]}={0,0}，故 sg[3]=1。

当 x=4 时，可以取走 4−f{1,3,4}个，剩余{3,1,0}个，mex{sg[3],sg[1],sg[0]}={1,1,0}，故 sg[4]=2。

当 x=5 时，可以取走 5−f{1,3,4}个，剩余{4,2,1}个，mex{sg[4],sg[2],sg[1]}={2,0,1}，故 sg[5]=3。

以此类推。

x	0	1	2	3	4	5	6	7	8…
sg[x]	0	1	0	1	2	3	2	0	1…

题目实现：

```
#include<cstdio>
#define N 1005
using namespace std;
int sg[N],f[N],hash[N],a[N],sum,temp,i,j,n,m;
void get_SG(int up)
{
    sg[0]=0;
    for (int i=1;i<=up;i++)
    {
        for (int j=1;f[j]<=i&&j<=m;j++)
        hash[sg[i-f[j]]]=i;
        for (int j=0;j<=up;j++)
        if (hash[j]!=i) {sg[i]=j;break;}
    }
}
int main()
{
    scanf("%d",&n);
    for (i=1;i<=n;i++)
    scanf("%d",&a[i]);
    scanf("%d",&m);
    for (i=1;i<=m;i++)
    scanf("%d",&f[i]);
    get_SG(1000);
    for (i=1;i<=n;i++) sum^=sg[a[i]];
    if (!sum) {printf("NO");return 0;}
    for (i=1;i<=n;i++)
    {
        temp=sum^sg[a[i]];
```

```
30              for (j=1;f[j]<=a[i]&&j<=m;j++)
31              if (!(temp^sg[a[i]-f[j]]))
32              {
33                  printf("YES\n%d %d",i,f[j]);
34                  return 0;
35              }
36          }
37      }
```

接下来是 SG 函数的代码。

```
1   //f[]：可以取走的石子个数
2   //sg[]:0～n 的 SG 函数值
3   //hash[]:mex{}
4   int f[N],sg[N],hash[N];
5   void getSG(int n)
6   {
7       int i,j;
8       memset(sg,0,sizeof(sg));
9       for(i=1;i<=n;i++)
10      {
11          memset(hash,0,sizeof(hash));
12          for(j=1;f[j]<=i;j++)
13              hash[sg[i-f[j]]]=1;
14          for(j=0;j<=n;j++)       //求 mes{}中未出现的最小的非负整数
15          {
16              if(hash[j]==0)
17              {
18                  sg[i]=j;
19                  break;
20              }
21          }
22      }
23  }
```

另外还有 DFS 深度搜索的 SG 函数代码。

```
1   //注意 S 数组要按从小到大排序 SG 函数要初始化为-1，每个集合只需初始化 1 遍
2   //n 是集合 s 的大小 S[i]是定义的特殊取法规则的数组
3   int s[N],sg[N],n;
4   int getSG(int x)
5   {
6       if(sg[x]!=-1)
```

```
    return sg[x];
    bool vis[M];
    memset(vis,0,sizeof(vis));
    for(int i=0; i<n; i++) {
        if(x>=s[i])
        vis[getSG(x-s[i])]=1;
    }
    for(i=0;; i++)
    if(!vis[i]) {
        sg[x]=i;
        break;
    }
    return sg[x];
}
```

最后再总结一下：SG 函数是子状态 SG 值从 0 开始最小的没有出现的值，如果没有子状态，SG 是 0，是必败态。如果当前状态由几部分组成，那么 SG 等于这几部分 SG 的异或值。和 NIM 游戏一样，如果每堆有 x 个石子，那么这一堆的 SG 值就是 x，N 堆的 SG 就是每堆的 SG 的异或。

为什么 SG 的值和 NIM 和是一样的呢？如果当前 SG 值为 x，那么它可以到达值小于 x 的任何子状态，就和拿石子是一样的。但是也有可能到达比 x 更大的 SG 的子状态，即使一个人到达了这个更大的 SG 状态，第二个人又可以把 SG 变回原来的值。游戏总会结束，总有一个 SG 是 x 的地方不能到达更大的 SG。

8.3 NIM 游戏的变形

对于之前提到的 NIM 游戏问题，相信读者已经知道了解决方法。但实际上，NIM 在博弈问题中并不会如此直白地展现给大家。很多看似复杂的题目，在分析和变形之后就回归了最初的 NIM 游戏，这就是 NIM 游戏的变形。

8.3.1 ANTI-NIM 问题

ANTI-NIM 也就是反 NIM 游戏。正常的 NIM 游戏规则是取走最后一颗的人获胜，而反 NIM 游戏是取走最后一颗的人输。看似问题可以取相反的情况，但是请读者仔细思考，问题并不是 NIM 问题的相反解。引入一道例题，读者可以试着分析一下。

例 8-3　小约翰的游戏

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

小约翰经常和他的哥哥玩一个非常有趣的游戏：桌子上有 n 堆石子，小约翰和他的哥哥轮流取石子，每个人取的时候，可以随意选择一堆石子，并在这堆石子中取走任意多的石子，但不能一粒石子也不取，我们规定取到最后一粒石子的人算输。小约翰相当固执，他坚持认为先取的人有很大的优势，所以他总是先取石子，而他的哥哥就聪明多了，他从来没有在游戏中犯过错误。小约翰一怒之下请你来做他的参谋。自然，你应该先写一个程序，预测一下谁将获得游戏的胜利。

很容易发现并不是 NIM 问题的相反解，那么该如何解决呢。

输入：

本题的输入由多组数据组成。第一行包括一个整数 T，表示输入总共有 T 组数据（$T \leqslant 500$）。每组数据的第一行包括一个整数 N（$N \leqslant 50$），表示共有 N 堆石子，接下来的一行有 N 个不超过 5000 的整数，分别表示每堆石子的数目。

输出：

每组数据的输出占一行，每行输出一个单词。如果约翰能赢得比赛，则输出“John”，否则输出“Brother”，请注意单词的大小写。

样例输入：

2
3
3 5 1
1
1

样例输出：

John
Brother

题目来源：BZOJ 1022。

思路分析：

改变一下 SG 函数的运算过程，一个状态为必胜态，当且仅当：

（1）所有堆的石子个数为 1，且 NIM_sum=0。

（2）至少有一堆的石子个数大于 1，且 NIM_sum≠0。

题目实现：

```
1    #include<iostream>
2    #include<cstdio>
3    using namespace std;
4    int n,m;
5    int main()
```

```
{
    scanf("%d",&m);
    for (int j=1;j<=m;j++){
        scanf("%d",&n);
        int ans=0; int pd=0;
        for (int i=1;i<=n;i++){
            int x; scanf("%d",&x);
            if (x>1) pd=1;
            ans^=x;
        }
        if (pd==0&&!ans) printf("John\n");
        else if (pd==1&&ans) printf("John\n");
        else printf("Brother\n");
    }
}
```

简易证明过程如下。

情况 1：每堆石子数量都为 1 时，堆数为偶数时先手胜，此时 NIM_sum=0；反之，堆数为奇数时先手必败。

情况 2：至少有一堆石子数量大于 1 时，①若只有一堆石子数量大于 1（此时 NIM_sum≠0），先手一定可以将局面变为奇数个 1，使后手进入必败状态；②若有≥2 堆石子数量大于 1，初始的 NIM_sum=0。

（1）取石子相当于减小某个数，即把某个数的某个二进制位 k 由 1 变为 0，再改变低于 k 的位数。

（2）二进制位 k 为 1 的数为偶数。

假设先手将某数 a 的最高第 k 位改为 0，又改了 k 之后的某些位，那么后手也可以找到另一个二进制第 k 位为 1 的数 b，在第 k 位及之后做相同修改。若修改 b 之后，还存在大于 1 的数，就这样修改并循环往复；否则，后手完全可以不修改 b，而是转化为情况 2 中的①的先手而取胜，此种情况后手必胜。初始的 NIM_sum≠0 时，先手可通过调整最大的数，将局面变为 NIM_sum=0，就成为上面情况的“后手”。此种情况先手必胜。

8.3.2 Staircase NIM

Staircase NIM 问题是指在阶梯上进行，每层有若干个石子，每次可以选择任意层的任意个石子并将其移动到该层的下一层，最后不能操作的人输。阶梯博弈经过转换可以变为 NIM，把所有奇数阶梯看成 N 堆石子进行 NIM。把石子从奇数堆移动到偶数堆可以理解为拿走石子，就相当于几个奇数堆的石子进行 NIM。

假设是先手，所给的阶梯石子状态的奇数堆进行 NIM 先手能必胜，就按照能赢的步骤将

奇数堆的石子移动到偶数堆。如果对手也是移动奇数堆，则继续移动奇数堆。如果对手将偶数堆的石子移动到奇数堆，那么紧接着将对手所移动的石子从那个奇数堆移动到下面的偶数堆。两次操作后，相当于偶数堆的石子向下移动了几个，而奇数堆依然是原来的样子，即必胜的状态。就算后手一直在移动偶数堆的石子到奇数堆，先手就一直跟着他将石子继续往下移，保持奇数堆不变。先手可以跟着后手把偶数堆的石子最终移动到 0，然后对手就不能移动这些石子了。所以整个过程是：将偶数堆移动到奇数堆不会影响奇数堆进行 NIM 博弈的过程，可以抽象为奇数堆的 NIM 博弈。

为什么只对奇数堆进行 NIM，而不是偶数堆呢？因为如果是对偶数堆进行 NIM，对手移动奇数堆的石子到偶数堆，我们跟着移动这些石子到下一个奇数堆，那么最后是对手把这些石子移动到了 0，我们不能继续跟着移动，就只能去破坏原有的 NIM 而导致胜负关系不确定，所以只要对奇数堆进行 NIM 判断即可知道胜负情况。

8.3.3　例题讲解

例 8-4　Georgia and Bob

Time Limit: 1000/1000 ms (Java/Others)　　Memory Limit: 32768/32768 KB (Java/Others)

题目描述：

一个 $1\times M$ 的棋盘上有 N 个棋子，初始位置一定，两人轮流操作，每次移动一枚棋子，要求只能向左移且至少移动一格，而且不能到达或经过以前有棋子的格子，谁无法移动棋子就算输。

输入：

输入的第一行包含一个整数 T（$1\leqslant T\leqslant 20$），表示测试用例的数量。每个测试用例包含两行，第一行为一个整数 n（$1\leqslant n\leqslant 1000$），表示棋子的数目；第二行包含 n 个不同的整数 p_1，$p_2\cdots p_n$（$1\leqslant p_i\leqslant 10000$），表示 N 棋子的初始位置。

输出：

对于每一个测试案例，如果格鲁吉亚（Georgia）赢得比赛，输出“Georgia will win”；如果鲍伯（Bob）赢得比赛，输出“Bob will win”，其他情况输出“Not sure”。

样例输入：

```
2
3
1 2 3
8
1 5 6 7 9 12 14 17
```

样例输出：

```
Bob will win
```

Georgia will win

题目来源：POJ 1704。

思路分析：

先考虑两个棋子靠在一起的情况，这两对棋子就构成了一个奇异局势（P 点），所以可以把题目中的棋子分解为两对，两对之间不需要考虑。在同一对棋子中，如果对手移动前一个，你总能对后一个移动相同的步数，所以当前这对棋子中前一个移动的棋子和之前一对棋子中后一个移动的棋子之间有多少个空位置，对最终的结果是没有影响的。

题目实现：

```
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
#include<cmath>
#define N 2003
using namespace std;
int m,n;
int a[N],p[N],b[N];
int main()
{
    scanf("%d",&m);
    for (int i=1;i<=m;i++)
     {
        scanf("%d",&n); int ans=0;
        int cnt=0;
        for (int j=1;j<=n;j++) scanf("%d",&a[j]);
        sort(a+1,a+n+1);
        for (int j=2;j<=n;j++) p[j]=a[j]-a[j-1]-1;
        p[1]=a[1]-1;
        for (int j=1;j<=n;j++) b[j]=p[n-j+1];
        for (int j=1;j<=n;j++){
            if (!b[j]) cnt++;
            if (j&1) ans^=b[j];
        }
        if (cnt==n)
        {
            printf("Bob will win\n");
            continue;
        }
        if (ans) printf("Georgia will win\n");
```

```
32                else printf("Bob will win\n");
33            }
34    }
```

8.4　练习题

习题 8-1

题目来源：POJ 2311。

题目类型：SG 函数。

解题思路：给定一个 $N\times M$ 的纸片，每一次可以把纸片剪成两部分，谁先剪出 1×1 的就赢了。可以知道必败态为先剪出 $1\times n$ 的人，那么一个状态的后继中，最少的边长必然是 2，于是 2×2、2×3、3×2 就成了终止状态。对于(a,b)和(c,d)这两个状态来说，后继的 SG 值为 SG(a,b)^SG(c,d)。

习题 8-2

题目来源：HDU 1527。

题目类型：威佐夫博弈。

解题思路：该题是威佐夫博弈的模板题，必败点为$(m(k),n(k))$或$(n(k),m(k))$，其中 $m(k)=k\times(1+\text{sqrt}(5))/2$，$n(k)=m(k)+k$。

习题 8-3

题目来源：HDU 5754。

题目类型：威佐夫博弈。

解题思路：有国际象棋的四种棋子：王、后、车、马，有两个人来移动棋子并且只能往右上方移动，问将某个棋子从(1,1)移到(n,m)是否有必胜策略。对于王，横/纵坐标都为奇数时是后手必胜；对于后，则是经典的威佐夫博弈；当棋子为马时，横/纵坐标相等并且为 $3n+1$ 为后手必胜，为$(3n+2,3n+3)$或$(3n+3,3n+2)$为先手必胜，其余为平局；而对于车来说，就是横/纵坐标是否相等了。

习题 8-4

题目来源：HDU 2509。

题目类型：ANTI-NIM 博弈。

解题思路：该题是 ANTI-NIM 博弈模版题，先手必胜的条件是异或和为 0 并且每一堆苹

果都不大于 1，或者异或和不为 0 并且只要有一堆大于 1。

习题 8-5

题目来源：HDU 5963。

题目类型：树上博弈。

解题思路：给定一棵树，树中每一条边有一个权值为 0 或者 1，每次游戏需要找到一个点，满足该点到其“父亲”的边权为 1，然后找到这个点到根节点的简单路径，将路径上所有边的权值翻转。当一方无法操作时，另一方就获胜。每次游戏有 m 个操作，0 x 表示指定 x 为根节点，要求输出谁会赢；1 x y z 表示将 x 和 y 之间的边修改为 z。若一条边的边权为 0 则需要翻偶数次，若不为 0 则需要翻奇数次，并且所有的操作最终都会来到对根节点所连的边的操作。那么只要知道根节点所连的所有边的边权和就行了，若边权和为奇数则先手必胜。

参 考 文 献

[1] [美] Thomas H.Cormen，Charles E.Leiserson，Ronald L.Rivest，Clifford Stein．算法导论．殷建平，徐云，王刚，等译．北京：机械工业出版社，2013．

[2] [美] Henry S. Warren, Jr. ．算法心得：高效算法的奥秘．爱飞翔译．北京：机械工业出版社，2014．

[3] [美] Mark Allen Weiss．数据结构与算法分析：C 语言描述．冯舜玺译．北京：机械工业出版社，2004．

[4] [美] Brian W.Kernighan，Dennis M.Ritchie．C 程序设计语言（第 2 版 • 新版）．徐宝文，李志译．北京：机械工业出版社，2004．

[5] [美] Robert Sedgewick，Kevin Wayne．算法（第 4 版）．谢路云译．北京：人民邮电出版社，2012．

[6] [美] 本贾尼 • 斯特劳斯特鲁普．C++程序设计语言（第 1～3 部分）．王刚译．北京：机械工业出版社，2016．

[7] 刘汝佳．算法竞赛入门经典（第 2 版）．北京：清华大学出版社，2014．

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

[9] [德] Reinhard Diestel．图论（第 4 版）．于青林，王涛，王光辉译．北京：高等教育出版社，2013．

[10] [美] Richard A. Brualdi．组合数学（原书第 5 版）．冯速等译．北京：机械工业出版社，2012．

[11] [美] Joseph H. Silverman．数论概论（原书第 3 版）．孙智伟，吴克俭，卢青林，等译．北京：机械工业出版社，2008．

[12] [美]斯基纳，[西]雷维拉．挑战编程：程序设计竞赛训练手册．刘汝佳译．北京：清华大学出版社，2009．

[13] 俞勇．ACM 国际大学生程序设计竞赛：算法与实现．北京：清华大学出版社，2013．

[14] [美] Kenneth H.Rosen．初等数论及其应用（原书第 6 版）．夏鸿刚译．机械工业出版社，2015．

[15] [英] B.Bollobas．现代图论．北京：世界图书出版公司，2003．

[16] [美] Harold Abelson，Gerald JaySussman，Julie Sussman．计算机程序的构造和解释（原书第 2 版）．裘宗燕译．北京：机械工业出版社，中信出版社，2004．