USACO 题解

Chapter1

Section 1.1

Your Ride Is Here (ride)

这大概是一个容易的问题，一个“ad hoc”问题，不需要特殊的算法和技巧。

Greedy Gift Givers (gift1)

这道题的难度相当于联赛第一题。用数组 incom、outcom 记录每个人的收入和支出，记录每个人的名字，对于送礼人 i，找到他要送给的人 j，inc(incom[j],outcom[i] div n)，其中 n 是要送的人数，最后 inc(incom[i],outcom[i] mod n)，最后输出 incom[i]-outcom[i]即可。（复杂度 O(n^3)）。

用 Hash 表可以进行优化，降复杂度为 O(n^2)。

Friday the Thirteenth (friday)

按月为单位计算，模拟运算，1900 年 1 月 13 日是星期六（代号 1），下个月的 13 日就是代号(1+31-1) mod 7+1 的星期。

因为数据小，所以不会超时。

当数据比较大时，可以以年为单位计算，每年为 365 天，mod 7 的余数是 1，就是说每过一年所有的日和星期错一天，闰年第 1、2 月错 1 天，3 月以后错 2 天。这样，只要先求出第一年的解，错位添加到以后的年即可。

详细分析：因为 1900.1.1 是星期一，所以 1900.1.13 就等于(13-1) mod7+1=星期六。这样讲可能不太清楚。那么，我来解释一下：每过 7 天是一个星期。n 天后是星期几怎么算呢？现在假设 n 是 7 的倍数，如果 n 为 14，那么刚好就过了两个星期，所以 14 天后仍然是星期一。但如果是过了 15 天，那么推算就得到是星期二。这样，我们就可以推导出一个公式来计算。(n 天 mod 7（一个星期的天数）+ 现在日期的代号) mod 7 就等于现在日期的代号。当括号内的值为 7 的倍数时，其代号就为 0，那么，此时就应该是星期日这样，我们可以得出题目的算法：

int a[13]={0，31，28，31，30，31，30，31，31，30，31，30，31}

int b[8]={0}

a 数组保存一年 12 个月的天数（因为 C 语言中数组起始下标为 0，所以这里定义为 13）。

b 数组保存星期一到星期日出现的天数。用 date 记录目前是星期几的代号，然后用两个循环，依次加上所经过的月份的天数，就出那个月是星期几，当然，要注意判断闰年！知道了这个方法，实现起来就很容易了。

注意考虑闰月的情况。

最后注意要换行，否则会错误。

Broken Necklace (beads)

这道题用标准的搜索是 O(n^2)的，可以用类似动态规划的方法优化到 O(n)。

用数组 bl，br，rl，rr 分别记录在项链 i 处向左向右收集的蓝色红色珠子数。

项链是环形的，但我们只要把两个同样的项链放在一块，就把它转换成线性的了。

我们只要求出 bl，br，rl，rr，那么结果就是 max(max(bl[i]，rl[i])+max(br[i+1]，rr[i+1])) (0<=i<=2*n-1)。

我们以求 bl，rl 为例：

初始时 bl[0]=rl[0]=0

我们从左往右计算

如果 necklace[i]='r'，rl[i]=rl[i-1]+1，bl[i]=0；

如果 necklace[i]='b'， bl[i]=bl[i-1]+1，rl[i]=0；

如果 necklace[i]='w'，bl[i]=bl[i-1]+1，rl[i]=rl[i-1]+1。

同理可以求出 br，rr。

事实上不必使用动态规划，直接搜索亦可达到 O(n)。

把两个同样的项链放在一块，从头开始用两个变量(变量)a，b 记录自左方某点至目前为止可搜集到之两种颜色珠子数，取途中所出现 a+b 之最大值，遇颜色变换时再将 b 指定给 a 即可，代码十分简洁。

思路二：每次将串 s 的首位移动至末位，每次均能从两头开始搜索，无需考虑环的问题。

思路三：无需考虑 w 的。进行分类讨论，只有 rb 和 br 两种情况。

Section 1.2

Milking Cows (milk2)

有三种思想。

离散化（其实就是进行了优化的搜索而已）

按照开始时间升序排序，然后从左到右扫一遍，复杂度是 O(nlogn+n)的（排序+扫一遍，用堆、合并、快排都可以）。

所谓从左到右扫一遍，就是记录一个当前区间，[tmp_begin,tmp_end]。

如果下一组数据的 begin 比 tmp_end 的小（或相等），则是连接起来的，检查这组数据的 end，取 max{end,tmp_end}。

如果下一组数据的 begin 比 tmp_end 的大，则是相互断开的，整理本区间，ans1 取 max{tmp_end - tmp_begin,ans1}。ans2 取 max{begin - tmp_end,ans2}。

看不懂？去看看 PASCAL 或 C 的范例程序就明白了。

线段树

本题的规模是 1e6，简单的模拟是 O(nm)(n 是奶牛个数，m 是最大范围)的，会超时。（但是本题数据远没有描述的那么恐怖，直接模拟也是很快的）。

用线段树统计区间，复杂度降为 O(nlogm+m)，可以接受。

标记数组（哈希）

1e6 的范围，开一个布尔数组完全可以，有人为 TRUE，无人为 FALSE，注意边界即可。最后线性扫描即可。

时间复杂度，应该是 O(n)，n 为最后结束的时间。缺点就是……比较慢。叫什么好呢？并查集加速直接模拟。

记录一个 fa[i]表示 i 之后第一个没覆盖的点。下次遇到这里的时候就可以直接跳过了。复杂度大概算 o（n）吧。

Transformations (transform)

设 a 是原始状态，b 是改变后的状态。

水平翻转：b[i，n-j+1]:=a[i，j]

右旋 90 度：b[j，n-i+1]:=a[i，j]

枚举方案就行了，或直接枚举变换。

需要注意的是，USACO 是不给用 GOTO 的。注意代码的清晰程度。

Name That Number (namenum)

一个数字对应 3 个字母，如果我们先枚举出数字所代表的所有字符串，就有 3^12 种，

然后再在 5000 的字典里寻找，可以用二分查找，数据规模是 3^12*log5000=6.5e6，空间规模是 5000。

其实可以做的更好！

一个字母只对应一个数字，从字典中读入一个单词，把它转化成唯一对应的数字，看它是否与给出的数字匹配，时间规模是 5000*12=6e4，空间规模是常数，而且编程复杂度较低。还可以先比较字符长度和数字长度，如果相等，逐位比较。

把字母对应成数字我给出个公式，证明不写了，很容易想明白：temp:=ord(nam[i][j])-64;if temp>17 then dec(temp)；

num[i]:=num[i]*10+((temp-1) div 3)+2;temp 是第 i 个名字的第 j 个字符 ASCII 码-64 后的值，num[i]是地 i 个名字转换成的数字。

还有一个方法，利用哈希表，把字典进行哈希. 然后对于新产生的字符串，在哈希表里进行查找，找到了则加入输出列表. 最后则快排后输出。

补充一下：可以用集合来处理 s[2]=['A'，'B'，'C']; s[3]=['D'，'E'，'F']; 由于以 3 为周期，这种集合的建立可以由 div 3 得到但注意其中挖掉了'Q'，这可以分两段来建立集合。

另一种思路：

如果我们就使用普通的搜索，可以这样优化：设一个有效奶牛名数量统计 s，在读字典时，首先粗略判断字典中的奶牛名是否是我们所要的（如：长度是否和 input 文件里一样，首字母代表的数字是否与 input 文件中的数字一样等等。注意 input 文件中的数字最好读成字符串处理。）如果不合法，s 就不加 1. 这样最终剪枝之后，每个数据的有效名称不超过 200 个，算最坏情况，时间复杂度共计 O(5000+200)，可以说相当优秀。

Palindromic Squares (palsquare)

这道题唯一的知识点就是数制的转换。

思路：好像没什么难的，主要就是考进制转换，以及回文数的判断。这里要注意，最大的 20 进制中 19 表示为 J，不要只 CASE 到 15 哦！

穷举 1~300 的所有平方数，转进制，比较，OK 了，除非你不会怎么转进制。短除，然后逆序输出。

Dual Palindromes (dualpal)

因为数据很小，所以只需要从 s 开始枚举每个十进制数然后判断就行了。

参见进制转换，但并非最优算法。

Section 1.3

Mixing Milk (milk)

简单的贪心算法：将所有的牛奶按价格升序排列（用快排），然后从低到高买入，直到买够 m 为止。

贪心的证明：

假设某次买了价格稍高的牛奶，可以得到最优解。那么把这次买的牛奶换成价格更低的牛奶，其它不变，那么所得的解较优。假设不成立。

利用桶排的思想可以把代码压缩到极限，参见代码三。因其价格范围为[0..1000]可以用计数排序来做，就可以得到一个傻瓜代码(参见代码四)。

最佳解题方法：

因为价格的范围在 1..1000 之内，所以，我们只要在读入数据的时候把相同价格的合并即可，进行计算时，也只需要 for i:=0 to 1000 do 进行扫描如果有价格为 i 的牛奶就收购即可，所以不需要排序。

Prime Cryptarithm (crypt1)

思路一

要使木板总长度最少，就要使未盖木板的长度最大。

我们先用一块木板盖住牛棚，然后，每次从盖住的范围内选一个最大的空隙，以空隙为界将木板分成两块，重复直到分成 m 块或没有空隙。可以用二叉堆来优化算法。贪心的证明略。

思路二

显然，当所有木板均用上时，长度最短（证明....）。正向思维，初始状态有 c 块木板，每块木板只盖住一个牛棚。由 c 块倒推至 m 块木板，每次寻找这样的两个牛棚：其间距在所有未连接的木板中最小。当这两个牛棚的木板连接时，总木板数减 1，总长度增加 1。

思路三

还可以用动态规划求解，将有牛的牛棚排序后，设置一个函数 d[i，j]表示第 i 个牛修到第 j 个牛需要使用的木版长度，设 f[i，j]表示用前 i 个木版修到第 j 头牛所用的最短长度。

f[i，j]=f[i-1，k-1]+d[k，j]　(i<=k<=j)

思路四

显然，当所有木板均用上时，长度最短。用上 m 块木板时有 m-1 各间隙。现在的目标是让总间隙最大。将相邻两个有牛的牛棚之间间隔的牛棚数排序，选取最大的 m-1 个作为间隙，其余地方用木板盖住即可。

Barn Repair (barn1)

没什么好说的，硬搜，数据刚好不会超时。

枚举中间数（也就是认为它是回文数最中间的字母），然后左右扩展（忽略非字母）至出界和不相等。最后更新最大值。要考虑回文是奇数和偶数两种情况。提示大家在开始扩展之前就判断（很巧妙，大家举几个例子就可以明白了）。输入中的换行符可以维持原样不变, PASCAL 不会输出成乱码。

来说说一种 O(n)的动态规划算法。方程 f[i]=f[i-1]+2 如果(a[i]=a[i-f[i-1]-1])

或 f[i]=2 如果(a[i]=a[i-1])

或 f[i]=1

其中 f[i]是以恰好第 i 个字符结尾的回文串最大长度。速度巨快如雷电……

另一种解法：

1.初始化答案为 0。S=T+'#'+Reverse(T)+$，得到串 S（O(n)）。

2.求出后缀数组 SA、名次数组 Rank （倍增法：O(nlogn) Dc3 算法：O(n)）

3.计算 height 数组并进行标准 RMQ 方法预处理（O(n)）

4.枚举 i，计算以 i 为对称中心的极长回文串并更新答案（O(n)）。

扩展 kmp+分治复杂度: O(nlogn)

Calf Flac (calfflac)

S1　S4　S5

×　　S2　S3

约束条件只有 3 个：第 3、4 行是 3 位数，第 5 行是 4 位数。按 S1 到 S5 的顺序搜索。

如果 S1×S2>10 或者 S1×S3>10，则 3、4 行肯定不是 3 位数，剪枝。

即 S1*S2+S4*S2/10>=10 || S1*S3+S4*S3/10>=10 剪枝。

补充：从高位到低位，从小到大填数据，如果发现乘积超过 999 就剪枝。

Section 1.4

Packing Rectangles (packrec)

只要将示例中的 6 种（其实是 5 种，图 4 和 5 是同一种情况）进行模拟，以求得最优解。

The Clocks (clocks)

可以用 bfs，dfs，枚举，数学方法解。

位运算加速

说说位运算加速吧，不仅编程复杂度低，效率也高。注意到时钟只有四种状态，12 点，3 点，6 点，9 点。令它们为 0，1，2，3（这样的定义好处很大）。

这样，它们对应的二进制数为：000，001，010，011。即，我们用三个位来记录一个时钟的状态（为什么不用两位？请思考）。

要 tick 一个时钟的时候，就给该位加上一，再用按位与的方法去除高位的 1。

令最高的三位为时钟 A，最低的三位为时钟 I，那么：

数“57521883”用于清除每个时钟状态最高位的一（按位与）。

const long move[9] = {18911232， 19136512， 2363904， 16810048， 2134536， 262657， 36936， 73， 4617}。

move[i]表示题述中的第 i+1 种方法。

令 f[q]为原状态，比如用题述中的第 k 种方法，那么可以写成 f[q + 1] = (f[q] + move[k - 1]) & 57521883；

当 9 个时钟都回归 12 点的时候，巧的是状态 f=0。这样，判断每个状态 f 是否为 0，就知道是否求出可行解。

方法 1：枚举

因为每种变换方法可以使用 0~3 次，并且每种变换方法在前与在后是等效的。所以，利用递归，按题中顺序枚举所有可能的解，并且每次记录下途径，那么第一种解就是我们要求的答案，输出即可。并且此方法简单易懂，复杂度很低。在 usaco 上测试最大的才用了 0.097 秒。

方法 2：DFS

显而易见地，方案的顺序并不重要。而每种方案最多只能选择 3 次，如果 4 次相当于没有选择。这样，总的搜索量只有(4^9=262144)，用 DFS 搜索所有状态，选择其中一个最小的输出即可。可以采用位运算加速。

方法 3：BFS

用 BFS 完全可以完美解决本题，虽然有些划不来（毕竟枚举都可以过），但思想是可以值得借鉴的。1.判重，把 3，6，9，12 分别对应于 1，2，3，0，这样一个状态就对应一个 9 位 4 进制数，hash 数组就开到 4^9 左右 2.剪枝(谁说 BFS 不能剪枝)，其实同 DFS 的剪枝，按非递减顺序扩展，每个操作最多做 3 次 3.空间，如果开 longint 可能会 MLE，于是改成 byte，成功！如此一来，BFS 可以应对所有数据，最大的也只要花了 0.119s 详细见 pascal 代码

方法 4：数学方法

假设时钟 abcdefghi 中除了 a 都到了某一状态，现在有一系列转换可以使 a 转 90 度而其它钟不变，进行这一系列的转换，虽然行动次数多了，但钟没有变，已知没有可以直接转 a 的转换方法，所以其它的钟都是转了 360*n 度的(0<=n 且 n 为整数)就是自己独自转了 4*n 次(0<=n 且 n 为整数)，既然是 4 的倍数，那么就可以 mod 4(and 3 来的更快些)来抵消掉这些操作而钟的位置没变。这样就得出一个结论了，用枚举或 bfs 得出的最优解，它的解的前一个步骤为之一个状态.用刚才的理论，对于每个没到 12 点的钟作只让它转的一次的一系列操作，这样达到了全为 12 的一种状态，然后将每种操作 mod 4，用枚举或 bfs 得出的最优解的前一个步骤时所做的步骤+最优解的最后一个步骤，这样继续向前推，就可以得到从初始状态开始，直接执行对于每个没到 12 点的钟作只让它转的一次的一系列操作，然后最后结果每种操作 mod 4，就是枚举或 bfs 得出的最优解。

方法 5：传说中的“解方程法”

和方法 4 差不多，原理便是对应每一种地图状态，都会有唯一的最优解（因为顺序的问题已经被题目本身排除了），所以对应每一个方法都可以列出九元一次式，然后根据输入数据用伟大的高斯同学的消元法解之然后取最优解即可。复杂度只是 O(1)。

方法 6：直接输出法

先编程求出只旋转其中一个钟的操作序列然后将这个结果作为 const 存储随后，将每个钟所需要的操作叠加起来，最后分别 mod 4。

Arithmetic Progressions (ariprog)

这道题就是暴力搜索，时限是 5s，方法是很简单的：枚举所有的可能解，注意剪枝。

但是在编程细节上要注意，很多时候你的程序复杂度没有问题，但常数过大就决定了你的超时（比如说，你比别人多赋值一次，这在小数据时根本没有区别，但对于 1 个运行 5s 的大数据，你可能就要用 10s 甚至更多）。

预处理把所有的 bisquare 算出来，用一个布尔数组 bene[i]记录 i 是否是 bisquare 另外为了加速，用 list 记录所有的 bisquare（除去中间的空位置，这在对付大数据时很有用），list 中的数据要有序。

然后枚举 list 中的数，把较小的作为起点，两数的差作为公差，接下来就是用 bene 判断是否存在 n 个等差数，存在的话就存入 path 中，最后排序输出。此时缺少重要剪枝，会超时

思考程序发现，费时最多的地方是枚举 list 中的数，所以对这个地方的代码加一些小修改，情况就会不一样：

1.在判断是否存在 n 个等差数时，从末尾向前判断（这个不是主要的）。

2.在枚举 list 中的数时，假设为 i，j，那么如果 list[i]+(list[j]-list[i])×(n-1)>lim（lim 是最大可能的 bisquare），那么对于之后的 j 肯定也是大于 lim 的，所以直接 break 掉。（这个非常有效）。

AC，最大数据 0.464s(请看 C++的 GPF 的程序)。

其实输出时可以不用排序，用一个指针 b[i]存公差为 i 的 a 的链表，由于搜索时 a 是有序的，存到 b[i]中也应是有序的，这样就可以直接输出。对极限数据 b 的范围应该不超过 m^2/n=2500，即 b:array[1..2500]of point; 而如果 qsort 的话，复杂度为 n*logn，n<=10，000

枚举 a，b 也可以过，要加几个小优化。

Mother's Milk (milk3)

方法一：DFS

因为牛奶的总量是不变的，所以可以用 a，b 中的牛奶量做状态，初始状态是(0，0)，每次只能有 6 种选择，a 倒 b，a 倒 c，b 倒 a，b 倒 c，c 倒 a，c 倒 b。用一个数组 vis[i][j]判重，s[i]记录 c 中所有可能值（s[i]=true 表示 c 中可能出现 i），如果当前状态是(0，x)，那么 s[mc -x]=true，最后输出 s 中所有 true 的就可以了。

方法二：递归

用 i=14 作为临界点能过，不过不知道为什么，没证明过当 i 等于 14 时能包括大多数的过程。 (骆驼绒同学补充了一下，他说用 BFS。只有在扩充的状态没有出现过的情况下，才把它加入队列，这样可以保证不重不漏）。

这里用到了一个输出技巧，当你不知道最后一个输出数据是什么而且又不能留空格时，我先把所有数据加空格存到一个 String 里，然后再把 String 里的最后一位去掉就 OK 了。

方法三：枚举

使用枚举的方法，按照顺序 A，B，C 寻找每一个牛奶非空的桶，假设我们找到了 B，则接下来只能是从 B->A 或 B->C 或则什么都不做这三种情况；使用递归来完成。我们需要

一个数组 state[500][3]来储存遍历过的三个桶中牛奶的状态，当 当前状态已经在之前出现过，就退出。同样也是使用一个数组 bool leave_c[]来储存 C 中可能的牛奶量。具体代码可以参考(c++ ID=herohan1).这样或许更容易理解。

Section 1.5

Number Triangles (numtri)

简单的动态规划。

设 f[i，j]表示到达第 i 层第 j 个最大的分数。状态转移方程： f[i，j]=Max{f[i+1，j]，f[i+1，j+1])+a[i，j] (1<=i<=n-1，1<=j<=i)我们只需要知道下层的情况，所以可以用滚动数组来记录，时间复杂度 O(n^2)，空间复杂度 O(n)。

Prime Palindromes (pprime)

这道题有两种思路：

用筛法求出 1..1e8 范围内的素数，然后判断每个素数是否是回文数。

生成 1..1e8 范围内的回文数，然后判断它是否是素数。

思路 1 的复杂度是 O(n)，思路 2 的复杂度是 O(√n* √n)=O(n)，从复杂度来看两种思路没有差别。但思路 1 用筛法用素数要开 O(n)的数组，在 n=1e8 是就是 90M，超出了空间限制，（而且有可能超时），而思路 2 的空间复杂度是 O(1)的，所以我们用思路 2。

设生成位数为 l 的回文数，若 l 是奇数，那么从小到大枚举(l+1) div 2 位的数，然后复制翻转生成一个回文数；若 l 是偶数，那么从小到大枚举 l div 2 位的数，然后复制翻转生成一个回文数。上诉两个过程交替进行就可以从小到大生成回文数了。

很有效的优化：任意偶数长度的回文数都不可能为质数（除了 11），因为它能被 11 整除，而 11 恰好只有自身和 1 两个因子。除 2 外，所有偶数均不可能是质数。

还有一个优化：尾数为 5 必不是质数。

第一种思路的可行性

直接使用筛法空间 90M，但注意两点，一，题中最大 size 为 1e8，实际上由于偶数长度无质数回文，只需开 1e7 即可。另，可以使用位操作来存储和操作标记数组。使用这两点之后，直接暴力筛，按素数查询是否为回数，在不跳过偶数长度的情况下，0.356 secs， 4148 KB 通过。比一般的第二种思路实现更快。（参考 blackco3 的 C++代码）

猥琐的解法

打表，将 5~10^8 之间的回文素数记录下来，在其中查找 a 和 b。

另一种解法

思路同样是先生成回文，然后判断生成的回文是否是素数。但是生成回文的方法有些不一样：

我们先生成一个数组 int p[7]，其中 p[0]表示回文的个位数，p[1]表示回文的十位数；这样我们可以采用 DFS 进行搜索。因为回文具有 p[i]=p[digits-i-1]， 所以 DFS 的深度为(digits-1)/2.假设所给最大数的位数为 N，则在对 N 进行 DFS 的过程中，我们事实上已经得到了位数 M=1，2，..N 的回文数。但是这样产生的回文并没有按照从小到大的顺序排列。因此我们可以设置一个 bool pp[9999999]的数组来判断，这样可以省去回文的排序。但是需要用 new 来分配空间。同样使用了优化：1）所有偶数都不是质数，2）任意偶数长度的回文数都不可能为质数（除了 11）这种方法的时间还是比较好：[0.130 secs， 12700 KB]通过，可以参考代码

SuperPrime Rib (sprime)

从数学的角度：1.首位只能是质数 2，3，5，7

2.其余位只能是 1，3，7，9

3.若 n=1，直接输出 2，3，5，7

DFS

不需要预处理。直接 DFS 1～9，加入当前数末尾，并判断是不是素数，是则递归处理下一位数，不是则回溯，直到 depth>n。不会超时。

用预处理…否则貌似会超时…先生成 1 位的质数，再用 1 位的质数生成 2 位，然后用 2 位的生成 3 位…就这样递归下去….最后一位直接全部输出然后 Halt 就好了… 下面代码是表示在原质数的基础的后面再加上一个奇数(因为质数除了 2 不可能是偶数)

tmp:=a[i-1，k]*10+b[l];

生成 tmp 之后再判断 tmp 是不是质数，易证循环只要到 根号 tmp 过就好了。

Checker Challenge (checker)

DFS+优化

因为要遍历整棵搜索树，所以用 DFS，我们可以在搜索时加入剪枝，记录横行和两条斜线是否被攻击，每次只放在不受任意方向攻击的地方。这个剪枝过最后一个数据需要 2s，超时。(准确的说，是 1.2s -Seek Final)

考虑对称

如果 n 是偶数，第一行枚举前一半的数，这样每搜索出一种方案后，沿中线对称过来又是一种方案，并且因为第一行的数小于一半，对称过来的方案总大于原方案（即在搜索树中后被搜索到），不会出现重复的情况。

如果 n 是奇数，先在中间行和中间列放子，并且位置都不超过半数(<n div 2)，且中间行>中间列，这样每搜索出一种方案，把它对称旋转后一共有 8 种方案，因为中间行和中间列的的不出现重复，所以 8 种方案不重复。这样只需枚举原来的 1/8 就可以了。这样就完全可以过了。{注意放在正中间位置的时候}

使用链表

还可以再优化，用链表（或用数组模拟）记录每行待选的位置，进行插入和删除，这样就不用记录横行是否被攻击了，并且每次枚举位置的个数减少。

最后一位算出来

由于 n 行每一行都要有一个，那么所有的皇后的位置之和必然为 n×(n+1)/2，在 dfs 的时候记录已经枚举的皇后位置之和，那么最后一个可以直接算出来。这样的话最后一个点 0.907s。

DFS+位运算

可以考虑将问题分为两个子问题来处理:

a)求 n 皇后具体的方案;

b)求 n 皇后方案的个数;

对于问题 a，因为只取前三个，用最简单的 DFS 即可办到.

对于问题 b，可以考虑采用位运算，来递归模拟试放(考虑的条件与 DFS 类似)，但不用储存具体方案.

用此方法通过全部数据耗时约 0.4secs，根据位运算的具体操作细节时间还可以更快。

Chapter2

Section 2.1

The Castle (castle)

Floodfill。

用一个二维数组记录每个格的四面是否有墙，根据输入，利用位运算得出每个格的墙的

情况（8 = 23，4 = 22，2 = 21，1 = 20）。 然后用 floodfill，对每个房间染色，求出房间数，和每个房间的面积，输出其中最大的。

枚举每堵墙，如果墙的两边不是同一个房间且其面积和更大，更新结果，为了结果唯一，我们从左下向右上，一列一列枚举格子，并且先枚举该格上面的墙再枚举右边的。

墙 1：西；2：北；4：东；8：南。我们可以用一个三维的布尔型数组来判断有没有墙，false 即为有墙，TRUE 为无墙。那么我们在读入时可以做以下操作：

并查集 也是可以的嘛…判断墙的状况可以用位运算。

连通分支 用链表记录，指针的元素记录下当前的行号列号与所指向的连通分支。

Ordered Fractions (frac1)

优化

1.0/1，1/x 不能漏了！有 1/1

2.若是两个偶数，不用判断绝对不行

3.两个分数比较大小，只需交叉相乘即可，无需另开 data 储存具体值。

如：a1/b1<a2/b2 ==>a1*b2<a2*b1

快排

枚举所有的分数，判断其是否是最简（分母分子最大公约数＝1），用一个数列记录所有最简分数，然后用快排排序。

归并

显然，0/i 1/i 2/i...i/i 这个序列是有序的对于 n 个序列归并即可（相等则取分母最小的一个——这样显然是最简分数）。

数学（来自 Russ 的更优算法）

我们可以把 0/1 和 1/1 作为“端点”，通过把两个分数的分子相加、分母相加得到的新分数作为中点来递归（如图）

0/1 1/1

1/2

1/3 2/3

1/4 2/5 3/5 3/4

1/5 2/7 3/8 3/7 4/7 5/8 5/7 4/5

每一个分数的分子和分母都是由求和得来的，这意味着我们可以通过判断和与 N 的大小关系来判断递归边界。

Sorting A Three-Valued Sequence (sort3)

Way1

用 l[i]记录 i(1，2，3)出现的个数。排序后一定是 l[1]个 1，l[2]个 2，l[3]个 3 这 3 段。

sn[i，j]记录在第 i 段中的 j 的个数。第 i 段中的 j 和第 j 段中的 i 交换，两个元素交换，只需要一次。所以取 sn[i，j]和 sn[j，i]中的较小数 s，累加到总交换数中。

这样交换后，剩下的肯定是 3 段同时交换，最少需要 2 次。我们只需累加(sn[1，2]+sn[1，3])×2 即可。

Way2

首先读入所有数，然后看原本的位置是“1”的里面有几个不是 1，如果那几个不是 1 的数字在它原应该呆的位置中找到 1 的话就直接换，如果找不到就在另一堆中找。

比如说 3 1 2 三个数，第一个位置原应该是 1，但是是 3，那么就在原应该是 3 的位置中找 1，但是是 2，那么就在另一堆找，找到了 1，那么把 3 跟 1 换一下，直到 1 全部到达自己的位置。然后在原应该是 2 的位置中找，如果碰到 3 就应该换一次，就这样找完就行了。

for i:=1 to n do

```
begin
  readln(a[i]);
  if a[i]=1 then inc(s[1，2])
  else
  if a[i]=2 then inc(s[2，2]);
end;
s[2，1]:=s[1，2]+1;
s[2，2]:=s[2，2]+s[2，1]-1;
s[3，1]:=s[2，2]+1;
```

这段代码是记录位置的，s[x，1]是 x 的初始位置，s[x，2]是 x 的终点位置。

记录好位置之后就按照上面的流程去找就好了，实现起来应该是很简单的。

Way3

这是极其诡异的做法：只要类似与方法二，记录下本是 N 的位置却不是 N 的个数 A[N]，表示 N 这个数有多少个不在自己的位置上。

然后，开一个数组 B，存储排序后的序列。每次比较原数列 Data[i]和 B[i]，如果不相等，则 dec(A[data[i]])，dec(A[b[i]])，inc(Total)。

本来只想着尝试这样的方法，没想到交上去后就 AC 了。莫名……只是求最小次数的问题这样可以 AC……

Way4

O(n)预处理，O(1)计算（注：数字 i 该在的位置即排序后数字 i 所在的位置区间）。d[i，j]表示排序后数字 i 该在的位置中含有的数字 j 的个数。

那么，ans:=d[2，1]+d[3，1]+d[2，3]+max(0，d[1，3]-d[3，1])。

原理很简单，首先要把所有的 1 交换到最前面:用的次数是 d[2，1]+d[3，1]。然后把所有的 3 交换到最后，这些 3 共两部分：一部分是在 2 该在的位置中的，这些需要 d[2，3]次交换；另一部分是在 1 该在的位置中的，这些 3 中有一些是通过处理 1 已经回来了，还有一些可能在处理 1 的时候交换到了 2 该在的位置，就需要再交换一下。这部分交换总数是 max(0，d[1，3]-d[3，1])。

我用的是这个方式的变形：ans:=d[1，2]+d[1，3]+max(d[2，3]，d[3，2])；

处理解释：d[1，2]+d[1，3]是所有占据 1 位的 2 和 3 的数量，也就是不在正确位置的 1 的数量（d[2，1]+d[3，1]），这两个加和是相同的。第一步求这个和，也就是让所有 1 归位。

此操作之后，存在两种情况。

一：原序列中不存在三值互换的现象，那么 d[2，3]与 d[3，2]应该是相同的。max 算子不起作用。

二：原序列中存在三值互换的现象。可知，d[2，3]与 d[3，2]中必有一个不变，而另一个在增大，且增大的量在数值上等于原三值交换的次数。就是例如上段文字所说：这些 3 中还有一些可能在处理 1 的时候交换到了 2 该在的位置，这时增加的就是 d[2，3]，而 d[3，2]不变。由于交换是相对的，变化后的 d'[2，3]与 d'[3，2]必相等，又由于其中有一个值没有变动，所以该值也就等于原 d[2，3]与 d[3，2]中的最大值。

然后取这个最大值就可以让 2 3 同时归位。问题得解。

补充一点

我们用 a[i，j]表示在 i 的位置上 j 的个数，比如 a[2，1]=5 就表示排好序后，上面应该是 2，但现在被 1 占领的位置数是 5。先贪心到不能两值内部交换。

那么操作之后不会存在 a[2，1]>0 和 a[2，3]>0 同时成立的现象。

反证法:比如交换之后 a[2，1]>0，且 a[2，3]>0 则在 1 的位置上只能有 3(1 和 2 能内部

相抵的已经全部抵消了)，3 的位置上只能有 1(同理)，那么 1 和 3 又可以内部交换了，与假设矛盾。得证。

还有最后 3 值交换是乘 2，而不是乘 3。

Healthy Holsteins (holstein)

对于每种饲料，只有两种选择：喂或者不喂。所以最多只能有 2^15 种方案，简单的枚举就可以了。可以枚举 1 to 2^p-1 的长度为 p 的二进制来获得每个养料是否要，0 表示不要，1 表示要。

如 m=3 则枚举 3 位的二进制: 001 010 011 100 101 110 111 以上就对应了除了都不取外的 7 种情况。

或者我们也可以 dfs，决策是取或不取第 now 个原料，当不可能是最优解(nown>totn)时回溯；如果营养满足就记录最优解回溯；如果 g 个饲料都决策完了但营养仍不足则回溯。很容易写，很漂亮。

Hamming Codes (hamming)

我们只要按递增顺序搜索要求的 n 个数，然后跟前面的数判断距离是否大于 d，在找到一组解后它肯定是最小的，输出。 数据不大，暴力搜索即可。注意：0 是必须出现的。

几个优化：

b 给出了搜索的最大值：2^b-1(这条好像不是优化) 。

计算两个数 a，b 的距离，只要计算 a xor b 的数的二进制形式中 1 的个数。

这样可以做一个预处理，把所有二进制中 1 的个数≥d 的数算出来（用 can[i]记录 i 是否是满足要求的数）。

或者不优化，直接从小到大（1--255）直接与已经生成的比较，若满足，则加入结果中。

Section 2.2

Preface Numbering (preface)

分析一：枚举

首先大多数人都能想到的算法就是枚举，因为枚举容易易懂。虽说枚举不超时，但是我们也应该尽可能优化，使程序更快速。所以我们应该制造一个数字表：

```
const
shu:array[1..4，0..9] of string=((，'I'，'II'，'III'，'IV'，'V'，'VI'，'VII'，'VIII'，'IX')，//个
                                 (，'X'，'XX'，'XXX'，'XL'，'L'，'LX'，'LXX'，'LXXX'，'XC')，//十
                                 (，'C'，'CC'，'CCC'，'CD'，'D'，'DC'，'DCC'，'DCCC'，'CM')，//百
                                                          (，'M'，'MM'，'MMM',,,,,,，))；//千
```

那么之后进行枚举时就可以加快速度……

分析二：数学问题

这道题的 n 只有 3000 多，从 1~n 把根据题目转换成字母，然后统计出现次数也不会超时。不过这里介绍另一种更高效的算法。

0~9 中 IVX 出现的次数和 10~19，20~29，x0~x9 中的相同（其它字母都未出现）。0~99 中仅十位产生的字母中，字母出现的次数相同，不同的是字母成了 XLC，后移了 2 位。

我们把 n 按十进制位从低到高的顺序统计每位出现的字母次数。

以 n=234 为例，

仅个位：23 个 0~9 的次数，1 个 0~3 的次数，1 个 4 的次数。(IVX)

仅十位：2 个 0~9 的次数*10，1 个 0~2 的次数*10，5 个 3 的次数。(XLC)

仅百位：0 个 0~9 的次数*100，1 个 0~1 的次数*100，35 个 2 的次数。(CDM)

分析三

数学统计首先，不考虑顺序，那么任意一个数对应的罗马数字，都可以用其他几个 n×10^k 表示，比如 999=900+90+9=CM+XC+IX;所以我们预先处理出 1~9，10~90，100~900，1000~3000 这几个数所对应的字符，这样再枚举效率就要高些（当然对于 3000 的数据看不太出来，要是 10^6 试试……）。

分析四

每一位对应三个字母

个位 I V X

十位 X L C

百位 C D M

千位 M

字母 A B C

然后：1 对应 A，2 对应 AA，3 对应 AAA，4 对应 AB，5 对应 B，6 对应 BA，7 对应 BAA，8 对应 BAAA，9 对应 C。

这样先写一个 1..9，A..C 的常量数组，然后对每一个数的每一位进行统计就 OK 了。

Subset Sums (subset)

分析一

n 个数的总和为 tot’:=n*(n+1)shr 1，当且仅当 tot’为偶数的时候才有解，tot’为奇数时直接输出 0 并且退出程序；但是这样做了也还是会超时。所以要进行优化处理。

分析二：动态规划

设分成的子集为 set1，set2，tot=(n*(n+1)div 2)div 2. 设 f[i，j]表示取前 i 个数，使 set1 总数和为 j 的方案数。

f[i，j]=f[i-1，j]+f[i-1，j-i]（j-i>=0）

f[i，j]=f[i-1，j]（j-i<0）

那么，最后结果为 f[n，tot] dp 步骤为：

```
f[1，1]:=1;
f[1，0]:=1;
for i:=2 to n do
  for j:=0 to sum do
    if j-i>=0 then f[i，j]:=f[i-1，j]+f[i-1，j-i]
    else f[i，j]:=f[i-1，j];
```

则最后的结果为 f[n，tot]div 2。

分析三

递推设 ans[i，x]表示在前个 i 元素里选择若干使其和为 x 的选法，有 ans[i，x]=ans[i-1，x]+ans[i-1，x-i]，最后的输出就是 ans[n-1，((n+1)*n div 4)-n]。

注意解不存在的情况单独判断，详细方法看分析一。

Runaround Numbers (runround)

方法 1

从开始数往后枚举，然后判断其是不是 runround number，如果是就输出退出，由于每次判断的复杂度是常数的，算法复杂度是 O(n)，不会超时。

如果枚举方法不当也会超时的!!!

方法 2

从开始数的位数开始，迭代搜索生成数，然后判断其是不是 runround number 且大于开始数，因为数字最多只有 9 位，且每位都不同，算法复杂度是 O(9!)，不会超时。

Party Lamps (lamps)

从按键个数而言，如果 C>=4，我们把它不断减小 2，使它小于 4（相当于把重复的按键去掉），最后我们再枚举每一种按键所能得到的结果，加以判断就可以了。

每个按钮按 2 次和没按效果是一样的。所以每个按钮或者按或者不按，一共有 2^4=16 中状态。枚举每个按钮是否按下，然后生成结果，排序输出即可（注意判重）。

c 的约束就是按下的按钮数<=c。(错) 貌似很难理解。

另外灯 1 和灯 7，2 和 8，3 和 9……是一样的因此当 N>=6 时只需处理前 6 个，排序时转换为 10 进制数， 输出时反复输出前 6 个的状态。

深究：

这道题如果深究的话会变得非常简单，但是提前声明，如果对这道题兴趣不大，或者是初学者，建议跳过，刚才的分析已经足以过这道题。我们现在记不按按钮，以及按下 1，2，3，4 按钮分别 O，①，②，③，④，那么，按下 3，4，可以记为③④，以此类推，我们发现一个问题，那就是①，②，③之间微妙的关系， ①②=③，而②③=①，①③=②（可以自己试试），于是我们知道，①②③也相当与不按，即相差 3 的倍数也可互相转换。

所以，所谓前四个的 16 种按法其实只有 8 种，分别为：O，①，②，③，④，①④，②④，③④。

然后讨论 c，由于当 c>4 时，均可化为当 c<=4 的情况，所以我们先讨论当 c<=4 的情况。

当 c=0 时，只有一种 O；

当 c=1 时，四种：①，②，③，④；

当 c=2 时，除了④均可（可以自己想想）；

当 c=3 时，由于 3-1=2，所以 c=1 的情况都满足，而在 c=2 中，把所有有前三类的展开，如①④变为②③④，可知满足 c=2 的同时满足 c=3，所以 c=3 其实是 c=2 和 c=1 的并集，即所有按法均可。

当 c=4 时，由于 4-1=3（①②③相当于不按），且 4-2=2，由上，c=4 也是所有按法均可。

当 c>4 时，我先有一个引理：对于任意的正整数 n>1，均可写成 n=2*p+3*q(p，q 为非负整数）的形式，证明如下：若 n 为偶数，必然成立，若 n 为奇数，必然大于 2，则 n-3 必为非负偶数，得证。由这个引理我们可以知道，任意 c>4 均可写成，c=2*p+3*q+3(p，q 为非负整数）的形式，而可知，对于两个相同的按键，以及情况①②③（按键三次），均相当于不按，所以任意 c>4 均可化归为 c=3 的情况，即当 c>4 时，所有按法均可。

综上所述，

当 c=0 时，只有一种 O ；

当 c=1 时，四种：①，②，③，④；

当 c=2 时，除了④均可；

当 c>2 时，所有按法均可。

好了，这样一来就非常简单了，只有四种情况，8 种按法。

另一种判断按下按钮数是否为 C 的方法（与上面的差不多）：

在判断是否按了 c 次时，就不好直接判断了，因为我们的思路是：“每个按钮按 2 次和没按效果是一样的。所以每个按钮或者按或者不按”但是实际上每个按钮按 2 次和没按效果有细微差别——就是按的次数。

为了解决这个问题，我从奇偶考虑：假如有两种情况结果相同，但按的次数不同，分别为 a，b。而如果 a-b 是偶数，那么按的次数少的那种情况完全可以通过按 2K 次来凑够次数（k 为正整数），因此，我们可以用 mod 语句来完成判断是否能凑够 c 次，用此方法可以发现不用判重了，因为我搜索的是基本情况，而其他的情况自然被无形的滤掉了具体过程如下：

（change(k)表示按第 k 个按钮，在这里就不贴出了。）

关于状态的记录，这题可以用位运算简化，如上面所说，用一个 longint 存储前 6 位的状态就行了。而 4 种操作也很简单，用异或就可以搞定：

操作 1：异或(111111)2=63

操作 2：异或(010101)2=21

操作 3：异或(101010)2=42

操作 4：异或(001001)2=9

判断状态是否符合题意，则可以用&和|完成：

先将题目给出的灯的状态映射到 0～5 之间，即(灯的编号-1)%6

生成一个 6 位二进制数 isOn，其中题目给出是 ON 的灯的对应位为 1，其余为 0

生成一个 6 位二进制数 isOff，其中题目给出是 OFF 的灯的对应位为 0，其余为 1

若当前状态 state 满足：

(state&isOn)==isOn && (state|isOff)==isOff

则 state 满足题目要求。

另外，关于判断某一状态是否能由 c 次按键得到，如果某一状态能由 x 次按键得到，若 c-x 为偶数，则该状态进行 c-x 次操作 1 后状态不变。

如果用 DFS 或枚举，x<=4 即可。

首先不要读错了题，不是正好按了 c 次，是<=c 次都可以。

其实不用这么麻烦。只用记录和前四个检验前四个灯就行了。

为什么呢。

1 号灯——代表了所有 3 的倍数+1 的奇数号灯；

2 号灯——代表了所有不是 3 的倍数+1 的偶数号灯；

3 号灯——代表了所有不是 3 的倍数+1 的奇数号灯；

4 号灯——代表了所有 3 的倍数+1 的偶数号灯；

读入数据的时候就可以处理。

Section 2.3

The Longest Prefix (prefix)

动态规划

设 dp[i]表示主串 S 中从 i 开始的最长的可组成的前缀，dp[1]就是所求，状态转移方程：

{dp[i]=max{dp[j]+j-i}|i 到 j-1 的字串是 primitive。 i<j<=n}

用 Hash 表判断 primitive，将每个 primitive 转成 10 位 27 进制数，再 mod 一个大质数。在判断 primitive 的时候只需把 i 到 j-1 的字串的 Hash 值算出来就可以了。

因为 primitive 最多只有 10 位，所以时间复杂度是 O(10n)，其中 n 是主串的长度。

我补充一下，判断 primitive 时，也可以用 kmp 算法提前造一个表，表示主串中所有的字串，这样的时间复杂度（仅造表）为 O（（m+n）*k），m 为基本串长（可忽略），n 为主串长，k 为基本串个数，比 hash 较高，但很稳定。

另一个转移方程：f[i]表示 s 前 i 个字符能否取得 f[i]:=(f[i-length(p[j])]) and (copy(s，i-length[p[j]]+1，length[p[j]])=p[j])；

i:=length(s); j:=P 集合中元素个数；

这个方程可以这样实现:对于当前位置 i，枚举每个词典如果，f[i]=true 且 pipei(i，length(table[j]))=true，那么 f[i+length(table[j])]=true.此时 ans:=max(ans，i+length(table[j])).并且，循环条件有两个:i:=1..length(s)和 i<=ans

初始化：fillchar(f，sizeof(f)，false); f[0]:=true;

最后取得最大的值为真的 f[ans]就行了 (当 i>ans 时即可 break 了) 。

Cow Pedigrees (nocows)

这是一个 DP 问题。我们所关心的树的性质是深度和节点数，所以我们可以做这样一张表：table[i][j]表示深度为 i、节点数为 j 的树的个数。根据给定的约束条件，j 必须为奇数。你如何构造一棵树呢？当然是由更小的树来构造了。一棵深度为 i、节点数为 j 的树可以由两个子树以及一个根结点构造而成。当 i、j 已经选定时，我们选择左子树的节点数 k。这样我们也就知道了右子树的节点数，即 j-k-1。至于深度，至少要有一棵子树的深度为 i-1 才能使构造出的新树深度为 i。有三种可能的情况：左子树深度为 i-1 ，右子树深度小于 i-1；右子树深度为 i-1，左子树深度小于 i-1；左右子树深度都为 i-1。事实上，当我们在构造一棵深度为 i 的树时，我们只关心使用的子树深度是否为 i-1 或更小。因此，我们使用另一个数组 smalltrees[i-2][j]记录所有深度小于 i-1 的树，而不仅仅是深度为 i-2 的树。知道了上面的这些，我们就可以用以下三种可能的方法来建树了：

table[i][j] += smalltrees[i-2][k]*table[i-1][j-1-k]；//左子树深度小于 i-1，右子树深度为 i-1

table[i][j] += table[i-1][k]*smalltrees[i-2][j-1-k]；//左子树深度为 i-1，右子树深度小于 i-1

table[i][j] += table[i-1][k]*table[i-1][j-1-k]；//左右子树深度都为 i-1 另外，如果左子树更小，我们可以对它进行两次计数，因为可以通过交换左右子树来得到不同的树。总运行时间为 O(K*N^2)，且有不错的常数因子。（官方标程见源码-c 部分）

首先明确一下题目的意思：用 N 个点组成一棵深度为 K 的二叉树，求一共有几种方法？设 dp[i，j]表示用 i 个点组成深度最多为 j 的二叉树的方法数，则：

dp[i，j]=∑(dp[k，j-1]×dp[i-1-k，j-1])(k∈{1..i-2})

边界条件：dp[1，i]=1

我们要求的是深度恰好为 K 的方法数 S，易知 S=dp[n，k]-dp[n，k-1]。但需要注意的是，如果每次都取模，最后可能会有 dp[n，k]<dp[n，k-1]，所以可以用 S=(dp[n，k]-dp[n，k-1]+v) mod v 。

Zero Sum (zerosum)

DFS

每层只有 3 个状态，层数最多只有 8 层，不会超时。

枚举时按照' '，'+'，'-'的顺序添加当前空格处的运算符，保证输出的顺序正确。用 sum 记录当前加和，用 last 记录当前的连续数，就是之前添' '连成的数。

' '：sum=sum-last+last*10+s+1(last>0)；sum=sum-last+last*10-s-1(last<0)；

last=last*10+s+1(last>0)；last=last*10-s-1(last<0)

'+'：sum=sum+s+1；last=s+1

'-'：sum=sum-s-1；last=-s-1

另一种方法是最后计算是否为 0。

注意:PASCAL 自带的字符串函数 val(s:string;var t:longint)中，s[1]可以为'+'或'-'，所以在计算时显得异常简单。

通过三进制枚举

实际上，我们也可以用三进制枚举，从 3^(n-1)到 1，每个数都转化为三进制。2 表示' '，1 表示'+'，0 表示'-'。具体见两个 pascal 代码。

Money Systems (money)

背包问题

设 dp[i，j]表示前 i 种货币构成 j 的方法数，用 cc 记录货币的面值，状态转移方程为：

dp[i，j]=dp[i-1，j]; 不用第 i 种货币

dp[i，j]=dp[i-1，j]+dp[i，j-cc[i]] 用第 i 种货币，j>=cc[i]

时间复杂度是 O(VN)的，如果把面值相同的货币记做一种，可以更快一些。参见 dd 牛的《背包 9 讲》。

可以优化得在空间上更好些。

```
for i:=1 to v do
        for j:=0 to n-w[i]do
                inc(f[j+w[i]]，f[j]);
```

f[i]表示构成 i 所有的方法数。输出 f[n]即可

分析 2 生成函数（母函数）

也许麻烦了点，但也是一种方法。

原问等价于求一个形如 a1X1+a2X2+...+avXv=n 的方程的非负整数解的个数。构造生成函数 f(x)=(x^0+x^a1+x^(2*a1)+x^(3*a1)+...)(x^0+x^a2+x^(2*a2)+x^(3*a2)+...)...(x^0+x^av+x^(2*av)+x^(3*av)+...) 则 x^n 的系数就是所求。

Controlling Companies (concom)

这里使用的解题方法如下。我们记录哪些公司控制哪些公司，且当每次我们得知某公司控制了某公司百分之多少的股份，就更新我们的信息。

数组“owns”记录了 i 公司拥有 j 公司的多少股份，包括直接控制以及间接控制。数组"controls"记录哪些公司被哪些公司控制。（官方源码见参考代码->C）

DFS

con[i，j]记录 i 控制 j 的股份。对于每个公司 i，搜索 i 直接控股的所有公司，用 cx[j]记录公司 i 控制了公司 j 的股份（直接和间接），将股份加到 cx 上，如果超过 50 就递归搜索公司 j，每个公司至多只递归一次，所以用 vis 判重。最后搜索所有 cx 超过 50 的 j，那么 i 控制 j。速度还是比较快的。

Section 2.4

The Tamworth Two (ttwo)

求出连通片先，然后 Floyd-Warshell 算法求每个连通片的周长，顺便记录连通片内的顶点到其他顶点的最大距离。

事实上，任意加入一条连接两个连通片 C1，C2 的边 e 之后，顶点对分为三类：均在 C1 中，均在 C2 中，或一个在 C1 中一个在 C2 中。前两者的距离的最大值是 C1，C2 的周长，最后这类边均经过 e=(v1，v2)，设 v1 在 C1 中，v2 在 C2 中，则此类边的最大长度为 C1 中的顶点到 v1 的最大距离＋e 的长度＋C2 中的顶点到 v2 的最大距离。而这两个最大距离，在用 Floyd-Warshall 算法求周长时，就已经得到了，只需存下来即可。因此，判断加边之后的周长的可在 O(n^2)时间内完成。

参见 blackco3 的 C++代码。其中，求周长可以不区分连通片，直接在整个矩阵上求解，编码上简单一些。

Overfencing (maze1)

我们用一个数组记录每个格子四面连通情况，然后从输入文件中读入每条边（判断），维护这个数组。然后分别从两个出口做 Flood fill，记录每个格子的最短距离。所有格子的最短距离的最大值为所求。

可以将整个图用 2*w+1，2*h+1 的布尔数组表示，门口处记为一步，最后统计所有偶数行列，输出 max div 2 源码。还可以把两个出口分别作为起点，分别将整个地图走两遍，此时求出两次每个点到两个出口的最短距离中短的一个，再找出最短距离最长的一个点。

小提示：使用广搜实现 Flood Fill 更快一点

补充：可将地图看为一个图，有一个额外的点表示出口，记可以到出口的一个或两个点

到出口距离为 1，其余的相邻联通点之间距离记为 1(用 bool 数组即可)，然后用堆优化的 Dijkstra 算法，只用一遍，就可求出各点到出口的最短距离，找出其中最长的即可，效率还不错。

Cow Tours (cowtour)

没错误情况的解法

求出连通片先，然后 Floyd-Warshell 算法求每个连通片的周长，顺便记录连通片内的顶点到其他顶点的最大距离。

事实上，任意加入一条连接两个连通片 C1，C2 的边 e 之后，顶点对分为三类：均在 C1 中，均在 C2 中，或一个在 C1 中一个在 C2 中。前两者的距离的最大值是 C1，C2 的周长，最后这类边均经过 e=(v1，v2)，设 v1 在 C1 中，v2 在 C2 中，则此类边的最大长度为 C1 中的顶点到 v1 的最大距离＋e 的长度＋C2 中的顶点到 v2 的最大距离。而这两个最大距离，在用 Floyd-Warshall 算法求周长时，就已经得到了，只需存下来即可。因此，判断加边之后的周长的可在 O(n^2)时间内完成。

参见 blackco3 的 C++代码。其中，求周长可以不区分连通片，直接在整个矩阵上求解，编码上简单一些。

分析

用 Floyd 求出任两点间的最短路，然后求出每个点到所有可达的点的最大距离，记做 mdis[i]。（Floyd 算法）

r1=max(mdis[i])

然后枚举不连通的两点 i,j，把他们连通，则新的直径是 mdis[i]+mdis[j]+(i，j)间的距离。

r2=min(mdis[i]+mdis[j]+dis[i，j])

re=max(r1，r2)

re 就是所求。

注意：上面的算法并不完善！[需要证实]

一组简单的数据

5

0 0

0 1

1 0

100 100

200 200

00000

00100

01000

00001

00010

正确的答案应该是 2.414214，而上面的算法得到的却是 141.421356。造成这结果的原因是：

每次尝试连接并比较，得到的新的牧场的直径应该是下面三个值的最大值：

1.节点 i、j 连接后，得到的新通路的长度(half[i]+half[j]+sqrt(sqr(x[i]-x[j])+sqr(y[i]-y[j]))；

2.原节点 i 所在牧场的直径 dia[i]；

3.原节点 j 所在牧场的直径 dia[j]。

其中 2、3 应先与 1 比较得出最大值，在与目前最优答案比较得出最小值。即 ans={min{max{dia[i]，dia[j]，half[i]+half[j]+dis(i，j)}}，其中 i，j 属于 V，(i，j)不属于 E}

Usaco 的数据一般比较强，这次也出现了疏忽。这一题数据较弱，我大小于号打反，竟然过了 6 个点。

Bessie Come Home (comehome)

这道题的规模很小（52），所以可以用 Floyd-Warshall 算法求出所有点对的最短路，然后找出其中到'Z'距离最短的点就可以了。

当然本题也可以用 Dijkstra 来做，会比 Floyd 快一些，不过编程复杂度要高一些。

思路二：借鉴广度优先搜索的思想。code 已给出。

建立图的邻接表存储，每次访问展节点 c 时，对其相连的顶点 v 进行检查，若有 dis['Z', c]+dis[c，v]<dis['Z'，v] 则 dis['Z'，v]:=dis['Z'，c]+dis[c，v]; 如果节点 v 未访问 则进队。

只需进行一次宽度优先搜索即可。

值得注意的是，由于有 10000 条边，而总共只有 52 个节点，也就是说，输入中的许多边都是没有用的，所以读入的时候应该判断一下这条边已有的权值是否比读入的权值要小，如果小的话，再赋值。

Fractions to Decimals (fracdec)

记得长除法吗？我们知道只有当出现了曾经出现过的余数时，小数部分才会出现重复。重复的部分就是自从我们上次见到同样的余数之后计算出的部分。

我们先读入并打印整数部分。接下来，我们对剩下的真分数部分进行长除直到我们发现了重复的余数或余数变为 0。如果我们发现了重复的余数，即出现了循环节，就分别恰当地打印重复的部分和不重复的部分。如果余数变为 0，即已经除尽，就打印整个小数部分。如果小数位根本没有被生成，那么打印一个 0 就是正确答案了。(源码见：参考代码-> C/C++ -> 官方源码 1)

另一个解法

计算循环开始前的小数位数，这样你甚至无需保存各个小数位和余数，程序的空间花费将大幅减小，而运行速度也能有所提高。我们知道 2 和 5 的幂是仅有的两种不导致循环的数，因此要找到循环前的各数位，我们只需分别找到分子分母中包含的因子 2 和 5 个数的差，再取两者的最大值（详见代码片段）。然后我们仅使用第一个余数，在计算时输出各个数位即可（此方法源码见：参考代码-> C/C++ -> 官方源码 2）（PASCAL 见 lsylsy21）。

我们知道，a/b 的每位运算所得的余数只可能是 0..b-1，如果在某处出现一个余数之前曾经出现过（在小数位上），那么可以肯定此时从该处到上次用出现这个这个商之间存在循环节。用 m 记录每种余数是否曾经出现过，ss 记录余数第一次出现的位置，如果余数为 0 就是整除，否则就找到循环节，输出。

实际上，后面的小数内容完全取决于试除这一位时的商和余数。另外，注意输出格式。

Chapter3

Section 3.1

Agri-Net (agrinet)

一个非常标准的最小生成树。

Score Inflation (inflate)

背包（题解很乱）

Humble Numbers (humble)

我们在数组 hum 中计算出前 n 个丑数。为了实现起来更简单，我们把 1 也作为一个丑数，算法也要因此略微调整一下。

当我们已知前 k 个丑数，想得到第 k+1 个，我们可以这样做：对于每个质数 p，寻找最

小的丑数 h，使得 h * p 比上一个丑数大。取我们找到的 h * p 中最小的一个，它就是下一个丑数。

为了使搜索更快，我们可以为每个质数维护一个索引“pindex”表示每个质数已经乘到了哪个丑数，每次都从那里开始，而不是再从头再来。(官方源码参见参考代码-C)

方法二

这道题可以用 BFS+Treap 来做。但这里的 BFS 不使用队列来扩展，而是用 Treap 来扩展。

建一个 Treap 保存已经得到的数，从小到大每次从堆中取出一个数，用它和集合中的质数相乘，查找判断它是否重复.如果没有重复，那么将它插入到 Treap 中。直到产生了 n 个数，那么再往后扩展一位，得到的第 n 个数既为所求的结果。时间复杂度：BFS 扩展为 O(N)，查第 k 大为 O(logN)判重为 O(logN)，插入为 O(logN)，因此总的复杂度为 O(NlogN)

我的思路是每次从堆中取一个，然后从质数集合里扫一遍。虽然不能保证新扩展出的数的顺序，但很类似 Astar，第 n 个开始扩展的节点可以确定，复杂度应该为 nklogn，可惜内存不够开…

方法三

这道题目可以直接应用 STL 中的 priority_queue 来做，复杂度为 O(NKlogN)，对于第 i 小的数，扩充它能得到的所有新元素并插入 priority_queue 中，然后 pop 第 i 小的数。时间没问题，但是空间会爆，当然只对于最后一个数据，我最后一个数据在自己机子上 RUN 完后，骗过去的，USACO 上分配内存太小!

假想中的方法：

读入数据数组 a，然后对二取对数，存为数组 b，然后把问题转化成装箱问题（求第 n 种 b 能组出的和，找出第 n 种），貌似需要离散化、位运算，我刚学，不懂，刚才试了一下，发现(ln(maxlongint)-ln(maxlongint-1))*maxlongint<1，也就是说当数达到一定程度的时候就无法精确到一了，有什么好办法？

Shaping Regions (rect1)

离散化先。然后，按行灌水（开矩阵灌水不行，空间限制通不过）。关键在于程序要有所优化，完全用指针操作数组，不要用下标引用。

Contact (contact)

1 <= A <= B <= 12，由于信号长度不同需要区别（0 与 00 不是同一信号），可以转化为三进制储存，也可以用二维数组 F[i，j]表示值为 i 位数为 j 的信号出现次数。最后看储存方法不同按频率由高到低，由短到长，由小到大输出。

也可直接转成二进制处理，只需要每种字符串加一个前导'1'。具体看代码

[补充]--1 补充一个比较“旁门左道”的方法，可以用一棵满二叉树储存，一维数组实现。每个节点值为 0 或 1，左儿子是值为 0 的节点，右儿子是值为 1 的节点。从根节点（根节点本身无值）到该节点路径上经过的节点组成的串就是该节点表示的串。数组中下标 1 shl a 到 1 shl (b+1)-1 的节点就是位数为从 a 到 b 的节点。具体实现见代码 4。

[补充]--2 最好边读入边处理，否则是存不下那么大的数据的。

[补充]--3 应该可以用 ansistring 存下的样子啊，但最好别用 ansistring 存，本来时间空间就有些紧张。

[补充]--4 先统计长度为 B 的各种串的个数，然后推算 A~B-1 长度的串的个数。排序用选择排序就可以了。

提供一种高效的方法。首先把问题分成几部分，先生成数据。初始化 f[i，j]表示长度为 i 的第几元素，f[1，1]:='0'; f[1，2]:='1'; i:=1; while ss[i]='1' do begin ss[i]:='0'; inc(i); end; ss[i]:='1'; inc(shu); f[a，shu]:=ss; 用上述方法可巧妙解决 a 的所有元素，接下来 a+1 到 b 的都可由 1

和 a 推导；然后是枚举数据的位置 i，由 i+a-1 到 i+b-1 枚举可能的字符，计算其十进制的数值 l，l+1 即是其位置 l:=0; k:=0; for j:=i to i+a-2 do//先算开头是 i，长度为 i+a-2 的字串的数值。

Stamps (stamps)

一个简单的 dp 题目。我们开设数组 minstamps。minstamps[i]表示凑成 i 分游资所需要的最少邮票数。对于每一种邮票，我们试图在每一分我们已经贴出的邮资上再贴一张这种邮票。在我们找到贴出贴出任何钱数所需要的最少邮票数，我们就寻找不能用所给的邮票数所能贴出的最少邮资。输出比它小 1 的数。参考代码(见 c 部分) 。

动态规划的题目，用 F[i]表示得到 i 面值所需要的最少邮票个数，Value[j](j=1..Stamps)表示每个邮票的面值，可得以下转移方程：

F[i]=Min (F[i-Value[j]] + 1) (i-Value[j]>=0 j=1..Stamps)

初始状态：F[0]=0;F[i]=INFINITE;如果计算中间某个 F[i]大于了最大可以使用的邮票数量，则退出循环，输出 i-1 即为解。

小优化：初始化时先计算出最大面值的邮票，令其面值为 MaxValue，然后在初始化 F 时可令：F[k*MaxValue]=k；这表示要得到 k*MaxValue 面值的邮票最少需要 k 张邮票，这是显然的。于是在计算时我们可以免去(1/MaxValue)*100%的运算量。

P.S.如果用官方算法，并且使用 pascal，会有一个超时 5%的点所以 pascal 必须想法优化。

只要在背包的时候注意放入物品的顺序重复时不重复计算如（1，3）和（3，1）都可以得到 4，注意不重复所有数据都能在 0.2s 以下过掉。见代码 1。

Section 3.2

Factorials (fact4)

简单的把末尾的 0 去掉是不行的，因为我们不知道现在不是 0 的位会不会乘上下一个数之后就变成 0 了。

因为 10=2*5，所以每有一个 0 就有一对 2*5=10 出现，反之，如果这个数的质因数分解没有成对的 2，5，我们就可以简单的对 10 求模，而不用管前面的数字，因为它一定不会产生 0。

所以我们只要在处理阶乘的时候消掉所有成对的 2 和 5 就行了，容易理解，N!的质因数分解式里因子 2 远比 5 要多，所以只需要记录因子 2 的个数，有因子 5 就消掉，最后再把 2 乘回去就行了。

可以直接用高精度乘法计算，5000 位就够了，每位存储一个四位数，打印时处理一下。

甚至可以连高精度都不用，用一个变量 j 记录 i!的最后四位末尾非零数，在计算(i+1)!时只需要计算(i+1)*j 即可。

更简洁的方法是，一遍循环消除 5 并记录 5 的个数，第二遍循环消除同样个数的 2 并计算末位。

关键一：消除 5 的同时记录个数，然后按照个数消除 2.

关键二：仅记录最后一位即可。

直接高精度，每位就储存一个一位数，当位数超过 1500 位时就把 1500 位以后的数去掉（阶乘乘出 1000 多个 0 好像不太现实），最后从各位往前查找不是 0 的数。可以 AC。

只记录最后的非 0 的 5 位数，不停地乘就 ok，极其简单，不涉及任何高精度之类的知识。详情见代码。

Stringsobits (kimbits)

官方题解

假设我们知道如何计算含有给定个数的二进制 0 和 1 的集合大小。也就是说，假设我们

有一个函数 sizeofset(n，m)返回含有至多 m 个 1 的 n 位二进制数的集合大小。

然后我们可以如下解决这个问题，我们在寻找至多 m 个 1 的 n 位二进制数的集合的第 i 个元素。这个集合有两部分，开始于 0 和 1 的。有 sizeofset(n-1， m)个数以 0 开始且至多有 m 个'1'，有(n-1，m-1)个数以 1 开始并且至少有 m-1 个 1。

所以如果这个序号(i)比 sizeofset(n-1，m)小，题目所求的数出现在开始于 0 的那部分，不然的话，它以一个'1'开始。

由"printbits"实现的递归算法很合适

所剩的唯一困难是计算"sizeofset"，我们可以用上面描述的特征使用动态规划解决这个问题。 sizeofset(n， m) = sizeofset(n-1， m) + sizeofset(n-1， m-1)

并且对于所有 m，sizeofset(0， m) = 1.我们使用 double 来存储大小，但这矫枉过正了，因为重写的问题只要求 31 位而不是 32 位。

动态规划

设长度为 j 的 01 串，1 的个数不大于 k 的个数为 f[j，k]!

方程：f[j，k]=f[j-1，k]+f[j-1，k-1]; //分别表示在当前位加上 0 和加上 1 时的两种状况

边界：f[j，0]=1，f[0，j]=1;f[j，k](k>j)=f[j，j]

这样我们得到了所有的 f[j，k](j，k∈Z， j，k>0)，需要做的就是据此构造出所求字符串。

设所求串为 S，假设 S 的位中最高位的 1 在自右向左第 K+1 位，那么必然满足 F[K，L]< I，F[K+1，L] >=I，这样的 K 是唯一的。所以 S 的第一个 1 在从右至左第 K+1 位.因为有 F[K，L]个串第 K+1 位上为 0，所以所求的第 I 个数的后 K 位就应该是满足"位数为 K 且串中 1 不超过 L-1 个"这个条件的第 I-F[K，L]个数。

于是我们得到这样的算法:

```
for K:=n-1 downto 0 do {题目保证有解，所以 f[N，L]>I}
 if F[K，L]<I then
 begin
    s[N-K]:='1';
    dec(I，F[K，L]);{第 K+1 位是 1，所以 I 减去第 K+1 位是 0 的串的个数}
    dec(L);
 end;
```

优化

其实从基本的组合原理出发，我们可以推出 F[j，k]的表达式:F[j，k]=C[j，0]+C[j，1]+C[j，2]+...+C[j，k].这里 C[i，j]表示从 i 个元素中取 j 的组合数因为 C[m，n]=C[m-1，n-1]+C[m-1，n]，可得

F[j，k]=C[j-1，0]+C[j-1，0]+C[j-1，1]+...+C[j-1，K]+C[j-1，k-1]+C[j-1，K]

=2*(C[j-1，0]+C[j-1，1]+...+C[j-1，k-1])+C[j-1，K]

=2*F[j-1，k-1]+C[j-1，k]

这样我们就可以用一个递归程序求解该问题.

用，我们首先可以仅用 O(N)的时间就可以计算出 F(j，L)的值，并记录 C(j，L).若需添"1"，则可由上式计算出 F(J-1，L-1)的值，并用 O(1)的时间将 C(J，L)改进为 C(J-1，L-1);若只需添"0"，则可先计算出 C(J-1，L)的值，再将 F(J，L)改进为 F(J-1，L)，也是 O(1)的时间. 因此我们只需要保留 C，F，N，M，K，I 等少量临时变量，空间复杂度为常数级，且可以在 O(N)的时间内计算出解，非常优化.

(还有 C[I，J]与 C[I-1，J]，C[I，J-1]之间的 O(1)改进方法，参见程序 3，自己推也很容易)

Spinning Wheels (spin)

在 360 秒后，所有轮子都回到原处，所以只需模拟 0~359。

对这个题官方数据的边界处理有点质疑请大家看看我的 c/c++代码 （c++，talenth1）

官方数据对[0，1][1，2]这种情形也认为是可以透光的，需要注意。

Feed Ratios (ratios)

数学方法。其实就是求解一个线性方程组，这里用向量的语言叙述了。

所求的实际上就是一个最简配比 x，y，z，k，使得 x(a1，b1，c1)+y(a2，b2，c2)+z(a3，b3，c3)=k(g1，g2，g3) 我们定义 有 我们用行列式求解这个问题，假设所得系数行列式分别为 D1，D2，D3，D，则 D=0(要么无解，要么多解，题目说没有任两个向量的线性组合是第三个，所以不成立)或是解不同号的情况则无解，要不然(D1，D2，D3，D)就是一组(x，y，z，k)，只需要用 gcd 约减下公约数就可以了。O(1)。

克莱姆法则，判断是否是分数解或者负解。枚举一下最后一个输出数的倍数即可。

基于分数表示的高斯消元

虽然比较复杂，某人大概写了 170 行，但几乎不需要用到什么特殊的理论知识，时间复杂度 O(n^3).

枚举

只用简单的穷举也可以，只有 100^3=1000000，约 0.3~0.4s 注意 0 的情况！

枚举 i=0..99.j=0..99，k=0..99。i，j，k 分别为三种饲料的份数。当三种饲料恰能构成目标饲料时，记录，保留最小值。很简单的算法，运行也很快。

Magic Squares (msquare)

宽搜，状态数 8! 不大。判重用 hash（如果 RP 较好你可以不用），用康托展开，或者使用 8 位 16 进制展开（相当于 32 位 2 进制），用 longword 或 qword 就可以表示。怎么开散列就随便了。

把 8 表示成 0，每 3 个二进制位可以存储一个数字，总共 2^24 种，考虑到每 1~8 都会出现，知道前 7 个数时最后个数也已经被确定，只需要将用来表示状态的二进制右移 3 位，达到 2^21，空间就没问题了。

因为 8 个数是确定的，其实用一个七维数组就可以判重了，flag[1..8，1..8，1..8，1..8，1..8，1..8，1..8] of boolean。

如上使用数组判重是个好方法，用 hash 太复杂啦！不过注意，如果是 C 的话，要用 short int，否则会超过 usaco 内存限制。

宽搜+全排列编码判重空间复杂度 40320*（8+2+1）。

生成哈希：1、只用七位就够了，因为第八位是确定的。 2、由于是不重复的，可以生产 8 的全排列，再 new(f[.....]); 这样总共只用了 7!=5040 的空间。 具体见 pascal 代码 5。

简单的 bfs+hash，hash 时记录 7 位，用八进制存，内存足够，不需要另外 hash 了。注意初始状态 12345678 的情况。

如果没限制，c++的用 map 可以，但是学 oi 的就不要想了。（搞 acm 可以用这个偷一下懒，但还是建议用康托，或位进制 hash）

Sweet Butter (butter)

此题的实际模型如下：给定一个有 P 个顶点的无向图，选择一个顶点，使得从该点到给定的 N 个顶点的权值之和最小。

Hint：一定要记得用邻接表存图

Bellman-Ford 算法

数据规模中边数小于 1450，使用 Bellman-Ford 求一次单源最短路的复杂度为 O(e*k)，k 为一个小常数，总复杂度也就是 O(n*e*k)。

d 数组储存最短路长，Bellman 算法的思想就是对每条边进行松弛，因为一共 n 个点，所以这种松弛操作最多执行 n 次后，d 数组的值就不会再改变。

还需要用一个布尔变量记录这次操作中是否产生了更小的最短路，如果没有，说明 d 数组中值已为最优，不需要再操作，直接退出。这样做或许时间上还是有点问题，便可以用 SPFA 优化。

SPFA 就是指每次松弛操作时记录哪些点更新了最短路值，将这些点加入数组，下次只对这些点关联的边进行松弛。进一步降低了复杂度。实现 SPFA 需要用邻接表储存边。

最后统计一次带权路径长，进行 n 次 Bellman-Ford 操作后得出结果。运行时间都在 0.4 秒以内。

Dijkstra 算法

求每对顶点之间的最短路径首先想到的便是 Floyd 算法:但 P 的范围是（P<=800）则时间复杂度为 O(800^3)，显然会超时。此时可以考虑使用 Dijkstra 算法求最短路径（Dijkstra 是 O(N^2)的算法），但直接使用 Dijkstra 仍然会超时 O(800*800^2)=O(800*^3)，此时可以用堆进行优化。

Dijkstra 算法每次都需要在 Dist[]数组中寻求一个最小值，如果图很大，则在此处花费的时间会很多，而堆则可以在 O(1)的时间内马上求得最小值，并且维护一个堆所需的时间都是在 log 级的。因此考虑把 Dist[]数组的值用堆来存储。

为了记录堆中的每一个元素是源点与哪个顶点的，可以增加了一个 point 变量，用来记录这条边所连接的另一个顶点。

用堆操作的 Dijkstra 的思想还是与一般的思想类似(在 OIBH 上找了一些信息)：

初始化 Dist[]=MAX，然后将源点 t 放入堆中(这就与 Dist[t]=0 类似)，再从堆中找出一个最小值没有被确定的点 minp(就是 Dist[minp]=MAX)，将其确定下来(Dist[minp]=mins，mins 为从堆中取出来的那个最小值)，接着由这个最小值所连接的点求出从源点到这些点的路径长，若所连接的点没有求出最小值(Dist[i]=MAX)，则将这个点放入堆中(这就好比用 for 循环去修改每一个 Dist[]的值，不同的地方在于堆中存放的是源点到各个顶点的最小值，如果刚求出来的顶点在 Dist[]中已经被赋值了，说明求出来的肯定不是最小值，就像普通 Dijkstra 的 Mark[]一样，mark 过的点是不能再被计算的，所以不能放 Dist[]中有值的点。)这样 Dijkstra 的部分就完成了。

仔细观察了一下 N<=800，而道路数 C<=1450。800*800=640000 与 1450 的差距实在是大……可以改用存边的方式(同时保留邻接矩阵，这样可以很方便地知道两点之间的权值)，用一个二维的数组便可以存下边的信息。

补充：用 STL 中的 priority_queue 优化 dijkstra 编程复杂度更小！

SPFA 算法

对于枚举的每个牧场 i，用 SPFA 求出到每个点的最短路如下：dist[j]表示 i->j 的距离，初始值为 maxint，其中 dist[i]=0。维护一个队列，初始为 q[1]=i;由队首枚举其他的牧场 j 更新牧场 i 到 j 的最短距离并同时拓展队列。直到队列空为止。这样就求出了点 i 到所有点的最短距离。

SPFA 的优化：

如果枚举到的牧场 i，那么以表明 1--i-1 牧场为源点的最短路都已经求出过，也就是说 i 牧场到 1--i-1 中任一牧场最短路已知，用这个已知条件来初始化 dist[1]--dist[i-1]这样效率会有很大提升。注意：如果 min[j，k]表示 j 到 k 的最短路，在以 k 点为源点并进行上述优化时，应将 dist[k]:=min[j，k]+1，而不是 min[j，k]。否则 j 点不会进入队列，也就无法拓展经过 j 点的路径。

Section 3.3

Riding The Fences (fence)

分析

这道题是要求我们求出一条欧拉路，所以我们要首先判断图中是否有欧拉路。对于一个无向图，如果它每个点的度都是偶数，那么它存在一条欧拉回路；如果有且仅有 2 个点的度为奇数，那么它存在一条欧拉路；如果超过 2 个点的度为奇数，那么它就不存在欧拉路了。

由于题目中说数据保证至少有 1 个解，所以一定存在欧拉路了。但是我们还要选一个点作为起点。如果没有点的度为奇数，那么任何一个点都能做起点。如果有 2 个奇点，那么就只能也这两个点之一为起点，另一个为终点。但是我们要注意，题目要求我们输出的是进行进制转换之后最小的（也就是输出第一个数较小的，如果还有多组解，输出第二个数较小的，等等），所以我们要以最小的点做起点。

找出欧拉路的方法就是采用深搜的方式，对于当前的点，把所有点从小到大的搜索，找到和它相连的，找到一个之后删除它们之间的连线，并去搜索新的那个点，如果没有找到点和它相连，那么就把这个点加入输出队列。

不过我们这么操作之后，顺序是反着的，输出时反着输出即可。

Shopping Offers (shopping)

0 <= b <= 5，1 <= k <= 5，可用 5*5*5*5*5 的 DP 每种买 0~5 个，可以用 6 进制表示，然后 5 维 DP~OK!

状态设置：F[a1][a2][a3][a4][a5]为买 a1 件物品 1，a2 件物品 2，a3 件物品 3，a4 件物品 4，a5 件物品 5 时，所需的最少价格。

边界条件：F[0][0][0][0][0]=0。

状态转移方程：

F[a1][a2][a3][a4][a5]=min{F[a1-P[i][1]][a2-P[i][2]][a3-P[i][3]][a4-P[i][4]][a5-P[i][5]]+P[i][0]}

其中 i=1..s+b; 且 ak-p[i][k]>=0。

其实这道题用 dfs 加一些优化也可以过。

用一个五维数组纪录每个已搜索过的状态和该状态所需要的钱数，如果搜索出了有重复状态并且所需要的钱数更多就剪枝。但是单纯这种方法最多只能通过第 9 个点。因为这种方法对数据的顺序有要求，所以可以在读入的时候将数据随机化排列，我用的倒序排列。

Camelot (camelot)

本题就是一个求所有点对间的最短路然后处理的问题。最短路好求，难点在于有王，王可以自己一格一格地走到汇集点；

也可以让某个骑士先跳到他所在的格子，再一起到汇集点；也可以先走几步，再让骑士跳到他所在的格子，再一起到汇集点。

d[i，j，x，y]表示点(i，j)到(x，y)的最短路长，可以用 BFS 求出。王的坐标为(kx，ky)，最终结果用 ans 储存。

基本思路是枚举一每个格子作为所有骑士的汇集点(i，j)，再枚举每个格子，以这个格子为王和某个骑士的相遇处(x，y)，再枚举每个骑士坐标(m，n)。

求出 d[i，j，x，y]+d[x，y，m，n]+max(abs(kx-x)，abs(ky-y))-d[i，j，m，n]的最小值，将汇集点到所有骑士的最短路和 s 加上这个值，与 ans 比较取小者。

枚举的时间复杂度为 O(n^3)，超时不是一般地严重……于是开始优化。最优化剪枝是必要的，当某个汇集点到所有骑士的最短路 s 已经大与了 ans 时直接退出。

尽管效率大大提高，但仍然严重超时。于是想一想，王和骑士的相遇点可能出现在哪些位置上？显然王走的比骑士要慢，那么应该尽量要让王少走，所以王需要先走的情况只可能是骑士无法到达的地方或者骑士到达需要绕一圈的情况。

可以构想一下骑士在王附近时达到王的路径，就不难发现，相遇点只应该在王的附近很短的距离以内。估算一下，相遇点就在王的坐标加减 2 的范围内枚举就可以了，经过证明，这样做是正确的（好象有人已经在 OIBH 上发表了证明方法）。

加上这两个优化，就可以比较快地 AC 了。

Home on the Range (range)

这道题可以动态规划。二维的动态规划。

状态定义：G[i][j]为以(i，j)为左上角顶点的正方形的最大边长。

边界条件：G[i][j]为初始读入的矩阵。

状态转移方程： G[i][j]=min{ G[i+1][j] ， G[i][j+1] ， G[i+1][j+1] } + 1;

解析： G[i+1][j] ， G[i][j+1] ， G[i+1][j+1]分别为(i，j)向下、向右、向右下一格的状况。在(n-1，n-1)当且仅当三者都为 1 的时候，正方形才能扩充。从最右下向上，依次扩充即可。

这道题也可以直接模拟计算以每个格为左上角顶点的最大正方形的边长: 对每一格，在当前已有的长度为 K-1 的正方形基础上，若右侧与下侧能同时加上长为 K 宽为 1 的矩形条，则得到长度为 K 的正方形.

在判断是否有完整的矩形条时，先预处理得到从左到右与从上到下的累加数组:

```
sum_horizon[i][j]=sum_horizon[i][j-1]+a[i][j];
sum_vertical[i][j]=sum_vertical[i-1][j]+a[i][j];
```

并利用等式判断:

```
sum_horizon[i+k-1][j+k-1]-sum_horizon[i+k-1][j-1]==K &&
sum_vertical[i+k-1][j+k-1]-sum_vertical[i-1][j+k-1]==K
```

算法复杂度 O(N3). --[[User:jay23jack] 17:27 2008 年 7 月 27 日 (CST)

另一算法：

opt[i，j]表示以 i，j 为右下角，可构成的最大正方形的边长

方程为：opt[i，j]:=min(opt[i-1，j]，opt[i，j-1]，opt[i-1，j-1])+1；

再开一数组 F，用来统计边长相同的正方形的个数，因为边长小的与边长大的存在包含关系，所以最后将边长大的正方形的个数再累加到边长比它小的正方形内，输出即可。详见代码 6 。

O(N^2logN)的一个算法：

枚举左上角后二分确定最大边长。结果处理同上一算法。详细内容见 C++代码 2。

O（N^3）简单易想算法

dp[k][i][j]表示以 k 为边长左上角为（i，j）的正方形是否能构成。那么 dp[k][i][j] = dp[k-1][i][j] && dp[k-1][i][j+1] && dp[k-1][i+1][j] && maze[i+k][j+k] (maze[i][j]表示矩形点(i，j)的 bool 值)dp 的过程中统计即可。期中 3 维数组超内存降维或者用滚动数组。代码见 C++。

其实这个题可以类比最优子矩阵，只不过求的是所有的矩阵，用类似最优子矩阵的方法枚举所有子矩阵即可，详见 pascal 代码 7。

A Game (game1)

分析 1

博弈问题，可以用动态规划解决。设 sum[i][j]表示从 i 到 j 的所有数字和，dp[i][j]表示从 i 到 j 这部分的先取者能获得的最大数字和。

dp[i][j]=sum[i][j]-min(dp[i][j-1]，dp[i+1][j]);

以 i~j 的区域的长度作为阶段即可。详细见 C++代码(ybojan2)。

分析 2

博弈问题，可以使用动态规划求解。状态定义：用 F[i，j]表示第一个玩家先取时，在第 i 到第 j 的子序列中能拿到的最高分；用 S[i][j]表示第 i 到第 j 的子序列中所有数字的和；用 num[i]表示第 1 到第 n 的序列中第 i 个数。

边界条件：F[i][i]=num[i]

状态转移方程：

F[i][j]=max{num[i]+S[i+1][j]-F[i+1][j]，num[j]+S[i][j-1]-F[i][j-1]}

结果

p1=F[1][n]；

p2=S[1][n]-F[1][n]；

解析：

num[i]+S[i+1][j]-F[i+1][j]表示的是，p1 拿第 i 到第 j 最左边的数，然后轮到 p2 在第 i+1 到第 j 的序列中先取，会剩下 S[i+1][j]-F[i+1][j]，这些归 p1。

分析 3

令 F(start，stop) 表示当前游戏者面对编号从 start 到 stop 的数字时可以得到的最大收益，Sum(i，j)表示编号 i 到 j 的数字和。可知：

F(start， stop)=Max(Sum(start+1， stop)-F(start+1， stop)+num[start] ， Sum(start，stop-1)-F(start，stop-1)+num[stop])；

边界条件 F(start，start)=num[start]；

F(1，N) Sum(1，N)-F(1，N) 即为所求解。

分析 4

当你在 i 到 j 中取了 i 之后，那么你就只能在 i + 1 到 j 中作为后手取值，故你所能取得的值为 i + 1 到 j 的和减去作为先手所取得的最大值；

初始化 f[i][i] = sum[i][i] = a[i];

状态转移方程:

f[i][j] = max(a[i] + sum[i + 1][j] - f[i + 1][j]， a[j] + sum[i][j - 1] - f[i][j - 1]);

f[i][j] 表示在从第 i 个数到第 j 个数里先手能取得的最大值;

a[i] 表示第 i 个数的值;

sum[i][j] 表示在从第 i 个数到第 j 个数的和;

Section 3.4

Closed Fences (fence4)

几何问题

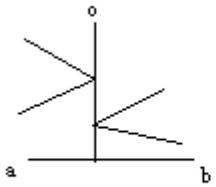
1.判断多边形是否合法

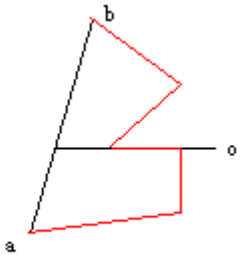
任两条边都不相交即合法，注意这里的相交是严格相交，顶点相交不算相交。

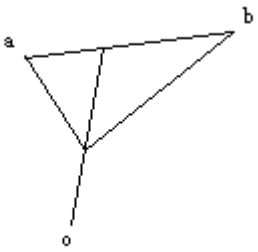
2.能看到哪些边

设人所在点为 o，从 o 分别到任意边的顶点和中点做射线，设为 op，然后求出与 op 相交的最近一条边，则这条边一定可以被看见。而对于每条边，它要么被过它中点的射线“看见”，要么被某条边遮住一部分，那么肯定会被过这条边的某个顶点“看见”。以上是不考虑特殊情况的粗略算法。

需要考虑的特殊情况：







1. o 到 ab 仅有一条边，此时 o 看不到 ab。

2. o 经过一条平行线到 ab，此时 o 看不到 ab。

3. o 经过一点到 ab，此时 o 看不到 ab。

针对以上情况，我们设 3 个变量 ml，mr，mm，ml 记录交点 s 到 o 的最近距离，且 s 的两条边在 op 的左侧，mr 与 ml 类似，s 的两条边在 op 的右侧，mm 记录横穿 op 的交点的最近距离。枚举每条边时，如果这条边横穿 op（严格相交），那么更新 mm，记录当前的边如果这条边与 op 交于一个顶点 s，那么： (1)如果 s 的两个相邻顶点在 op 的两侧（考虑情况 1），更新 mm，记录边 (2)如果 s 的两个相邻顶点在 op 的左侧（或一个在左侧，一个在 op 上，考虑情况 2），那么更新 mr (3)类似地更新 ml。

最后如果 ml 和 mr 都比 mm 近，那么属于情况 1，ab 看不见。这样就 ok 了。

提示

其实并不用判断多边形是否合法，测试数据根本没不合法情况。判断是否可见可用二分。

本题数据较奇怪。尽管二分法可以 ac，但能看到栅栏不一定能看到栅栏的两个端点之一。如下面数据： 6 3 0 2 -1 2 1 0 2 6 2 4 1 4 -1 答案应该是 4，因为为上面的栅栏可以看见，但是看不到两个端点。

尽管这样的情况没有出现，但数据里出现了只能看见很短的栅栏的数据（ Data9）。（data9，锯齿状）。

American Heritage (heritage)

输入二叉树的前序遍历、中序遍历，求后序遍历

对于这个题，完全可以扩展。如果我不仅仅是求出后续遍历，而要求出这棵二叉树，然后进行其他计算。那么就必须想办法把这棵树求出来。关键在于如何建立这棵二叉树。在 NOIP 初赛中，经常有这样的题目。手算不算难，写成程序实现就需要编程的基本功了。

采取递归的方法建立二叉树。首先取前序遍历的首元素当作二叉树的根，当前元素为根。把前序遍历中的当前元素当作中序遍历的分割点，中序遍历分割点前面的元素一定在当前元素的左子树，分割点后面元素一定在当前元素的右子树。然后加入下一个顶点，再把它当作分割点。如此递归的进行，直到二叉树建立完成。

Electric Fence (fence9)

可以算是一道数学题吧。如果知道皮克定理就行了。皮克定理说明了其面积 S 和内部格点数目 a、边上格点数目 b 的关系：S = a + b/2 - 1。根据三角形面积公式求出 S。如果知道了 b，那么三角形内部格点数目 a 也就求出来了。可以证明，一条直线((0，0)，(n，m))上的格点数等于 n 与 m 的最大公约数+1。即 b=gcd(n，m)+1. gcd(n，m)为 n 与 m 的最大公约数。代入皮克公式，即可求出 a 的值。

方法二：局部枚举结果等于(线段(0， 0)--(N， M)与线段(N， M)--(P， 0))与 X 轴之间的格点数，我们分别计算两条线段与 X 轴之间的格点数，然后根据 N 与 P 的大小关系进行加减计算即可！ 在计算线段与 X 轴之间的格点数时，我们枚举每一个 X 轴上的坐标，得到对应于该 X 坐标下的所有格点数。 具体代码请看 C++程序。

Raucous Rockers (rockers)

提示

不仅同光盘上的歌曲写入时间要按顺序，前一张光盘上的歌曲不能比后一张歌曲写入时间要晚。

分析-穷举

一道动态规划题，但观察数据规模，穷举就行了。穷举每首歌是否选取所有的组合可能(2^20 种)，算出每种情况所有光盘上一共能存的歌曲数目，保留最大值即可。

对于穷举每首歌是否选取所有的组合可能，采用位运算的高效方法。

limit=(1 << N)-1; for (i=0;i<=limit;i++)。

然后 i 对应的每种状况计算能装进光盘中的最大的歌曲数目即可。

分析-动态规划

此题用动态规划做。

A[g，k，r] 表示：在以第 g 首歌曲开头的序列中(不一定要取第 g 首歌曲)，还剩下 k 张 CD，且最后一张光盘上还剩下 r 分钟时间，能取到的最大曲目数。

A[g，k，r]=max{A[g+1，k，r+t[g]](r+t[g]<=l(每张盘的时间)，k<n)，A[g+1，k+1，t[g]]（t[g]<=l，k<n)，A[g+1，k，r]}(g<=m，k<=n) 。

A[g，k，r]=0(g>m)

一个平方复杂度的动态规划，dp[i][j] 记录 在前 i 个歌曲中选出 j 个所花费的时间（这里的时间是指 已用了几张 CD 和最后一张所剩的时间）

然后目标使每个 dp[i][j]最小，然后根据 j 以及 dp[i][j]记录的时间确定答案。复杂度平方。

分析-双重动态规划

用动态规划很容易的。

先利用三重循环计算出每个区间内的在背包重量内的最大值 （利用 01 背包)，再利用三重循环，求解

F[I，J]代表前 I 个歌曲装在 J 个唱片的最优值，所以

F[I，J]:=Max{F[K，J]+Maxit[K+1，I]}

输出 F[N，M]

Chapter4

Section 4.1

Beef McNuggets (nuggets)

问题分析

这是一个背包问题。一般使用动态规划求解。

一种具体的实现是：用一个线性表储存所有的节点是否可以相加得到的状态，然后每次可以通过一个可以相加得到的节点，通过加上一个输入的数求出新的可以相加得到的点。复杂度是 O(N×结果)。但是可以证明结果不会超过最大的两个数的最小公倍数（如果有的话）。

已知不定方程 ax + by = c(a,b>0 且 c>= ab)存在一组整数解(x0 ,y0)，求证，该不定方程存在一组非负整数解 (xn,yn)。证明：由不定方程通解式得：xn = x0 + b * t ，yn = y0 - a * t (t 是整数)令 xn，yn>= 0，解出-(x0/b)<=t<=(y0/a)。因为 c>=a*b，即 a*x0+b*y0>=a*b，两边同除 a * b，得：y0 / b - (- x0 / a) >= 1，所以一定存在整数 t 使得- (x0/b)<=t<= (y0/a)。所以方程一定有非负整数解。证毕。

所以复杂度也是 O(Na2)，完全可以接受了。

判断无限解可以按上面的方法，另外也可以算所有的数的最大公约数。如果不是 1，也就是说这些数不互质，那么一定没办法构成所有的不被这个最大公约数整除的数。当且仅当这种情况会有无限解。另外有一种不需要任何数论知识的方法是判断是不是按照每个输入的数的循环节循环，如果是的话，继续算显然不会有任何结果。

判断有没有更大的解也可以按这种方法，另外如果连续最小的数那么多个数都可以构成，也不会有更大的符合条件的解。

通过位压缩可以使程序在 32 位机上的运行速度快 32 倍（由于程序代码会变长，也可能是 16 倍或者更小的倍数）。这样可以相当快的解决这个问题。不过复杂度还是 O(Na2)。

MS 就是斐蜀定理。

Fence Rails (fence8)

问题分析

抽象一下就可以发现，算法的本质是多重背包问题。 补充：这题与破锣乐队都是多个背包，不可重复放物品。区别在于破锣乐队要有顺序，此题不需要，这样此题就必须要搜索才行。 单个背包的问题我们可以用 DP 解决，但是对于这种问题我们只能用搜索了。 但是可以看一看这道题的数据规模：1<=n<=50，1<=r<=1023。如此大的规模我们只能考虑进一步的优化。

采用 dfsid 搜索每一个 rail 来源的 board。以下技巧都是针对这种搜索顺序来制定的。(注：rail 是所需要切成的东西，board 是供应商提供的原料）

如果使用 dacing links 的话，可以让程序的常数快 2 倍。

优化技巧

很容易就能注意到，由于每块 rail 的价值是相等的——也就是说切小的要比切大的来的划算。那么我们在搜索能否切出 i 个 rail 的方案是自然要选最小的 i 个 rail 来切。

经过一些实验可以发现，先切大的 rail 比先切小的 rail 更容易提前出解。好像先切大的 board 要比先切小的更快。

由于 r 最大可能是 1023，但是 rail 长度的范围却只有 0~128，这点提醒了我们有很多 rail 的长度会是相同的。所以我们要避免冗余，优化搜索顺序。若有 rail[i+1]=rail[i]，则 rail[i+1]对应的 board 一定大于等于 rail[i]对应的 board。可以通过这种方法剪掉很多冗余的枝条。

相应的，如果 board[i]=board[i+1]，那么从 board[i]切下的最大的 rail 一定大于等于从 board[i+1]切下的最大的 rail。

对于切剩下的 board（无法再切下 rail），统计一下总和。如果这个值大于 board 长度的总和减去 rail 长度的总和，一定无解，可以剪枝。这个剪枝最关键。

二分答案

加上上述优化策略后，程序就能很快出解了。

Fence Loops (fence6)

做法一

求无向图中的最小环。算法很简单，就是做 m 遍 dijkstra——每次找到一条边，拿掉，求这条边两个顶点之间的最短路，那么如果这两点间存在最短路，则这条路径与原来的边就构成了一个环。这样所有环中最小的一个就是答案。问题是题目给出的是边连通的信息而非点连通。也就是说我们得到的信息无法按照常规的方法（邻接矩阵，邻接表）来构图。这里就需要一个转化。由于我想不到什么好的算法，所以就用了一个复杂度为 O(n^2)的转化。首先将每条边的两个顶点都看作单独的点（也就是说假设所有边都不连通，为了方便，可以分别设第 i 条边的两个顶点编号为 i*2-1 和 i*2），然后对于两条连通的边，将连通这两条边的点并在一起。具体做法就是将其中一个点的连通情况全部赋给另一个点，并修改图中其他与该点连通的点信息使得合并成立。这里我借助了并查集，使得每次查找的时间都近似为常

数，所以总的时间复杂度就是 O(n^2)。其中 n 是合并后总的点的个数经过上述转化以后，再用 m 遍 dijkstra，总的算法时间复杂度就是 O(mn^2)。

做法二

这道题给的数据是边之间的关系，首先想到的是构图，然后套经典的最短路做.但是这道题的数据输入会使按点构图很麻烦，其实这道题搜索即可，而且加剪枝后很快.由 v 条边开始搜索在它 b 方向的边，重复这个过程，直到回到 v 而且是由 a 方向回到的，为了判断环可以加个判重，一个剪枝:对于路径长度大于已知最小环的可剪。

做法三

一种比较诡异的方法。就是把边变成“点”。在把“边”的长度设定为边两端的“点”（既原来的边）的长度的和。

用由 floyd 算法改进的求最小环的算法求最小环。注意的是枚举“点”时，注意“点”连接的两“点”要是分别连接到该“点”代表的边的两个端点。

这个复杂度为 O(n^3)。超级快。证明略，有兴趣的自己想一下。

做法四

仍然是求最小环，先将边邻接矩阵转为点邻接矩阵，再 dfs 求出所有环。可以证明时间复杂度为 O(n^3)，所以能快速 AC。

Cryptcowgraphy (cryptcow)

问题分析

很考验人的一道搜索题。感觉比 fence rails 还要难。基本的思路很简单：按从小到大的顺序依次找出 C，O，W，然后交换中间的两个串并将这三个字符删掉。如此往复直到没有成对的 C，O，W，判断一下最后生成的字符串是否是题目所给的串。但是单纯的按上述算法只能通过 20%的数据。我们需要进一步优化。

优化技巧

由于添加的 COW 是一起的，因此给出的字符串的字符个数应该等于 47（目标字符串的长度）+3*k。如果不满足就可直接判断无解。

除了 COW 三个字符外，其他的字符的个数应该和目标串相一致。如果不一致也可直接判断无解。

搜索中间肯定会出现很多相同的情况，因此需要开一个 hash 来记录搜索到过哪些字符串，每搜索到一个字符串，就判重。如果重复直接剪枝。这里的字符串的 hash 函数可以采用 ELFhash，但由于 ELFhash 的数值太大，所以用函数值对一个大质数（我用的是 99991）取余，这样可以避免 hash 开得太大，同时又可以减少冲突。

对搜索到的字符串，设不包含 COW 的最长前缀为 n 前缀（同样也可以定义 n 后缀），那么如果 n 前缀不等于目标串的长度相同的前缀，那么当前字符串一定无解，剪枝。N 后缀也可采取相同的判断方法。

一个有解的字符串中，COW 三个字母最早出现的应该是 C，最后出现的应该是 W，如果不满足则剪枝。

当前字符串中任意两个相邻的 COW 字母中间所夹的字符串一定在目标串中出现过。如果不符合可立即剪枝。

需要优化搜索顺序。经过试验我们可以发现，O 的位置对于整个 COW 至关重要。可以说，O 的位置决定了整个串是否会有解。因此，我们在搜索时，应该先枚举 O 的位置，然后再枚举 C 和 W 的位置。其中 W 要倒序枚举。这样比依次枚举 COW 至少要快 20~30 倍。

在判断当前串的子串是否包含在目标串中的时候，可以先做一个预处理：记录每一个字母曾经出现过的位置，然后可以直接枚举子串的第一个字母的位置。这样比用 pos 要快 2 倍左右。

经过上述优化，程序对于极端数据也可以在 1s 以内出解。其实只用 3、4、6、7 就可以进 1s。事实证明优化 6 相当重要！

Section 4.2

Drainage Ditches (ditch)

明显的最大流模型。用最普通的 BFS 找增广路都可以过。但是这个图可能不是简单图，也就是说两点之间可能存在不止一条边，而且边的容量还不一定相同。这样对于邻接矩阵来说就有点问题了。解决方法也很简单：增加一个点。如果在读入的时候，给出的两个点 x，y（边容量为 r）之间已经存在一条边了，我们就增加一个点 z，在 x，z 和 z，y 之间分别连一条边，并且容量均为 r。这样做的效果就是：不影响结果，而且避免了很麻烦的计算。

还有更简单的方法，就是读入的时候不是采取直接赋值，而是采取累加的方法。这样所有重复的边就等于是一条边，这条边的容量等于先前所有重复边的容量和。原来的方法更适用于比较复杂的处理，比如涉及到具体的边的时候。

最大流的算法也有很多，最原始的 BFS，最短增广路算法，一般预流推进算法，以及更好的预流推进。我用的是 highest relabel-to-front，效率相当高，所有数据全部是 0s。

补充一点，题目中说要注意雨水环形流动的情形，这个其实是不必纳入考虑的，因为在环中，出的还是等于进的，容量守恒，所以整个点容量守恒。在 Preflow-Push 中，这条 loop 不会有值，因为点自己和自己的高度相同。

The Perfect Stall (stall4)

二分图最大匹配的模型。可以用网络流，但是推荐用匈牙利算法，代码很短，而且效率很高。二分匹配用网络流做就没必要用 Dijkstra 了，难写又超时。最简单的搜出一条路就行了。

Job Processing (job)

第一问直接贪心就行了。第二问可以直接按照对称的方法得到解。即 a[i]对应 b[n+1-i]，证明略。提示：利用单调性。

我们令 delay[A][i]表示第 i 个 A 型机器的延时，显然，初始时所有 delay[A]为 0。然后对于所有的工件，我们选取 delay[A][i]+machine[A][i](machine[A][i]表示第 i 个 A 型机器的加工用时）最小，也就是能在最快时间内完工的机器加工。更新 delay[A][i]=delay[A][i]+machine[A][i]（这样这个机器就进入新的一轮加工中）。在此过程中，我们用 cost[A][j]记录 A 操作加工出第 j 个零件的用时。用同样的方法求出 delay[B][i]，cost[B][j]。对于 A 操作的最短用时，显然就是 cost[A]中的最大值。

因为我们在求解过程中是一个工件一个工件地送去加工，每次把一个工件送给用时最短的机器加工，那么显然有 cost[A][1]<=cost[A][2]...<=cost[A][n]，同样适用于 cost[B]。为了使 B 操作的用时最少，我们应该把 A 先加工出来的工件给 B 中加工最久的，即 cost[A][1]->cost[B][n]，cost[A][2]->cost[B][n-1]...cost[A][i]->cost[B][n-i+1]，那么 max(cost[A][i]+cost[B][n-i+1]) i=1，2，3，4...n 就是 B 操作的最短用时。

要是 M1，M2 很大，可以用堆去维护。

Cowcycles (cowcycle)

问题分析

搜索。题意不难理解，注释写得很清楚了。数据不是很 BT，加些优化基本上就过了。最大的数据不会出，放心做。按照从小到大的顺序扩展。同样算法用 C++比 Pascal 快得多。

优化技巧

最大传动比率至少是最小的 3 倍。这个其实不用算出比率在判断，只要不满足 s1[F]/s2[1]-s1[1]/s2[R]>=3 的都剪掉 (s1 表示前齿轮型号，s2 表示后齿轮型号)。另外乘法计

算要比除法快得多，上面判断式可以写成 s1[F]*s2[R]>=3*s1[1]*s2 [1]。这个剪枝很有效果。

改变求方差的方法。

当找到比当前更优秀的解的时候，不要用 For 循环一个一个复制当前解，用 memcpy 函数会更快。

最想不到的地方，排序方法！开始我写成快排，一直过不了。其实数据的数量太少的时候，用快排效率很低的。在这里最适合的是简单插入排序。这个剪枝的效率很惊人！

就这些优化就可以通过了，求方差可以用期望来求。

Section 4.3

Buy Low,Buy Lower (buylow)

问题分析

动态规划题，就是最长下降序列问题。第一问可以用 O(N^2)的算法解决。

s[i]为序列中第 i 项的值，MaxLength[i]为以第 i 项为末尾中最长下降序列长度。

状态转移方程为 MaxLength[i]=max{MaxLength[j]}+1 (j=1..i-1 and s[j]>s[i])

初始条件：MaxLength[1]=1

对于第二问求最长下降序列的数量，可以通过求第一问的过程解决。设 MaxCnt[i]为第 i 项为末尾中最长下降序列的个数。

对于所有的 j(1≤j≤i-1)如果有(s[j]>s[i] 并且 MaxLength[j]+1>MaxLength[i])则 MaxCnt[i]=MaxCnt[j]，否则如果(MaxLength[j]+1= =MaxLength[i])可利用加法原理，MaxCnt[i]=MaxCnt[i]+MaxCnt[j]。

考虑到题目中说的不能又重复的序列，我们可以增加一个域 Next[i]表示大于 i 且离 i 最近的 Next[i]使得第 Next[i]个数与第 i 个数相同。如果不存在这样的数则 Next[i]=0。这样我们在 DP 的时候如果出现 Next[i]不为 0 且 Next[j]<i 可直接跳过。

这个题数据规模很大，需要用到高精度计算，还好只是加法。

优化技巧

可以在给出的序列的末尾增加一个 0，这样直接统计以最后一个 0 结尾的最长下降子序列即可。显然题目的规模要求我们用高精度。可以使用 longint，8 位 8 位的加以节省时间。

判重方法 2

对于任意一个 f[i]=max(f[j])+1; 1<=j<i; 设 g[i] 为到计算 i 的方案总数 若完成后 f[i]值为 ans[i] 若不判重 那么 g[i]=sum(g[j]){ f[j]=ans[i];1<=j<i} 那么容易知道对于所有具有相同 f[j] 的原数列值必为不降，如果有一降序，则必定会导致以该位置结尾的最长下降子序列长度增加，所以可以记录上一位置的 a[j]数值，相同 a[j]的 g[j]无需加入即可。 (及去掉这个不升序列中的相等部分，注意去掉靠前的)。

```
for i:=2 to n do
    for j:=i-1 downto 1 do//注意此处从 i-1 开始循环
```

判重方法 3

记录 ago[i]为 1 到 i-1 中位置与 i 最接近且股价相同的位置(没有就记录为 0)，显然 ago[i]的状态是由之前的状态推出来的，所以 i 的状态如果也由 ago[i]之前的状态推出来的，那么定然重复，所以 i 的状态一定是由 ago[i]+1 to i-1 所推出来的，此时因为 ago[i]+1 to i-1 已经处理过，其中也无重复，所以此时 i 的状态一定没有重复。(因为最初是一定有不重复的，所以由这些不重复的状态推出来的也一定不会重复)。具体程序见 Pascal 程序代码。

The Primes (prime3)

问题分析

这道题比较难。采取枚举模拟搜索，但是需要优化。

优化技巧

一行行的枚举连样例都过不了，考虑这题格子之间有相互约束，所以首先要优化枚举顺序。先枚举两条对角线，因为他们控制的行列是最多的，而且左上角的数已经确定了。接着枚举中间的一竖列，因为它同样控制的行列最多，接着枚举最上面一行和最下面一行。此时第 3 行的 2，4 两个数也可以通过减法得到了。最后枚举第 2，4 行，第 3 行减法得到。于是所有格子都已经确定了，验证一下就 OK。

数据结构方面，需要多个数组记录满足条件的素数，比如已知第 1，3，5 位的所有素数或已知第 1 位的所有素数都要预处理出来。并且每个数用 5 个 1 位数储存，这样操作方便而且效率高。全程都需要时刻剪去不满足要求的情况。注意细节，此题相当繁杂。

Street Race (race3)

问题分析

一道基础的连通分量的图论题。这个题默认了 0 为起点，N 为终点，如果不放心可以再读入的时候判断起点和终点，即入度为 0 的点为起点，出度为 0 的点为终点。

该题有两问，第一问很简单，可以尝试去掉每一个点，判断从起点到终点是否有通路，如果没有则该点为“必经点”。

第二问重点在于理解题意。首先可以确定第二问的解集是第一问的子集，所以我们可以第一问得出的每个点深搜，记录下可以到达的点。然后去掉该点，从起点深搜。如果不存在两次深搜皆可到达的点，就说明它是分割。

关于第二问我使用的办法是深度判断，过了测试。对原图 bfs 后，标记深度，对于结果 1 中的点 bfs，如果遍历到的点深度小于起点，则不为中间点。

与方法一类似一个方法

首先，在第一次 dfs 的时候，先标志检查的点为红色，然后从 0 开始 dfs，标上红色。如果这时候 n 点没有标上红色，则检查点为必经点。然后，第二问。。在第一问标上红色后，如果是必经点，把检查点标上黑色，从检查点开始遍历，标上黑色，如果遍历过程中没有见过红色点，最后到达了 n 点，则检查点是中间路口。应该很容易证明。

Letter Game (lgame)

这个题要仔细读，理解题意。起初我理解得不对，以为是输出的每个结果都必须是输入的一个排列，不能多也不能少。这种理解不正确。

这道题正确的理解应该是，在结果的单词或词组构所成的字母中，每个字母出现的频数不大于输入的字符串中每个字母的频数。也就是例如输入是 aaa，字典中有单词 a，aa，aaa，aaaa，其中 a，aa，aaa 都是符合条件的解，但只有 aaa 才是要输出的最优解。

理解题意以后编程不难，直接枚举每种可行解，单词可以在 O(n)内解决，词组要 O(N^2)，但是对于 40000 个单词但是太多了。可以发现，大多数单词都是用不到的，而所以我们可以在读入字典的时候直接去除非可行解的单词，即该单词有输入字串中未出现的字母，或者在该单词中的字母中，有频数大于输入字串所含该字母频数的（例：输入字串为 aabb，该单词为 aaa，其中 a 的频数大于输入字串，必定无法有输入字串构成）。

这样优化可以去掉绝大部分的冗余，使得程序能够在很快的时间内算出结果。

所后别忘了对结果排序，可以把单词当作一个第二个单词为空串的词组，按字典顺序排序即可。

注意

题目中提到"find the highest scoring words or pairs of words that can be formed." 就是说，让你找到最高的得分，在第一行输出然后下面要输出所有的得分最高的单词或词组，并且按照字典序输出。给出的单词长度在 3~7 之间，所以最多只能用两个单词。

Section 4.4

Shuttle Puzzle (shuttle)

构造法

根据答案寻找规律（可以证明正确性）

3

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

2 1 -2 -2 -1 2 2 2 -1 -2 -2 1 2 -1

4

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

2 1 -2 -2 -1 2 2 2 1 -2 -2 -2 -2 1 2 2 2 -1 -2 -2 1 2 -1

5

5 7 8 6 4 3 5 7 9 10 8 6 4 2 1 3 5 7 9 11 10 8 6 4 2 3 5 7 9 8 6 4 5 7 6

2 1 -2 -2 -1 2 2 2 1 -2 -2 -2 -2 -1 2 2 2 2 2 -1 -2 -2 -2 -2 1 2 2 2 -1 -2 -2 1 2 -1

6

6 8 9 7 5 4 6 8 10 11 9 7 5 3 2 4 6 8 10 12 13 11 9 7 5 3 1 2 4 6 8 10 12 11 9 7 5 3 4 6 8 10 9 7 5 6 8 7

2 1 -2 -2 -1 2 2 2 1 -2 -2 -2 -2 -1 2 2 2 2 2 1 -2 -2 -2 -2 -2 -2 1 2 2 2 2 2 -1 -2 -2 -2 -2 1 2 2 2 -1 -2 -2 1 2 -1

数据分析

Usaco 在这题上并没有指明不可以用分析法，而且 dfs 肯定 TLE，所以我们取巧。

先观察样例数据，如果把还没移动的那一步也算上，那么空格的位置为

4 3 5 6 4 2 1 3 5 7 6 4 2 3 5 4 (n=3，样例)

5 4 6 7 5 3 2 4 6 8 9 7 5 3 1 2 4 6 8 7 5 3 4 6 5 (n=4)

我们凭借极其敏锐的眼光发现这组序列为

4 35 642 1357 642 35 4 (n=3，样例)

5 46 753 2468 97531 2468 753 46 5 (n=4)

即长度为 1，2，3，4，...，n，n+1，n，...，4，3，2，1 这样的 2n+1 组等差序列

我们讨论第 1~n+1 组序列，这些序列满足

*公差的绝对值为 2

*奇数组为降序列，偶数组为升序列

*对于第 i 组(1<=i<=n+1)，若为奇数组则首项为 n+i，偶数组则首项为 n-i+2

对于第 n+2~2n+1 组，可以由对称性求出。

输出时从第二组开始即可。把规律总结成这样后，代码应该很好写了吧。

其实这道题用搜索也能过，详见代码（最后一个）。

Pollutant Control (milk6)

本题比较麻烦，要用到最小割最大流定理。首先求出最大流，那么最小割的容量就是最大流的流量。然后找有没有容量=最小割容量的“桥”（这里的桥就是去掉这条边后，由源点出发无法到达汇点），如果有那么这个桥就是答案。然后对每条容量不大于最小割的边，去掉后求一次最大流（估计这里我的算法太麻烦了）。如果流量的减少量=这条边的容量，那么这条边一定属于最小割。找出这样所有的边后，如果这些边的容量和=最小割容量，那么符合题目条件的最小割已求出。剩下的工作就是搜索了。用 dfsid，每次增加深度限制，按边的编号大小顺序扩展，搜索是否存在一种方案使得这些边的权值与已求出的必然属于最小割的边的权值之和等于最小割。然后 judge 一下是否去掉这些边后源点与汇点不连通。如果不连通，则说明已经找到解，可直接输出。数据规模不算大，一般的 BFS 已经足可胜任最大流的工作。不过我用的是预流推进，最大数据也不超过 0.1s。在熟练的前提下，还是尽

量用更好的算法吧。

数据中可能存在两点之间有多条边的情况，可以在求最大流时把这些边并成一条边，然后在求最小割的边集时再拆开。

标程用了一种很不错的方法因为总共只有 1000 条边 那么将每个边容量*1001+1 作为新的容量（注意 这里指每次读入的每个边）。

那么最大流 mod 1001 就是最小割的边数。最大流=值 di1001。首先这对最大流求肯定没影响，其次，我们知道对以每一条增广路大小取决于最小的那个这里的+1 就派上了用场。

在同样可一减去 2 条边和减去一条边使其不连通时。减去两条边对应+2>+1。所以，求最大流时会选择+1。同样，对于那些必须删的边多少个+1，其实就等于删了多少条边。

直接枚举满流量的边求删的边是不对的。例如：

1-3:5

1-2-3:5

3-4:5;

可能求出最大流为 5，可能 1-2-3 为满流，但此时减去 1-2-3 或 1-3 都是不对的，必须减去 3-4。可以用残量网络求出传递闭包可以求出真正必须删的边。

很明显地，这是一个最小割的模型，设每边的权值为原始权值，则最大流的值就是最小割的容量，即最小损失。根据胡兄的论文(07WC)，只要从源点进行一次 floodfill 就可以得出最小割集。现在有两个问题：停止的线路数，使开始输入顺序最小为解决这两个问题，可将每边的权值修改为 500000+i+500000*1001*c，这个式子有什么用呢？首先，每边的权值都加上 500000，那么最大流(maxflow%(500000*1001))/500000 就是停止的线路的数量，取 500000 的原因是 500000> (0+999)*1000/2，即最大情况下 0~999 的 i 的和，否则可能由于 i 的值而使结果偏大。i 项的含义就很明显了，为使输入顺序最小。最后一项中的 1001 则是因为最多有 1000 条边，所以 500000*1001*c > 500000*1000，这三项分别独立，从而得出结果。

Frame Up (frameup)

拓扑排序

由题意可知，不存在一个矩形的一条边被完全覆盖，所以我们可以计算出矩形的坐标。读入时记录矩形的每个点中最小 x1，最小 y1，最大 x2，最大 y2。可知左上角 (x1，y1) 右下角 (x2，y2) 右上角 (x2，y1) 左下角 (x1，y2)。

查找该矩形 A 边上，非该矩形的字母 B，即覆盖矩形 A 被矩形 B 覆盖。建立一条有向边 B->A，表示 B 是 A 的必要条件。然后进行拓扑排序，搜索求所有的拓扑序列，并排序输出。

注意，该题中的字母可能不连续，不要直接 for(i='A';i<='Z';i++)。

字符串处理

每次找到一个完全框（也就是这个框的所有字母都可见，要么是原来的字母，要么是“*”，“*”是后面用到的临时标记），拿掉，把框上的所有字母都标记为“*”。重复上述过程直到所有框都被拿掉。因为要求输出所有答案，所以用 dfs 来寻找所有解。

Chapter5

Section 5.1

Fencing the Cows (fc)

问题分析

标准的凸包。所谓凸包（这里说的是平面点集的凸包），就是一个最小的凸多边形，把所有给定点包含在内。显然，凸包一定是由给定的点中的某一些点构成。

下面介绍一种快速而且简便的凸包算法——Graham Scan。

预处理要对所有点进行排序。找出一个最左边的点，如果有多个再找最下边的。然后以这个点为准对其他所有点按照逆时针顺序排序（其实顺时针也可以，但是大多数人的习惯是逆时针），这里要用到向量叉积。

算法很简单。先开一个栈，分别放入第 1、2、3 个点（这三个点显然都在凸包中）。然后从第 4 个点开始枚举。如果栈顶元素指向当前点的有向线段与栈顶下方元素指向栈顶的有向线段构成右手系（就是说往右转啦，这里又要用到叉积），则栈顶元素出栈，重复以上过程直到这两个线段构成左手系，把当前点放入栈顶。这样到第 n 个点枚举完后，栈中的元素就构成了一个凸包，而且还是有序的。由于每个点最多入栈和出栈各一次，所以这个算法的时间复杂度是 O(n)。加上先前排序的 O(nlogn)，总的时间复杂度就是 O(nlogn)。

题目就是求出凸包后统计一下周长，有了 Graham Scan，这个对大家就是小菜一碟了。

关于凸包的计算方法，详情请见 Graham Scan。

Starry Night (starry)

相信这题大多数人都会。朴素算法即可。先用 floodfill 找出星座，然后和以前的星座比较，如果相同则标上以前这个星座的号。如果没找到，编号 +1，这个星座的编号就是当前编号。一个优化：将所有的星座的大小（即横向的最大长度和纵向的最大长度）记录一下，比较的时候如果待比较星座的大小无论是否旋转都和当前星座不一样，则直接跳过。

旋转的时候怎么转（转坐标或者直接重新赋图）都行，有了上述优化，肯定不会超时。

在判断相似的时候可以借鉴 transform。除了大小之外还可以记录星座中星的数量，数量不一样马上可以 Pass。可以用最小表示法来优化。

Another Algorithm：因为我极暴力的枚举弄了 tle，一气之下改用 hash 随机进行标号。优化差不多。但是判断相同的时候用的是 hash 码。

Musical Themes (theme)

算法 0 O(n3)

枚举两串的开头，贪心向下找即可（注意优化和重叠）。

算法 1 O(n4)

很直观的想法，就是枚举每一段的头尾，然后找在整个序列中有没有和这一段匹配的（也就是说所有的数都是原来的那一段对应位置的数加一个数或减一个数得到的，题目中有说明）。这样算法的时间复杂度可以达到 O(n4)。所以这个算法必须进行优化。

优化 1：如果以 i 开头的长度为 l 的序列没有匹配序列，那么比 l 更长的序列一定不会有解，可以直接跳过。

优化 2：每次枚举末端那一位时都从开始的一位加上当前最大长度开始枚举。

优化 3：对于以 i 开头的序列和以 k 开头的序列，如果 a[i]-a[i+1]<>a[k]-a[k+1]，那么不用 judge，直接跳过。

其中优化 1 的效果是最明显的，优化 3 次之，优化 2 的效果最不明显。加上上述优化，程序就可以过了，但是有点慢。加上二分的效率会更好。

算法 2 O(n3)

考虑到相同的旋律之间的差是常数，可以把读入的序列变换一下。就是每个元素与其前一个元素做差。例如原序列 {3，5，7，3，4，4，6，8，4}，做差后是{2，2，-4，1，0，2，2，-4}。这样就可以再变换后的序列中直接查找最长的重复序列即可。上述例子中是 2，2，-4，长度为 3，对应原序列中 3，5，7，3，长度为 4。

寻找最长的重复序列，我采用了一种 O(N3)的算法。就是枚举两个序列的开头的序列的长度。但对于 5000，O(N^3)还是难以过全的。于是采用了一个优化：每次枚举末端那一位时都从开始的一位加上当前最大长度开始枚举，前一个序列开头只用枚举到 (N-当前的最大

长度)即可。这个优化的力度很大，加上这个优化就全过了，而且很快。

算法 3 O(n2)

时间复杂度只有 O(n2)。但是要在对第一种算法比较深入分析的基础上才可以得出。显然，一个序列最多只需要一个匹配序列即可，所以我们只需找到这样两个序列。两层循环，外层循环枚举两个序列开头的位置差（也就是每个元素与另一个序列对应元素的位置差），内层循环枚举第一个序列的第一个元素的位置。如果当前位匹配则当前长度加 1，否则赋为 1。这样，我们可以直接找出合法的序列，并记录最大长度。这种算法抓住了问题的本质，比第一种算法无论在时间上还是编程复杂度上都降低了不少。

或者动态规划

令 theme(i，j)表示第一个主题开头为 i，第二个主题开头为 j 所能构成的重复主题的长度，那么有转移方程：

if note(i+1)-note(i)=note(j+1)-note(j) then theme(i，j)=them(i+1，j+1)+1 else theme(i，j)=1;

这个方程需满足 theme(i+1，j+1)+1 <= j-i 否则两主题将会重叠。

路人注释：其实是 O（n^3）

算法 4 O(n2lgn)

二分(lg(n))，然后枚举开头用 KMP 判断子串(O(N^2)/2)。

算法 5 O(nlogn)

1.利用算法二的想法，也就是当成串来匹配。

2.假如已经知道长度 d 存在，那可以利用哈希表，扫过一次整个数组(O(n))，如果前面出现过，代表有重复。需要注意的是，两个可匹配串之间必须至少有一个其它元素。

3.对于长度 d 去做二分搜(O(lgn))。因为若长度 d 存在重复旋律，那对于长度 e<d 必也存在重复旋律。故总复杂度 O(nlgn)。

USACO 对内存的限制比较狠，所以需要 Hash 冲突问题。

算法 6 O(nlogn)

计算做差，然后用后缀数组求最长重复子串。

Section 5.2

Snail Trail (snail)

问题分析

直接深搜就行了。每次沿着一个方向一直扩展直到碰到障碍物或者到达边界或者碰到已经走过的点。中间记录一下所有走到过的点，同时记录路径长度。当碰到已经走过的点时结束，判断以下路径长度与最优解的大小，如果更优则替换。

注意题目中说的“当 N>26 时，输入文件就不能表示 Z 列以后的路障了”，这句话不用专门理他。其实就是从 A 的 ascii 码开始向后顺延，不管是什么字母就行了。意思是：路障的列数不会超过 26。

Electric Fences (fence3)

首先很容易看出，这道题目精度要求并不高，所以我们完全可以把所有数据扩大 10 倍，这样就可以按照整数来处理了。 PS：针对下面两个算法提下自己的意见。这种逐渐加精度的搜索方法是可以过 Usaco 的，但是并不代表是严谨的。我们如果根据每个点做电源时的最短电线绘制一个类似等高线的等距图，那么可能存在一个大坑和一个小坑，大坑的最低点就是答案，但是在搜索的时候我们掉进了小坑出不来，那么将无法找到答案。 PS：针对上面的评论提一下自己的意见。下面的模拟退火似乎不太标准。如果用标准的模拟退火的话应该可以随机掉这个问题，因为标准的模拟退火在比原解劣的情况下仍有可能接受这个解。

算法一 局部搜索

当供电点固定后，计算电网的总长度并不难，这道题的核心任务就在于找到这个点。对于查找这个点，采用一般的搜索是会超时的。采用了“局部搜索”的方法。就是先以一个较大的距离网格扫描，找到目前使结果最短的点的坐标。最终结果一定在目前找到的点的附近，然后对这个范围进行细化的搜索，直到找到要求的精度为止。

算法二 随机化

题目的精度要求不高，采用迭代逼近的思想，首先随便选一个点，确定一个长度范围，在这个长度范围内随便向一个方向走任意的长度，如果这个点比先前的点更优则替换。重复足够多次以后，将长度范围缩小一点，继续迭代，直到长度范围 =0。这样所得到的解在精度范围内几乎绝对是正确的。（我取的长度范围是 30，随机 500 次，长度范围缩小值为 1，以上全部都是扩大 10 倍以后的数据） ——即是传说中的模拟退火算法

算法三 数学计算

具体方法很恶心，不推荐。[需要证实]

算法四 二维三分搜索

题目相当于求二元函数的极值，可先对 y 进行三分。y 固定后，题目相当于求一元函数的极值，此时对 x 进行三分搜索。

算法五

很多随机算法，比如遗传算法也是可以的。详情请见 C++代码.

Wisconsin Squares (wissqu)

搜索。随便加一点优化，只要你的算法能在 5s 内通过样例，就尽管提交，保证不超时。只有一个数据就是样例。

本题一般来说同样的算法，用 c++比较不会 tle。如果用 pascal，然后不 cheat（例如说前 3 位规定为 DCA 或 if count=149** then halt）的话 pascal 要 ac 要充分的常数优化技巧，包括各种循环的位置、编译开关和固定内存空间、inline 等。

（如果你的程序在你自己的机子上 2s，就可以 ac，否则 tle。貌似 USACO 这道题 5s 实际只类似于 2s。）

Section 5.3

Milk Measuring (milk4)

要用到迭代加深搜索(DFSID)。由于要求输出的是使用最少的牛奶桶，所以要先找牛奶桶数量为 1 的时候所有的组合，如果没有解再找牛奶桶数量为 2...直到牛奶桶数量为 P。

当搜索到一个组合，判断用这些牛奶桶是否能组成目标解的时候，可以用动态规划的方法来做。设 f[i]是当需求的牛奶为 i 时，能否形成这个组合，是一个布尔型数组。

初始条件 f[0]=true

状态转移方程 f[i]=f[i] or f[i-v[j]] (j 为使用的所有牛奶桶)

目标状态 f[Q]。如果 f[Q]为 true，则当前解合法，直接输出即可。

{但是如果仅仅这样写还是有一组数据过不去，需要进行一些优化。要优化动态规划的过程。

注意一个重要的信息，找到的组合中，每个牛奶桶至少用了一次。上面的状态转移方程中有许多某个牛奶桶使用 0 次的冗余状态。可以在初始的时候对(i=1..Q/v[第一个桶]) f[i*v[第一个桶]]赋值为 true。对每个其他的桶的状态可以直接由前面的状态得出。 经过这个优化，数据就可以全过了。

裸 DFSID+DP（记忆化搜索）。

Window Area (window)

算法一

算法并不难，只要想清楚了就很容易写出程序了。首先，对于每块放入的窗户，设定一个高度值。这个高度值应该是所有窗户中最大的。同时要维护两个值：所有窗户的最大高度和最小高度，初始时这两个值均为 0。每次新放入窗户时，这个窗户的高度就设为最大高度+1，同时更新最大高度。置顶和置底的操作相应进行。去掉窗户时更新最大高度和最小高度。这样前 4 个操作就很容易了。现在来考虑第 5 个操作。需要用到矩形切割的知识。所谓矩形切割，就是把两个矩形不相交的部分切开，以利于统计。比如两个矩形 a，b，如果 a 有在 b 左边的部分，那么就把它切下来；接下来如果有在 b 右边的部分，也切下来；同样，上下也可以仿照这种方法操作。实现第 5 个操作需要开一个队列。初始时队首为待统计的矩形。实现的时候每次先找一个高度比待统计矩形高，且没有计算过的矩形，记录一下当前队尾。取出一个队首元素，用这个矩形切割队首，切得的所有矩形依次放入队尾。重复上述步骤直到当前队首等于原来的队尾。所有符合的矩形都统计完以后，队列中所有矩形的面积和除以待统计矩形的面积就是这个操作的答案。

注：数据规模很小，最大的数据也只有 600 行而已，所以队列完全不必要用链队，数组即可。

算法二

其实思想和算法一是一样的，但是使用了字符串操作，大大简化了代码。另，计算面积时参考了 3.1 的 rect1。详细实现可以参考源代码。

算法三

这道题用的是矩形切割。（我所理解的）矩形切割就是某一个矩形被上面的矩形所切割，最多切成 9 个小矩形。然后再用更上面一层的矩形（递归地）切割这切出来的周围 8 个小矩形，一直处理到上面每一层的矩形都结束。

我第一次做这个题用的方法是： 1.用一个数组 win 来记录每“层”对应的窗口。 2.每次新建窗口，新建一个最上层，放在最上层。 3.每次删除窗口，就把 win[这一层]赋值为 0 对应的 ASCII 码。 4.每次置顶：在顶上新建一层，放在上面。删除原窗口。置底与置顶类似。 5.每次 s()操作：对这个矩形调用矩形切割函数。用上面的每层来切割它。最后没有被覆盖的就是露着的面积。这个方法很简单，所以我感觉这个题很简单。结果，一交，发现第 10 组数据超时。

这组数据的 s 操作相当多，其他操作很少。所以反复调用函数就会超时。

于是我想了一个新的方法。用 see[]记录某个矩形能看见的面积。每次新建、删除、置顶、置底都要维护，而 s()就可以直接出结果。

1.新建窗口时：能看见面积=总面积。从上向下做矩形切割。下面的矩形减去被这个矩形盖住的面积。

2.置顶时：能看见面积=总面积。从上向下做矩形切割。这个矩形原来的位置以上的所有矩形减去相应的面积。原来位置一下的矩形面积不变。

3.置底时：能看见面积=总面积。从上向下做矩形切割。每次该矩形减去相应面积。原位置以上所有矩形面积不变。以下矩形加上相应面积。

4.删除时：从上向下做矩形切割。原位置以上所有矩形面积不变。以下矩形加上相应面积。

注意数据阴人。给出的矩形两个对角，可能是左上右下，也可能是左下右上！一定要处理！附：我写的矩形切割是自己发明的代码，不是正统的版本，比较麻烦。

Network of Schools (schlnet)

这是一道收缩强连通分量的题。

该题描述的是一个有向图。我们都知道，在一个有向图强连通分量中从任意一个顶点开始，可以到达强连通分量的每个顶点。由此可以把该题中所有强连通分量收缩成分别一个顶

点，则入度为 0 的顶点就是最少要接受新软件副本的学校。

第二问就是，问至少添加多少条边，才能使原图强连通。也就问在收缩后的图至少添加多少条边，才能使之强连通。

可以知道，当存在一个顶点入度为 0 或者出度为 0 的时候，该图一定不是强连通的。为了使添加的边最少，则应该把入度为 0 顶点和出度为 0 的顶点每个顶点添加 1 条边，使图中不存在入度为 0 顶点和出度为 0 的顶点。

当入度为 0 的顶点多于出度为 0 的顶点，则应添加的边数应为入度为 0 的顶点的个数。当出度为 0 的顶点多于出入度为 0 的顶点，则应添加的边数应为出度为 0 的顶点的个数。

这样就可以解决问题了。但是不要忘了还有特殊的情况，当原图本身就是强连通分量时，收缩成一个顶点，该顶点入度和出度都为 0，但第一问应为 1，第二问应为 0。

求强连通分量，我用的两遍深搜的 Kosaraju 算法，时间复杂度为 O(n+m)。把找到的每个强连通分量收缩为一的顶点，组成新图。设 r(x)为 x 所在的强连同分量的代表节点，如果原图中存在边 e(x，y)，那么新图中有边 e(r(x)，r(y)) 。然后根据点的邻接关系统计出度和入度即可。

（我认为此方法第二问不完全正确：很显然，此方法仅在只有一个联通的 DAG（有向无环图）中是成立的，但是当有多个 DAG 时，便不再成立。

比如：

2

0

0

很显然 答案是 2，而非 0.

正确的做法是：求出每一个 DAG 的入度为 0 的点的数量，出度为 0 的点的数量。若 DAG 的数量>1，则对于强连通分量点答案要+1.

压缩强连通分支另构建成新图。独立的点应该既算作入度为 0 的点，也是出度为 0 的点，所以第二问答案依然是 max(入度为 0 的点数，出度为 0 的点数)，只是当整个图只有一个点的时候，第二问的答案为 0。

对于 3 楼的解释：现已知两个 DAG，那么一定可以将其中一个 DAG 先连成 scc，然后再将其中任意一个环断开（由于一个 scc 内可能含有不止一个环），然后再去接上另外一个 DAG。第二个 DAG（另外的那个）同样先连成 scc，然后断开某环曲接另外的环。这样能保证在两个新的 scc（即将断开并合并）在都包含不止一个环时仍有解。

换一种思路：现在第二问的目的是：要把已经再第一问里做好的 scc 图连成一个新 SCC，那么只要求出来 min（入度为 0 的点数，出度为 0 的点数）即可。

另一种方法

Sinya 觉得写深搜求太麻烦了，所以 Sinya 就用了另一种求强连通分量的方法。

用 floyed 算出两点间是否可以互相到达。然后枚举任意两个顶点 Vi 还有 Vj，如果两个点互相可以到达，那么他们就是属于同一个强连通分量了。

虽然是 O(n^3)的算法。可是这道题能过。可以大幅降低编程复杂度。

旁门左道

第一问是求最小点基。这个我是分步骤计算的：

首先，所有入度为 0 的点肯定要得到软件，因为如果得不到，那么没有别的点来通过网络传输给它。找出所有入度为 0 的点，把这些点以及他所能到达的点全都作上标记。

对于剩下的点，找出块的个数，这里定义两个点连通当且仅当两个点之间存在路径，不考虑方向。原因很简单，两个点之间只要其中一个能到达另一个即可，这样的块中必然有一个点可以作为起点，而由于前一步已经把入度为 0 的点去掉了，所以这样的块一定从起点可

以到达所有点。

第一问的答案就是上述两者个数之和。

第二问首先统计整个图的入度为 0 和出度为 0 的点的个数，前者再加上上一步求出来的块的个数（当整个图就是一个块 并且 存在入度为 0 的点的时候不用加），答案就是两者中的较大者。

Big Barn (bigbrn)

动态规划。可参考 home on the range 一题

状态：F[i][j] 表示以(i，j)为左上角最大正方形的边长

初始条件：

如果(i，N)没有障碍 F[i][N]=1 否则 F[i][N]=0

如果(N，i)没有障碍 F[N][i]=1 否则 F[N][i]=0

状态转移方程：if f[i][j]>0 then //(i，j)没有树。

F[i][j]=min(F[i+1][j]，F[i][j+1]，F[i+1][j+1])+1;

极大化思想。这题还可以用极大化思想的做，这种方法的延伸性更强一些，但就这题而言，程序比较长，同样是 O(n^2)，相比 DP 没有优势。这题也可用做到 O(N*log(n))级别，方法同 IOI 一题。

Section 5.4

All Latin Squares (latin)

去想怎么构造纯属浪费时间。这题似乎没有什么规律。只能搜索。朴素搜索只能过 6 个点，7 就 TLE 了。

经过测试发现，在 n=7 时，经过剪枝共有 12198297600 个解。如果将 DFS 改成枚举，再修炼一下代码能力，应该不会 TLE 的。

优化技巧

只需要搜边长 n-1 的正方形即可。在第一行的排列已经确定以后，第 1 列的每一种排列对应的方案数都相同。所以只需要搜索(2，2)开始的边长为 n-1 的正方形即可。 搜到第 4 行就不用搜了，第五行已经确定了。

超级优化

用到一点置换的知识。对于相邻的两行，搜索出来后可以得到几个置换圈，把这些圈的个数从小到大排序后记录下来。那么对于两种方案，如果圈的个数相同且对应圈的大小相同，那么这两种方案本质上是相同的。可以用一个 hash 记录。记录的时候把所有置换圈的长度相乘，那么对于个数相同的方案肯定不会冲突。（这样貌似不对，因为对于置换圈(2，2，2)和(4，2)算出来的记录是一样的，但它们的方案数貌似不一样。）

Canada Tour (tour)

动态规划

把返回的路线反向，那么整条路线就变成了两条不相交的从起点到终点的路线。这样我们可以用 DP 解决。

状态设定：f[i，j] 为假定的甲乙两人，甲走到第 i 个城市，乙走到第 j 个城市时，两人走过的城市数目的和。

初始状态：f[1，1]=1。

状态转移方程：f[j，i]=f[i，j]=max{f[i，k]+1}(k 到 j 存在飞机航线，以及 f[i，k]>0，就是说存在 f[i，k]的走法，1<=k<j。

交换甲乙，则肯定有 f[j，i]=f[i，j]。

目标结果：由于题中告知必须走到终点才能返回，输出结果一定是 max{f[i，N]}(i 到 N

存在飞机航线)。如果没有经过城市数目大于 1 的可行目标状态，则无法完成这次环游，输出 1。

网络流

建模方法：把第 i 个城市拆分成两个顶点<i.a>，<i.b>。 1、对于每个城市 i，连接(<i.a>，<i.b>)一条容量为 1，费用为 1 的有向边，特殊地(<1.a>，<1.b>)和(<N.a>，<N.b>)容量设为 2。 2、如果城市 i，j(j>i)之间有航线，从<i.b>到<j.a>连接一条容量为 1，费用为 0 的有向边。 求源<1.a>到汇<N.b>的最大费用最大流。 如果(<1.a>，<1.b>)不是满流，那么无解。否则存在解，即为最大费用最大流量 - 2。

建模分析：每条航线都是自西向东，本题可以转化为求航线图中从 1 到 N 两条不相交的路径，使得路径长度之和最大。转化为网络流模型，就是找两条最长的增广路。由于每个城市只能访问一次，要把城市拆成两个点，之间连接一条容量为 1 的边，费用设为 1。因为要找两条路径，所以起始点和终点内部的边容量要设为 2。那么费用流值-2 就是两条路径长度之和，为什么减 2，因为有两条容量为 2 的边多算了 1 的费用。求最大费用最大流后，如果(<1.a>，<1.b>)不是满流，那么我们找到的路径不够 2 条（可能是 1 条，也可能 0 条），所以无解。

可能的简化方法--最大流：因为这种建图方法的特殊性，我们注意到：一条从源到汇的路上的边费用必然交替是 1/0 （因为先经过一条由原节点拆成的边（费用 1） 再经过本来原图就有的边（费用 0））。那么我们只需要求得最大流减去 2 再除以 2 就可以得到答案。样例：（7 + 9）/ 2 = 7。

Character Recognition (charrec)

乍一看有点无从下手。本来想着是统计问题，后来看了北极天南星大牛的题解，发现有最优子结构。所以这题的标准算法是 DP。首先从 font.in 中把 27 个字符提取出来，单独构成 27 个图，称为字符图集。这样是为了处理方便。设 b[i]表示给定图从第 i 行开始匹配所能得到的最小差距，c[i，j]表示给定图从第 i 行开始连续匹配 j 行所能得到的最小差距，dif[i，j，k]表示第 i 个字符图的第 j 行与给定图的第 k 行的差距。

b[i]:=min(b[i+19]+c[i，19]，b[i+20]+c[i，20]，b[i+21]+c[i，21])

其中 c[i，j]的计算方法：

j=19：枚举字母。设 pre[i]表示字符图前 i 行匹配的差距，tail[i]表示后 i 行匹配的差距，则 c[i，j]:=min(pre[k]+tail[19-k]).

j=20：直接枚举字符，统计即可。

j=21：与 j=19 相仿。

对于其中涉及到的统计问题，可以从 dif[i，j，k]直接获得，避免了很多重复计算

多了一行或少了一行是不用计入到差异值之中的，所以上面说的 c[i][19] 可以这样算：

先枚举 27 个字母，再枚举缺的是哪一行，比如当前枚举到缺的是第 13 行，那么差异值就是 给定图第 i 行与字符图第 i 行 (i <= 12) 的差异值之和 与 给定图第 i 行与字符图第 i + 1 行 (i >= 13) 的差异值之和。c[i][21] 同样做。c[i][j] 算出来了之后，就好 DP 了。

DP 状态表示为： DP[i]表示匹配到了 给定图 的第 i 行，所得到的最小的差异值，

那么 DP[i] 就可以由 DP[i-19] DP[i-20] DP[i-21] 得到。

Betsy's Tour (betsy)

搜索

这道题要使用 DFS 加上优化才可以过。朴素的搜索只能解决到 N=5，6 会超时。于是要加上一些优化。

优化 1

不走死胡同！所谓死胡同，就是走进去以后就再也无法走出来的点。

一种简单的想法是：任意时刻，必须保证和当前所在位置不相邻的未经点至少有两个相邻的未经点。基于这种想法，可以采取这样的优化：

当前点周围的点 D，如果只有一个与 D 相邻的未经点，则点 D 为必经点。

显然，如果当前点周围有两个或两个以上的符合上述条件的必经点，则无论走哪个必经点都会造成一个死胡同，需要剪枝。

如果当前点周围只有一个必经点，则一定要走到这个点。

如果该点周围没有必经点，则需要枚举周围每一个点。

该优化的力度很大，可以在 0.2 秒内解决 N=6，但 N=7 仍然需要 2 秒左右的时间。

优化 2

不要形成孤立的区域。如果行走过程中把路一分为二，那么肯定有一部分再也走不到了，需要剪枝。

形成孤立的区域的情况很多，如果使用 Floodfill 找连通快，代价会很大，反而会更慢。我只考虑了一种最容易出现特殊情况，即：

当前点左右都是已经点(包括边缘)，而上下都是未经点；

当前点上下都是已经点(包括边缘)，而左右都是未经点。

这样就会形成孤立的区域，只要将出现这种情况的状态都剪掉即可。

加上优化 2，N=7 也能在 0.3s 解决了。

常数优化

Pascal 的时限还是蛮紧的，应尽量减少函数/过程/循环，实在不行使用 inline 也可以大幅加速。

基于连通性状态压缩的动态规划

参见 WC2008 的雅礼中学陈丹琦论文《基于连通性状态压缩的动态规划》。

小技巧

终点和起点间连一条“[”形的边，将哈密顿路转换成哈密顿回路，就不需要独立插头和一大堆繁琐的判断了。再将棋盘顺时针旋转 90 度，更加简化了问题。

TeleCowmunication (telecow)

算法和 4.4 的 pollutant control(求最小边割集)类似，但这道题是求最小点割集。

我们可以把每个点 i 拆成两个点 i1，i2，这两个点之间建立一条边权为 1 的有向弧。对于原图中边(i，j)，建立(i2，j1)和(j2，i1)两条边权为∞的有向弧。这样就把求最小点割转化成了求最小边割。

根据最大流最小割定理：源 S 到汇 T 的网络最大流等于 S 与 T 间最小边割集的容量和。只需对新图求网络最大流，记为 netflow，在这里我用了 Dinic。这样就完成了第一问，结果为 netflow。

第二问，求最小边割集的办法可参考 07 年胡伯涛的论文，用一次 floodfill 即可。最后，因为要输出字典序最小，边权可修改为 6000+i，其中 i 为编号，最后 maxflow 的结果再除以 6000 即可

其实不拆点也能做，求最大流的时候不断用 Dijkstra 求源点到汇点的最短路，总流量加一，删去路径上的点，直到找不到路径为止。先求原图的最大流 value，然后枚举每一个点，如果删除它后最大流减少 1 则将其放入数组。value 就是答案的个数，dfs 一下就能得到答案。(详见 PASCAL 程序)

一点想法

改边权为 6000+i 不能保证字典序最小，只能保证选出来的点的编号和最小。我想的方法和 Chapter 4 的那题差不多，顺序枚举删除每一条拆点连的边，然后看是否流量的变化只是少了 1。程序贴在 C++的范例程序里了。

Section 5.5

Picture (picture)

算法一 离散化

把所有矩形离散化(就是将整个平面分成许多“竖条”或“横条”，对其操作)，每个矩形都由四条边组成，分为纵边和横边。对纵边和横边分别扫描一次，以横边为例：

每个矩形的两条横边中，称下面的为始边，上面的为终边。把每条横边以纵坐标从小到大排序，如果纵坐标相同，则应把始边排到终边之前。

依次枚举每条横边。如果当前边为始边，则把这条边的横向的所有点 j 的层数增加 1，即为 level[j]++。如果层数由 0 变为了 1，则这一点一定是边缘点，总周长 ans++。如果当前边为终边，则把这条边的横向的所有点 j 的层数减少 1，即为 level[j]--。如果层数由 1 变为了 0，则这一点一定是边缘点，总周长 ans++。

同理按此方法扫描纵边，即可得到最后结果。

算法二

总思路：离散+线段树 (其实就是提高 level[j]++的效率)

首先是离散：

显然，我们有 2n 条纵线和 2n 条横线。算法中，我们只考虑纵线，因为横线的做法同纵线的做法是相同的。离散就是将这 2n 条线段按照从左到右的顺序排序（也就是按照每条纵线的横坐标从小到大排序)，这里需要注意一点：如果出现两个相同横坐标的纵线段，属于所在矩形左边的线段 要排在属于所在矩形右边的线段的左边。(这两个线段属于不同的矩形)。

然后是线段树：每个节点有 6 个属性：s，t，l，r，c，m，分别表示左边界、右边界、左子树、右子树、覆盖数、区间内线段总长度。

对于 2n 条纵线段，属于矩形左边的线段添加该线段到线段树(覆盖数+1)，属于矩形右边的线段则从线段树中删除该线段(覆盖数-1)。做添加、删除线段的同时，要维护 m 属性。规则如下：

如果该段线段覆盖数(c)>0，则 M 即为线段长，

如果覆盖数(c)=0，则 M 为 左儿子的 M+右儿子的 M。(如果本身是叶子就为 0)

每次操作线段后改变总长度（也可能不变)，如果原来的长度为 now，新的长度为 new，如果 new>now，则 new-now 算入答案 ans。

Hidden Passwords (hidden)

首先，为了解决环的情况，可以在这个字符串后面再接一个相同的字符串。这样不影响结果。本题可以采用比较法。我们可以用两个标记 i、j 表示两个待比较字符串（长度为 n）的起始位置。初始时 i=0，j=1。每次比较以 i 开头和以 j 开头的字符串 s[i]和 s[j]的大小，如果 s[i]>s[j]则 i=j，否则 j 向后移 k 位，k 表示 s[i]和 s[j]的最长相同前缀的长度，因为 s[i]的前 K 位的字母都大于等于 s[i]的第一个字母，而 s[i]与 s[j]的前 K 为字母相同，所以下一次只需比较 s[i+k]与 s[i]即可，k 在比较大小时顺便可以求出。不停比较直到 j≥n。这种算法可以应付大多数数据，但是因为最坏情况下算法的时间复杂度是 O(n^2)的，对于特殊构造的数据就不行了。不过 USACO 没有出这种数据，放心做吧。

几个需要注意的细节：当比较时发现两个字符串相同时，直接输出 0。k 的最小值为 1。

Two Five (twofive)

以下叙述中，“单词”均指合法单词。

举个例子说明：若为单词转编码，如求单词 ACF……的编码，则设一累加器，先累加以 AB 开头的单词的个数，再累加以 ACB 开头的单词的个数（这个数为 0，但若已知 6 个

字母的位置，B 拐到了第 2 行，则可能不为 0），再累加以 ACD 开头的单词的个数，再累加以 ACE 开头的单词的个数……最后加 1 即得答案。若为编码转单词，如求第 n 个单词，同样设一累加器 s，先累加以 AB 开头的单词的个数，若 s>=n 了，说明第二个字母就是 B，否则继续累加以 AC 开头的单词的个数……直到 s>=n，这样第二个字母就确定了。将最后一次累加的数减去，用类似的方法确定第三、第四……个字母，直至结束。

现在的问题是：如何求出以某给定序列开头的单词的个数？这个问题是用记忆化搜索解决的。用 f[a，b， c，d，e](5>=a>=b>=c>=d>=e>=0)表示把前 a+b+c+d+e 个字母填入第 1 行的前 a 个格，第 2 行的前 b 个格……第 5 行的前 e 个格，且已经确定位置的字母各就各位时可能的单词数，那么 f[0，0，0，0，0]就表示以给定序列开头的单词数。下面以求以 AC 开头的单词数为例说明递归求 f 数组的方法：

第一层递归安置字母 A。因其位置已固定，故 f[0，0，0，0，0]=f[1，0，0，0，0]，进入第二层递归计算 f[1，0，0，0，0]。

第二层递归安置字母 B。B 的位置尚未固定，于是枚举所有合法位置（合法位置指左、上方均已填有字母的位置，认为第 0 行与第 0 列均已填满。此例中为 12、21），分别进入第三层递归计算 f[2，0，0，0，0]（这个值等于 0，下面会讨论其原因）与 f[1，1，0，0，0]。f [1，0，0，0，0]即等于这二者之和。

第三层递归安置字母 C。这层递归的过程与第一层递归类似。更深层递归的过程与第二层递归类似。若在某一次递归中，需要计算的 f 值已经算出，则不必再递归下去，直接退出即可。

因为每次计算单词个数时给定的序列都不同，所以每次都必须从头递归。 (在具体实现的时候，可以像后面的程序一样，使用一个 5 位的 6 进制数来表示状态，这样可以把 5 维数组 f 降到 1 维) 。

优化技巧

程序的实现用了一点小技巧。上文中说，B 的合法位置有两个，分别是 12 和 21。但实际上，12 这个位置已经被字母 C 占据，只是在第二次递归时程序并不知道这一点。请看程序第 26 行中的这一条件：len[x[c]]+1=y[c]。如果某个位置已固定的字母的位置已被别的字母占据，那么这个条件就为假，从而求出的单词数就成了 0。当然，可以在递归之前把已被占据的位置做上标记，但因为需要搜索的状态总数本来就不多（只有 252 种），做标记不会显著提高时间效率，反而会增加编程复杂度。

除上段所说的以外，还有几点可以优化的地方，如以 ACB 开头的单词数可不经搜索直接得到 0，再如当递归深度超过了所有已固定的字母时，可直接利用版本 1 中的 DP 获得结果，而不须递归下去。然而，不加这些优化的算法的复杂度已经很低了（最坏情况下为 25(25 个位置)*25(每个位置可能放 25 个字母)*252(记忆化搜索的状态总数)*5(每个状态最多有 5 个合法位置)=787500），这些优化都显得不太值得。

Chapter6

Section 6.1

Postal Vans (vans)

诡异算法（至今不知道原理是什么）：设 a[i]表示第 i 列的左上和左下角有直线连通时的路径数，b[i]表示第 i 列的左上和左下角没有直线连通时的路径数，则有： b[i]:=a[i-1]+b[i-1]; a[i]=2+a[i-2]+2(a[i-2]+b[i-2]+a[i-1]+b[i-1]+...) =2+a[i-2]+2(b[i-1]+b[i-2]+...) 其中 a[i]的值可以在计算过程中累加。这样算法的时间复杂度仅为 O(n)。加上高精度运算也能轻松通过。

DP

考虑到要经过每一个点，那么对于最后形成的路，每一竖条横向的路不是 4 条就是 2 条，总共 8 种（不考虑路径方向）：

1 2 3 4 5 6 7 8

— — — — — —

— — — — — —

— — — — — — —

— — — — — — —

（7 中最上面的与最下面的相连、8 中最上面的与第二个相连）

并且对于经过最后面的 4 个点，路径形状只可能是两种（3、8），于是我们定义两个状态：

a[i]——有 i 竖条，以 3 号形状结尾的路径数

b[i]——有 i 竖条，以 8 号形状结尾的路径数

例如：对于 i=3：a[3]=2 b[3]=2

接下来的问题，就是如何进行状态转移。

我们考虑下面的几种情况：

对于 a[i]，它可以接在 a[i-2]、a[i-3]……与 b[i-2]、b[i-3]……上（这里的接上，指的是中间不出现 3 号与 8 号路径），并且除了 a[i-2]以外，其他都有两种接法（a[i-2]有 3 种），因此经过分类，我们可以得到：

a[i]=2*(a[i-2]+a[i-3]+……+a[2]+b[i-2]+b[i-3]+……+b[2]+1)+a[i-2]

而对于 b[i]，不难发现，他只能接在 a[i-1]与 b[i-1]上，因此有 b[i]=a[i-1]+b[i-1]。

于是我们可以将 a[i]化简为：a[i]=a[i-2]+2+2*(b[i-1]+b[i-2]+……b[3]) 。

至此问题大致解决，另外需要注意的是要用高精度。

A Rectangular Barn (rectbarn)

本题是求一个没有损坏区域的最大子矩阵。显然要用 DP 来做。当然，由于本题是矩形，所以 big barn 的方法就无法用到这里了。现在我们换一种思路。如果找到一个点，把这个点向上、左、右三个方向扩展，那么统计扩展出来的尽可能大的矩形就行了。看起来这种算法的时间复杂度很高，这也正是 DP 的用处——解决大量重复子问题。否则这样的算法就成了枚举了。

DP

设 h[i，j]为点(i，j)向上方扩展的最大高度，l[i，j]为(i，h[i，j])这条线段向左边扩展的最长距离，r[i，j]为(i，h[i，j])向右边扩展的最长距离。转移方程：

其中，tl 表示点(i，j)向左扩展的最大距离，tr 表示点(i，j)向右扩展的最大距离。

实现的时候有一个技巧：不用把整个图用一个二维数组表示出来，因为损坏点是很稀疏的。直接用一个数组保存点的坐标，然后离散化一下，这样可以很方便地找到每行的最长无损坏区间，这个区间内的所有点的数据都可以在 O(1)时间内计算出来。用这种方法可以节省很多空间。上述算法时间复杂度为 O(RC)，空间复杂度为 O(max(p，r，c))。

枚举

本题应该是利用极大矩阵思路，即求出所有的矩阵，然后找到最大值，而此题的关键就是怎样找所有的矩阵。显然此题我们应该找到一种效率高于 O(n^2*logn)的算法，要不然会超时。上面的算法效率比较高，usaco 上最后一个测试数据用了 0.588s。

极大化思想

参见 2003 年国家集训队，王知昆论文。复杂度 O（N^2)。

路径压缩

注意 USACO 的内存限制比较严格，需要用滚动数组。

最大子矩形

这道题初看很想 Big Barn，我也尝试用类似的方法做，后来发现这条路走不通。重新考虑：对于任意一个矩形，如果它的四边都不能继续扩展（碰到边界或坏点），则此矩形为极大矩形。显然，所求的最大矩形一定是极大矩形。对于任意一个点(i，j)，我们向上找到一个坏点或边界(k，j)，得到线段(k，j)-(i，j)，我们把线段向左右两边扩展，直到不能扩展为止。此时得到的矩形在上边和左右两边都不能扩展了。我们只要枚举每个点(i，j)，计算它依照上诉方法扩展的矩形的面积，取其中最大的一个就是所求。考虑动态规划：设 h[i，j]表示点(i，j)向上扩展的最大距离，l[i，j]，r[i，j]分别表示所得线段向左右扩展的最大距离，此时矩形的面积。

Cow XOR (cowxor)

位运算。首先，根据题目规模，O(n^2)的算法肯定不行。这就引导我们考虑 O(nlogn)或者 O(n)的算法。O(nlogn)的算法肯定要用到二分，可是本人暂时还没想出来。下面来说 O(n)的算法。

首先，如果把最大的数转换成二进制，也不过 21 位，而二进制的操作比起十进制来要简单的多，并且存在很多特殊性，可以有很多十进制没有的优化。这就是采用位运算的原因。维护 tot[i]表示前 i 个牛的异或值，则任意一段(i，j)的异或值为 tot[i-1] xor tot[j]。那么对于 i 以前的序列的最大值应为 max(tot[j-1] xor tot[i])(j<=i)。但我们不用枚举 j，而是用位运算来直接得到最优解。另外再维护一个 s，s[i]=k 表示离当前位最近的 k-1，使得 tot[k-1]的某个前缀为 i。对于 tot[i]，从大到小枚举每一位 j。令 q 等于 tot[i，j]的异或，则若 q 曾经出现过，那么解的第 j 位可以为 1，否则只能为 0。重复上述过程就可以得到当前最优解 t。然后用 tot[i]的所有前缀更新 s。最后的最优解就是答案。