第 1 章　算法设计基础

在《算法竞赛入门经典》一书中，已经讨论过算法分析、渐进时间复杂度等基本概念，以及分治、贪心、动态规划等常见的算法设计方法。本章是《算法竞赛入门经典》的延伸，通过例题介绍更多的算法设计方法和技巧。

1.1　思维的体操

例题 1　勇者斗恶龙（The Dragon of Loowater, UVa 11292）

你的王国里有一条 n 个头的恶龙，你希望雇一些骑士把它杀死（即砍掉所有头）。村里有 m 个骑士可以雇佣，一个能力值为 x 的骑士可以砍掉恶龙一个直径不超过 x 的头，且需要支付 x 个金币。如何雇佣骑士才能砍掉恶龙的所有头，且需要支付的金币最少？注意，一个骑士只能砍一个头（且不能被雇佣两次）。

【输入格式】

输入包含多组数据。每组数据的第一行为正整数 n 和 m（$1 \leqslant n,m \leqslant 20\,000$）；以下 n 行每行为一个整数，即恶龙每个头的直径；以下 m 行每行为一个整数，即每个骑士的能力。输入结束标志为 $n=m=0$。

【输出格式】

对于每组数据，输出最少花费。如果无解，输出“Loowater is doomed!”。

【样例输入】

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

【样例输出】

```
11
Loowater is doomed!
```

【分析】

能力强的骑士开价高是合理的，但如果被你派去砍一个很弱的头，就是浪费人才了。因此，可以把雇佣来的骑士按照能力从小到大排序，所有头按照直径从小到大排序，一个一个砍就可以了。当然，不能砍掉“当前需要砍的头”的骑士就不要雇佣了。代码如下。

```
#include<cstdio>
#include<algorithm>              //因为用到了 sort
using namespace std;

const int maxn = 20000 + 5;
int A[maxn], B[maxn];
int main() {
  int n, m;
  while(scanf("%d%d", &n, &m) == 2 && n && m) {
    for(int i = 0; i < n; i++) scanf("%d", &A[i]);
    for(int i = 0; i < m; i++) scanf("%d", &B[i]);
    sort(A, A+n);
    sort(B, B+m);
    int cur = 0;                 //当前需要砍掉的头的编号
    int cost = 0;                //当前总费用
    for(int i = 0; i < m; i++)
      if(B[i] >= A[cur]) {
        cost += B[i];            //雇佣该骑士
        if(++cur == n) break;    //如果头已经砍完，及时退出循环
      }
    if(cur < n) printf("Loowater is doomed!\n");
    else printf("%d\n", cost);
  }
  return 0;
}
```

例题 2　突击战（Commando War, UVa 11729）

你有 n 个部下，每个部下需要完成一项任务。第 i 个部下需要你花 B_i 分钟交待任务，然后他会立刻独立地、无间断地执行 J_i 分钟后完成任务。你需要选择交待任务的顺序，使得所有任务尽早执行完毕（即最后一个执行完的任务应尽早结束）。注意，不能同时给两个部下交待任务，但部下们可以同时执行他们各自的任务。

【输入格式】

输入包含多组数据，每组数据的第一行为部下的个数 N（$1 \leqslant N \leqslant 1\,000$）；以下 N 行每行两个正整数 B 和 J（$1 \leqslant B \leqslant 10\,000$，$1 \leqslant J \leqslant 10\,000$），即交待任务的时间和执行任务的时间。输入结束标志为 N=0。

【输出格式】

对于每组数据，输出所有任务完成的最短时间。

【样例输入】

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

【样例输出】

```
Case 1: 8
Case 2: 15
```

【分析】

直觉告诉我们，执行时间较长的任务应该先交待。于是我们想到这样一个贪心算法：按照 J 从大到小的顺序给各个任务排序，然后依次交待。代码如下。

```
#include<cstdio>
#include<vector>
#include<algorithm>
using namespace std;

struct Job {
  int j, b;
  bool operator < (const Job& x) const {    //运算符重载。不要忘记 const 修饰符
    return j > x.j;
  }
};

int main() {
  int n, b, j, kase = 1;
  while(scanf("%d", &n) == 1 && n) {
    vector<Job> v;
    for(int i = 0; i < n; i++) {
      scanf("%d%d", &b, &j); v.push_back((Job){j,b});
    }
    sort(v.begin(), v.end());                //使用 Job 类自己的 < 运算符排序
    int s = 0;
    int ans = 0;
    for(int i = 0; i < n; i++) {
```

```
        s += v[i].b;                //当前任务的开始执行时间
        ans = max(ans, s+v[i].j); //更新任务执行完毕时的最晚时间
      }
      printf("Case %d: %d\n", kase++, ans);
    }
    return 0;
}
```

上述代码直接交上去就可以通过测试了。

可是为什么这样做是对的呢？假设我们交换两个相邻的任务 X 和 Y（交换前 X 在 Y 之前，交换后 Y 在 X 之前），不难发现其他任务的完成时间没有影响，那么这两个任务呢？

情况一：交换之前，任务 Y 比 X 先结束，如图 1-1（a）所示。不难发现，交换之后 X 的结束时间延后，Y 的结束时间提前，最终答案不会变好。

情况二：交换之前，X 比 Y 先结束，因此交换后答案变好的充要条件是：交换后 X 的结束时间比交换前 Y 的结束时间早（交换后 Y 的结束时间肯定变早了），如图 1-1（b）所示。这个条件可以写成 $B[Y]+B[X]+J[X]<B[X]+B[Y]+J[Y]$，化简得 $J[X]<J[Y]$。这就是我们贪心的依据。

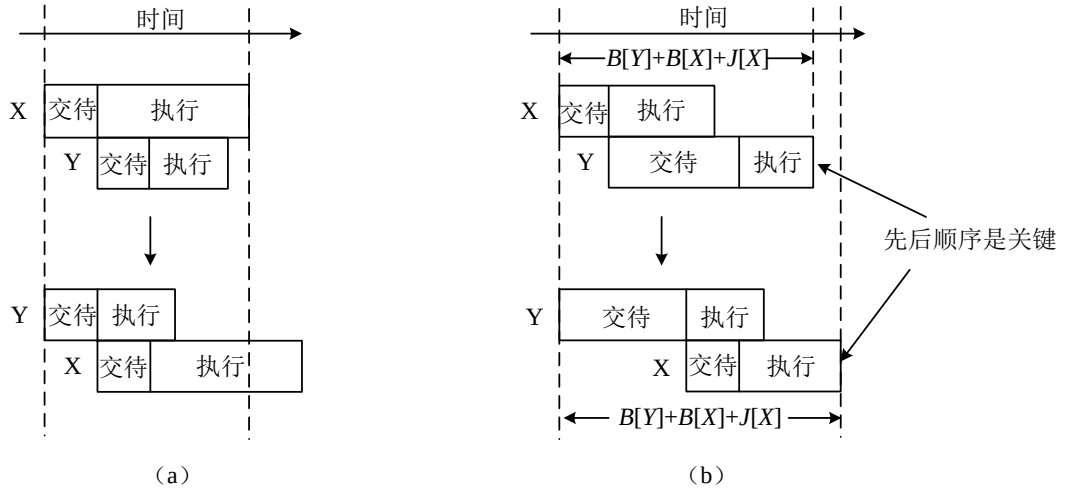


图 1-1

例题 3　分金币（Spreading the Wealth, UVa 11300）

圆桌旁坐着 n 个人，每人有一定数量的金币，金币总数能被 n 整除。每个人可以给他左右相邻的人一些金币，最终使得每个人的金币数目相等。你的任务是求出被转手的金币数量的最小值。比如，n=4，且 4 个人的金币数量分别为 1,2,5,4 时，只需转移 4 枚金币（第 3 个人给第 2 个人两枚金币，第 2 个人和第 4 个人分别给第 1 个人 1 枚金币）即可实现每人手中的金币数目相等。

【输入格式】

输入包含多组数据。每组数据第一行为整数 n（n≤1 000 000），以下 n 行每行为一个

整数，按逆时针顺序给出每个人拥有的金币数。输入结束标志为文件结束符（EOF）。

【输出格式】

对于每组数据，输出被转手金币数量的最小值。输入保证这个值在 64 位无符号整数范围内。

【样例输入】

```
3
100
100
100
4
1
2
5
4
```

【样例输出】

```
0
4
```

【分析】

这道题目看起来很复杂，让我们慢慢分析。首先，最终每个人的金币数量可以计算出来，它等于金币总数除以人数 n。接下来我们用 M 来表示每人最终拥有的金币数。

假设有 4 个人，按顺序编号为 1, 2, 3, 4。假设 1 号给 2 号 3 枚金币，然后 2 号又给 1 号 5 枚金币，这实际上等价于 2 号给 1 号 2 枚金币，而 1 号什么也没给 2 号。这样，可以设 x_2 表示 2 号给了 1 号多少个金币。如果 $x_2<0$，说明实际上是 1 号给了 2 号 $-x_2$ 枚金币。x_1，x_3 和 x_4 的含义类似。注意，由于是环形，x_1 指的是 1 号给 4 号多少金币。

现在假设编号为 i 的人初始有 A_i 枚金币。对于 1 号来说，他给了 4 号 x_1 枚金币，还剩 A_i-x_1 枚；但因为 2 号给了他 x_2 枚金币，所以最后还剩 $A_1-x_1+x_2$ 枚金币。根据题设，该金币数等于 M。换句话说，我们得到了一个方程：$A_1-x_1+x_2=M$。

同理，对于第 2 个人，有 $A_2-x_2+x_3=M$。最终，我们可以得到 n 个方程，一共有 n 个变量，是不是可以直接解方程组了呢？很可惜，还不行。因为从前 $n-1$ 个方程可以推导出最后一个方程（想一想，为什么）。所以，实际上只有 $n-1$ 个方程是有用的。

尽管无法直接解出答案，我们还是可以尝试着用 x_1 表示出其他的 x_i，则本题就变成了单变量的极值问题。

对于第 1 个人，$A_1-x_1+x_2=M \rightarrow x_2=M-A_1+x_1=x_1-C_1$（规定 $C_1=A_1-M$，下面类似）

对于第 2 个人，$A_2-x_2+x_3=M \rightarrow x_3=M-A_2+x_2=2M-A_1-A_2+x_1=x_1-C_2$

对于第 3 个人，$A_3-x_3+x_4=M \rightarrow x_4=M-A_3+x_3=3M-A_1-A_2-A_3+x_1=x_1-C_3$

…

对于第 n 个人，$A_n-x_n+x_1=M$。这是一个多余的等式，并不能给我们更多的信息（想一想，为什么）。

我们希望所有 x_i 的绝对值之和尽量小，即$|x_1|+|x_1-C_1|+|x_1-C_2|+\ldots+|x_1-C_{n-1}|$要最小。注意到$|x_1-C_i|$的几何意义是数轴上点 x_1 到 C_i 的距离，所以问题变成了：给定数轴上的 n 个点，找出一个到它们的距离之和尽量小的点。

下一步可能有些跳跃。不难猜到，这个最优的 x_1 就是这些数的“中位数”（即排序以后位于中间的数），因此只需要排个序就可以了。性急的读者可能又想跳过证明了，但是笔者希望您这次能好好读一读，因为它实在是太优美、太巧妙了，而且不少其他题目也能用上（我们很快就会再见到一例）。

注意，我们要证明的是：给定数轴上的 n 个点，在数轴上的所有点中，中位数离所有顶点的距离之和最小。凡是能转化为这个模型的题目都可以用中位数求解，并不只适用于本题。

让我们把数轴和上面的点画出来，如图 1-2 所示。

图 1-2

任意找一个点，比如图 1-2 中的灰点。它左边有 4 个输入点，右边有 2 个输入点。把它往左移动一点，不要移得太多，以免碰到输入点。假设移动了 d 单位距离，则灰点左边 4 个点到它的距离各减少了 d，右边的两个点到它的距离各增加了 d，但总的来说，距离之和减少了 $2d$。

如果灰点的左边有 2 个点，右边有 4 个点，道理类似，不过应该向右移动。换句话说，只要灰点左右的输入点不一样多，就不是最优解。什么情况下左右的输入点一样多呢？如果输入点一共有奇数个，则灰点必须和中间的那个点重合（中位数）；如果有偶数个，则灰点可以位于最中间的两个点之间的任意位置（还是中位数）。代码如下。

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

const int maxn = 1000000 + 10;
long long A[maxn], C[maxn], tot, M;
int main() {
  int n;
  while(scanf("%d", &n) == 1) { //输入数据大，scanf 比 cin 快
    tot = 0;
    for(int i = 1; i <= n; i++) { scanf("%lld", &A[i]); tot += A[i]; }
                                  //用%lld 输入 long long
    M = tot / n;
    C[0] = 0;
    for(int i = 1; i < n; i++) C[i] = C[i-1] + A[i] - M; //递推 C 数组
    sort(C, C+n);
```

```
    long long x1 = C[n/2], ans = 0; //计算 x1
    for(int i = 0; i < n; i++) ans += abs(x1 - C[i]);
    //把 x1 代入，计算转手的总金币数
    printf("%lld\n", ans);
  }
  return 0;
}
```

程序本身并没有太多技巧可言，但需要注意的是 long long 的输入输出。在《入门经典》中我们已经解释过了，%lld 这个占位符并不是跨平台的，比如，Windows 下的 mingw 需要用%I64d 而不是%lld。虽然 cin/cout 没有这个问题，但是本题输入量比较大，cin/cout 会很慢。有两个解决方案。一是自己编写输入输出函数（前面已经给过范例），二是使用 ios::sync_with_stdio(false)，通过关闭 ios 和 stdio 之间的同步来加速，有兴趣的读者可以自行搜索详细信息。

中位数可以在线性时间内求出，但不是本例题的重点（代数分析才是重点），有兴趣的读者可以自行搜索“快速选择”算法的资料。另外，这个程序里的 A 数组实际上是不必保存的，你能去掉它吗？

例题 4　墓地雕塑（Graveyard, NEERC 2006, LA 3708）

在一个周长为 10000 的圆上等距分布着 n 个雕塑。现在又有 m 个新雕塑加入（位置可以随意放），希望所有 $n+m$ 个雕塑在圆周上均匀分布。这就需要移动其中一些原有的雕塑。要求 n 个雕塑移动的总距离尽量小。

【输入格式】

输入包含若干组数据。每组数据仅一行，包含两个整数 n 和 m（$2 \leqslant n \leqslant 1\,000$，$1 \leqslant m \leqslant 1\,000$），即原始的雕塑数量和新加的雕塑数量。输入结束标志为文件结束符（EOF）。

【输出格式】

输入仅一行，为最小总距离，精确到 10^{-4}。

【样例输入】

```
2 1
2 3
3 1
10 10
```

【样例输出】

```
1666.6667
1000.0
1666.6667
0.0
```

【样例解释】

前 3 个样例如图 1-3 所示。白色空心点表示等距点，黑色线段表示已有雕塑。

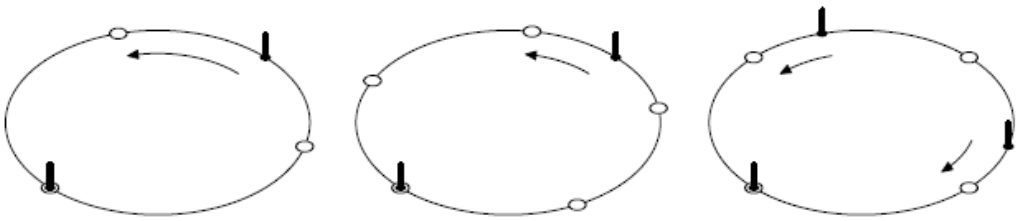


图 1-3

【分析】

请仔细看看样例。3 个样例具有一个共同的特点：有一个雕塑没有移动。如果该特点在所有情况下都成立，则所有雕塑的最终位置（称为“目标点”）实际上已经确定。为了简单起见，我们把没动的那个雕塑作为坐标原点，其他雕塑按照逆时针顺序标上到原点的距离标号，如图 1-4 所示。

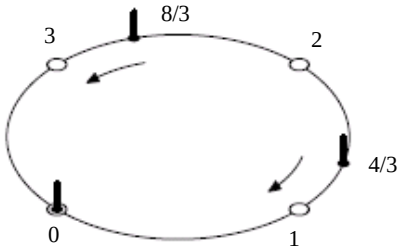


图 1-4

注意，这里的距离并不是真实距离，而是按比例缩小以后的距离。接下来，我们把每个雕塑移动到离它最近的位置。如果没有两个雕像移到相同的位置，那么这样的移动一定是最优的。代码如下。

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

int main() {
  int n, m;
  while(scanf("%d%d", &n, &m) == 2) {
    double ans = 0.0;
    for(int i = 1; i < n; i++) {
      double pos = (double)i / n * (n+m);        //计算每个需要移动的雕塑的坐标
      ans += fabs(pos - floor(pos+0.5)) / (n+m); //累加移动距离
    }
    printf("%.4lf\n", ans*10000);                //等比例扩大坐标
  }
  return 0;
}
```

注意在代码中，坐标为 pos 的雕塑移动到的目标位置是 floor(pos+0.5)，也就是 pos 四舍五入后的结果。这就是坐标缩小的好处。

这个代码很神奇地通过了测试，但其实这个算法有两个小小的“漏洞”：首先，我们不知道是不是一定有一个雕塑没有移动；其次，我们不知道会不会有两个雕塑会移动到相同的位置。如果你对证明不感兴趣，或者已经想到了证明，再或者迫不及待地想阅读更有趣的问题，请直接跳到下一个例题。否则，请继续阅读。

第一个“漏洞”的修补需要证明我们的猜想。证明思路在例题 3 中我们已经展示过了，具体的细节留给读者思考。

第二个“漏洞”有两种修补方法。第一种方法相对较容易实施：由于题目中规定了 $n, m \leqslant 1\,000$，我们只需要在程序里加入一个功能——记录每座雕塑移到的目标位置，就可以用程序判断是否会出现“人多坑少”的情况。这段程序的编写留给读者，这里可以明确地告诉大家：这样的情况确实不会出现。这样，即使无法从理论上证明，也可以确保在题目规定的范围内，我们的算法是严密的。

第二种方法就是直接证明。在我们的程序中，当坐标系缩放之后，坐标为 x 的雕塑被移到了 x 四舍五入后的位置。如果有两个坐标分别为 x 和 y 的雕塑被移到了同一个位置，说明 x 和 y 四舍五入后的结果相同，换句话说，即 x 和 y“很接近”。至于有多接近呢？差距最大的情况不外乎类似于 $x=0.5, y=1.499\,999\ldots$。即便是这样的情况，$y-x$ 仍然小于 1（尽管很接近 1），但这是不可能的，因为新增雕塑之后，相邻雕塑的距离才等于 1，之前的雕塑数目更少，距离应当更大才对。

例题 5　蚂蚁（Piotr’s Ants, UVa 10881）

一根长度为 L 厘米的木棍上有 n 只蚂蚁，每只蚂蚁要么朝左爬，要么朝右爬，速度为 1 厘米/秒。当两只蚂蚁相撞时，二者同时掉头（掉头时间忽略不计）。给出每只蚂蚁的初始位置和朝向，计算 T 秒之后每只蚂蚁的位置。

【输入格式】

输入的第一行为数据组数。每组数据的第一行为 3 个正整数 L, T, n（$0 \leqslant n \leqslant 10\,000$）；以下 n 行每行描述一只蚂蚁的初始位置，其中，整数 x 为蚂蚁距离木棍左端的距离（单位：厘米），字母表示初始朝向（L 表示朝左，R 表示朝右）。

【输出格式】

对于每组数据，输出 n 行，按输入顺序输出每只蚂蚁的位置和朝向（Turning 表示正在碰撞）。在第 T 秒之前已经掉下木棍的蚂蚁（正好爬到木棍边缘的不算）输出 Fell off。

【样例输入】

```
2
10 1 4
1 R
5 R
3 L
10 R
10 2 3
```

```
4 R
5 L
8 R
```

【样例输出】

```
Case #1:
2 Turning
6 R
2 Turning
Fell off

Case #2:
3 L
6 R
10 R
```

【分析】

假设你在远处观察这些蚂蚁的运动，会看到什么？一群密密麻麻的小黑点在移动。由于黑点太小，所以当蚂蚁因碰撞而掉头时，看上去和两个点“对穿而过”没有任何区别，换句话说，如果把蚂蚁看成是没有区别的小点，那么只需独立计算出每只蚂蚁在 T 时刻的位置即可。比如，有 3 只蚂蚁，蚂蚁 1=(1, R)，蚂蚁 2= (3, L)，蚂蚁 3=(4, L)，则两秒钟之后，3 只蚂蚁分别为(3,R)、(1,L)和(2,L)。

注意，虽然从整体上讲，“掉头”等价于“对穿而过”，但对于每只蚂蚁而言并不是这样。蚂蚁 1 的初始状态为(1,R)，因此一定有一只蚂蚁在两秒钟之后处于(3,R)的状态，但这只蚂蚁却不一定是蚂蚁 1。换句话说，我们需要搞清楚目标状态中“谁是谁”。

也许读者已经发现了其中的奥妙：所有蚂蚁的相对顺序是保持不变的，因此把所有目标位置从小到大排序，则从左到右的每个位置对应于初始状态下从左到右的每只蚂蚁。由于原题中蚂蚁不一定按照从左到右的顺序输入，还需要预处理计算出输入中的第 i 只蚂蚁的序号 order[i]。完整代码如下。

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

const int maxn = 10000 + 5;

struct Ant {
  int id;    //输入顺序
  int p;     //位置
  int d;     //朝向。 -1: 左; 0:转身中; 1:右
  bool operator < (const Ant& a) const {
    return p < a.p;
```

```
  }
} before[maxn], after[maxn];

const char dirName[][10] = {"L", "Turning", "R"};

int order[maxn]; //输入的第 i 只蚂蚁是终态中的左数第 order[i]只蚂蚁

int main() {
  int K;
  scanf("%d", &K);
  for(int kase = 1; kase <= K; kase++) {
    int L, T, n;
    printf("Case #%d:\n", kase);
    scanf("%d%d%d", &L, &T, &n);
    for(int i = 0; i < n; i++) {
      int p, d;
      char c;
      scanf("%d %c", &p, &c);
      d = (c == 'L' ? -1 : 1);
      before[i] = (Ant){i, p, d};
      after[i] = (Ant){0, p+T*d, d};    //这里的 id 是未知的
    }

    //计算 order 数组
    sort(before, before+n);
    for(int i = 0; i < n; i++)
      order[before[i].id] = i;

    //计算终态
    sort(after, after+n);
    for(int i = 0; i < n-1; i++)        //修改碰撞中的蚂蚁的方向
      if(after[i].p == after[i+1].p) after[i].d = after[i+1].d = 0;

    //输出结果
    for(int i = 0; i < n; i++) {
      int a = order[i];
      if(after[a].p < 0 || after[a].p > L) printf("Fell off\n");
      else printf("%d %s\n", after[a].p, dirName[after[a].d+1]);
    }
    printf("\n");
  }
  return 0;
}
```

例题 6　立方体成像（Image Is Everything, World Finals 2004, LA 2995）

有一个 $n \times n \times n$ 立方体，其中一些单位立方体已经缺失（剩下部分不一定连通）。每个单位立方体重量为 1 克，且被涂上单一的颜色（即 6 个面的颜色相同）。给出前、左、后、右、顶、底 6 个视图，你的任务是判断这个物体剩下的最大重量。

【输入格式】

输入包含多组数据。每组数据的第一行为一个整数 n（$1 \leqslant n \leqslant 10$）；以下 n 行每行从左到右依次为前、左、后、右、顶、底 6 个视图，每个视图占 n 列，相邻视图中间以一个空格隔开。顶视图的下边界对应于前视图的上边界；底视图的上边界对应于前视图的下边界。在视图中，大写字母表示颜色（不同字母表示不同颜色），句号（.）表示该位置可以看穿（即没有任何立方体）。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出一行，即物体的最大重量（单位：克）。

【样例输入】

```
3
.R. YYR .Y. RYY .Y. .R.
GRB YGR BYG RBY GYB GRB
.R. YRR .Y. RRY .R. .Y.
2
ZZ ZZ ZZ ZZ ZZ ZZ
ZZ ZZ ZZ ZZ ZZ ZZ
0
```

【样例输出】

```
Maximum weight: 11 gram(s)
Maximum weight: 8 gram(s)
```

【分析】

这个问题看上去有点棘手，不过仍然可以找到突破口。比如，能“看穿”的位置所对应的所有单位立方体一定都不存在。再比如，如果前视图的右上角颜色 A 和顶视图的右下角颜色 B 不同，那么对应的格子一定不存在。如图 1-5 所示。

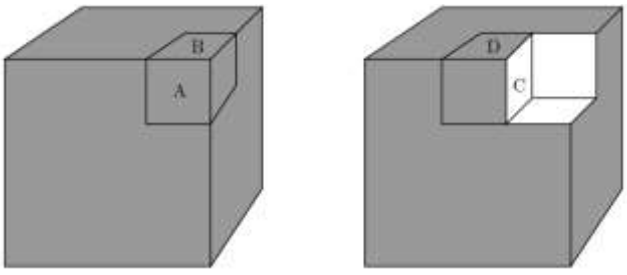


图　1-5

在删除这个立方体之后，我们可能会有新发现：C 和 D 的颜色不同。这样，我们又能

删除一个新的立方体，并暴露出新的表面。当无法继续删除的时候，剩下的立方体就是重量最大的物体。

可能有读者会对上述算法心存疑惑。解释如下：首先不难证明第一次删除是必要的（即被删除的那个立方体不可能存在于任意可行解中），因为只要不删除这个立方体，对应两个视图的“矛盾”将一直存在；接下来，我们用数学归纳法，假设算法的前 k 次删除都是必要的，那么第 $k+1$ 次删除是否也是必要的呢？由刚才的推理，我们不能通过继续删除立方体来消除矛盾，而由归纳假设，已经删除的立方体也不能恢复，因此矛盾无法消除。

下面给出完整代码。

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

#define REP(i,n) for(int i = 0; i < (n); i++)

const int maxn = 10;
int n;
char pos[maxn][maxn][maxn];
char view[6][maxn][maxn];

char read_char() {
  char ch;
  for(;;) {
    ch = getchar();
    if((ch >= 'A' && ch <= 'Z') || ch == '.') return ch;
  }
}

void get(int k, int i, int j, int len, int &x, int &y, int &z)
{
  if (k == 0) { x = len; y = j; z = i; }
  if (k == 1) { x = n - 1 - j; y = len; z = i; }
  if (k == 2) { x = n - 1 - len; y = n - 1 - j; z = i; }
  if (k == 3) { x = j; y = n - 1 - len; z = i; }
  if (k == 4) { x = n - 1 - i; y = j; z = len; }
  if (k == 5) { x = i; y = j; z = n - 1 - len; }
}

int main() {
```

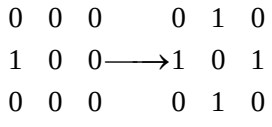
```
  while(scanf("%d", &n) == 1 && n) {
    REP(i,n) REP(k,6) REP(j,n) view[k][i][j] = read_char();
    REP(i,n) REP(j,n) REP(k,n) pos[i][j][k] = '#';

    REP(k,6) REP(i,n) REP(j,n) if (view[k][i][j] == '.')
      REP(p,n) {
        int x, y, z;
        get(k, i, j, p, x, y, z);
        pos[x][y][z] = '.';
      }

    for(;;) {
      bool done = true;
      REP(k,6) REP(i,n) REP(j,n) if (view[k][i][j] != '.') {
        REP(p,n) {
          int x, y, z;
          get(k, i, j, p, x, y, z);
          if (pos[x][y][z] == '.') continue;
          if (pos[x][y][z] == '#') {
            pos[x][y][z] = view[k][i][j];
            break;
          }
          if (pos[x][y][z] == view[k][i][j]) break;
          pos[x][y][z] = '.';
          done = false;
        }
      }
      if(done) break;
    }

    int ans = 0;
    REP(i,n) REP(j,n) REP(k,n)
      if (pos[i][j][k] != '.') ans ++;

    printf("Maximum weight: %d gram(s)\n", ans);
  }
  return 0;
}
```

程序用了一个 get 函数来表示第 *k* 个视图中，第 *i* 行 *j* 列、深度为 len 的单位立方体在原立方体中的坐标(*x*,*y*,*z*)，另外还使用了宏 REP 精简程序。尽管用宏缩短代码在很多时候会降低程序可读性，但本题却不会（如果到处都是 for 循环，反而容易令人犯晕）。

1.2　问题求解常见策略

例题 7　偶数矩阵（Even Parity, UVa 11464）

给你一个 $n \times n$ 的 01 矩阵（每个元素非 0 即 1），你的任务是把尽量少的 0 变成 1，使得每个元素的上、下、左、右的元素（如果存在的话）之和均为偶数。比如，如图 1-6（a）所示的矩阵至少要把 3 个 0 变成 1，最终如图 1-6（b）所示，才能保证其为偶数矩阵。



（a）　　（b）

图　1-6

【输入格式】

输入的第一行为数据组数 T（$T \leqslant 30$）。每组数据的第一行为正整数 n（$1 \leqslant n \leqslant 15$）；接下来的 n 行每行包含 n 个非 0 即 1 的整数，相邻整数间用一个空格隔开。

【输出格式】

对于每组数据，输出被改变的元素的最小个数。如果无解，应输出−1。

【分析】

也许最容易想到的方法就是枚举每个数字“变”还是“不变”，最后判断整个矩阵是否满足条件。遗憾的是，这样做最多需要枚举 $2^{255} \approx 5 \times 10^{67}$ 种情况，实在难以承受。

注意到 n 只有 15，第一行只有不超过 $2^{15}=32\,768$ 种可能，所以第一行的情况是可以枚举的。接下来根据第一行可以完全计算出第二行，根据第二行又能计算出第三行（想一想，如何计算），以此类推，这样，总时间复杂度即可降为 $O(2^n \times n^2)$。代码如下。

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

const int maxn = 20;
const int INF = 1000000000;
int n, A[maxn][maxn], B[maxn][maxn];

int check(int s) {
  memset(B, 0, sizeof(B));
  for(int c = 0; c < n; c++) {
    if(s & (1<<c)) B[0][c] = 1;
```

```
    else if(A[0][c] == 1) return INF; //1 不能变成 0
  }
  for(int r = 1; r < n; r++)
    for(int c = 0; c < n; c++) {
      int sum = 0; //元素 B[r-1][c]的上、左、右 3 个元素之和
      if(r > 1) sum += B[r-2][c];
      if(c > 0) sum += B[r-1][c-1];
      if(c < n-1) sum += B[r-1][c+1];
      B[r][c] = sum % 2;
      if(A[r][c] == 1 && B[r][c] == 0) return INF; //1 不能变成 0
    }
  int cnt = 0;
  for(int r = 0; r < n; r++)
    for(int c = 0; c < n; c++) if(A[r][c] != B[r][c]) cnt++;
  return cnt;
}

int main() {
  int T;
  scanf("%d", &T);
  for(int kase = 1; kase <= T; kase++) {
    scanf("%d", &n);
    for(int r = 0; r < n; r++)
      for(int c = 0; c < n; c++) scanf("%d", &A[r][c]);

    int ans = INF;
    for(int s = 0; s < (1<<n); s++)
      ans = min(ans, check(s));
    if(ans == INF) ans = -1;
    printf("Case %d: %d\n", kase, ans);
  }
  return 0;
}
```

例题 8　彩色立方体（Colored Cubes, Tokyo 2005, LA 3401）

有 n 个带颜色的立方体，每个面都涂有一种颜色。要求重新涂尽量少的面，使得所有立方体完全相同。两个立方体相同的含义是：存在一种旋转方式，使得两个立方体对应面的颜色相同。

【输入格式】

输入包含多组数据。每组数据的第一行为正整数 n（$1 \leqslant n \leqslant 4$）；以下 n 行每行 6 个字符串，分别为立方体编号为 1~6 的面的颜色（由小写字母和减号组成，不超过 24 个字符）。

输入结束标志为 n=0。立方体的 6 个面的编号如图 1-7 所示。

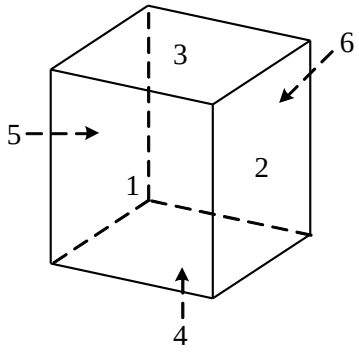


图　1-7

【输出格式】

对于每组数据，输出重新涂色的面数的最小值。

【分析】

立方体只有 4 个，暴力法应该可行。不过不管怎样“暴力”，首先得搞清楚一个立方体究竟有几种不同的旋转方式。

为了清晰起见，我们借用机器人学中的术语，用姿态（pose）来代替口语中的旋转方法。假设 6 个面的编号为 1~6，从中选一个面作为“顶面”，然后在剩下的 4 个面中选一个作为“正面”，则其他面都可以唯一确定，因此有 6×4=24 种姿态。

在代码中，每种姿态对应一个全排列 P。其中，$P[i]$表示编号 i 所在的位置（1 表示正面，2 表示右面，3 表示顶面等，如图 1-8 所示）。如图 1-9 所示的姿态称为标准姿态，用排列{1,2,3,4,5,6}表示，因为 1 在正面，2 在右面，3 在顶面等。

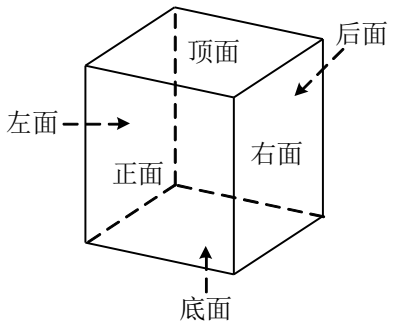


图　1-8

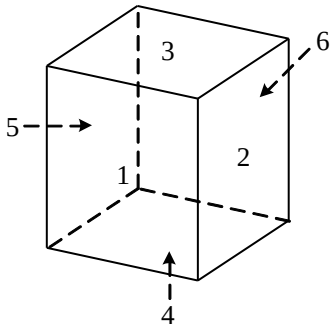


图　1-9

图 1-10 是标准姿态向左旋转后得到的。对应的排列是{5,1,3,4,6,2}。

接下来有两种方法。一种方法是手工找出 24 种姿态对应的排列，编写到代码中。显然，这种方法比较耗时，且容易出错，不推荐使用。下面的方法可以用程序找出这 24 种排列，而且不容易出错。除了刚才写出的标准姿态向左翻之外，我们再写出标准姿态向上翻所对应的排列：{3,2,6,1,5,4}，如图 1-11 所示。

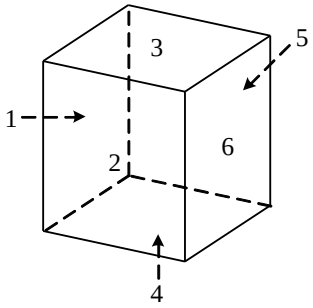


图 1-10

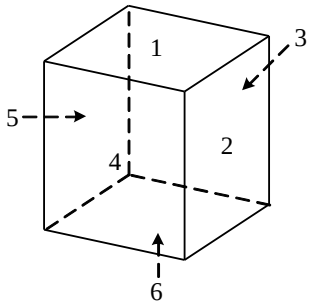


图 1-11

注意到旋转是可以组合的，比如，图 1-11 标准姿态先向左转再向上翻就是 5→5, 1→3, 3→6, 4→1, 6→4, 2→2，即{5, 3, 6, 1, 4, 2}。因此，有了这两种旋转方式，我们就可以构造出所有 24 种姿态了（均为从标准姿态开始旋转）。

1 在顶面的姿态：向上翻 1 次（此时 1 在顶面），然后向左转 0~3 次。

2 在顶面的姿态：向左转 1 次（此时 2 在顶面），向上翻 1 次，然后向左转 0~3 次。

3 在顶面的姿态：（3 本来就在顶面）向左转 0~3 次。

4 在顶面的姿态：向上翻 2 次（此时 4 在顶面），然后向左转 0~3 次。

5 在顶面的姿态：向左转 2 次，向上翻一次（此时 5 在顶面），然后向左转 0~3 次。

6 在顶面的姿态：向左转 3 次，向上翻一次（此时 6 在顶面），然后向左转 0~3 次。

这段代码应该写在哪里呢？一种方法是直接手写在最终的程序中，但是一旦这部分代码出错，非常难调；另一种方法是写到一个独立程序中，用它生成 24 种姿态对应的排列，而在最终程序中直接使用常量表。生成排列表的程序如下（注意，在代码中编号为 0~5，而非 1~6）。

```
#include<cstdio>
#include<cstring>

int left[] = {4, 0, 2, 3, 5, 1};
int up[] = {2, 1, 5, 0, 4, 3};

//按照排列 T 旋转姿态 p
void rot(int* T, int* p) {
  int q[6];
  memcpy(q, p, sizeof(q));
  for(int i = 0; i < 6; i++) p[i] = T[q[i]];
}

void enumerate_permutations() {
  int p0[6] = {0, 1, 2, 3, 4, 5};
  printf("int dice24[24][6] = {\n");
  for(int i = 0; i < 6; i++) {
```

```
    int p[6];
    memcpy(p, p0, sizeof(p0));
    if(i == 0) rot(up, p);
    if(i == 1) { rot(left, p); rot(up, p); }
    if(i == 3) { rot(up, p); rot(up, p); }
    if(i == 4) { rot(left, p); rot(left, p); rot(up, p); }
    if(i == 5) { rot(left, p); rot(left, p); rot(left, p); rot(up, p); }
    for(int j = 0; j < 4; j++) {
      printf("{%d, %d, %d, %d, %d, %d},\n", p[0], p[1], p[2], p[3], p[4], p[5]);
      rot(left, p);
    }
  }
  printf("};\n");
}

int main() {
  enumerate_permutations();
  return 0;
}
```

下面让我们来看看如何“暴力”。一种方法是枚举最后那个“相同的立方体”的每个面是什么，然后对于每个立方体，看看哪种姿态需要重新涂色的面最少。但由于 4 个立方体最多可能会有 24 种不同的颜色，最多需要枚举 24^6 种“最后的立方体”，情况有些多。

另一种方法是先枚举每个立方体的姿态（第一个作为“参考系”，不用旋转），然后对于 6 个面，分别选一个出现次数最多的颜色作为“标准”，和它不同的颜色一律重涂。由于每个立方体的姿态有 24 种，3 个立方体（别忘了第一个不用旋转）的姿态组合一共有 24^3 种，比第一种方法要好。程序如下（程序头部是生成的常量表，为了节省篇幅，合并了一些行）。

```
int dice24[24][6] = {
{2, 1, 5, 0, 4, 3},{2, 0, 1, 4, 5, 3},{2, 4, 0, 5, 1, 3},{2, 5, 4, 1, 0, 3},{4,
2, 5, 0, 3, 1},
{5, 2, 1, 4, 3, 0},{1, 2, 0, 5, 3, 4},{0, 2, 4, 1, 3, 5},{0, 1, 2, 3, 4, 5},{4,
0, 2, 3, 5, 1},
{5, 4, 2, 3, 1, 0},{1, 5, 2, 3, 0, 4},{5, 1, 3, 2, 4, 0},{1, 0, 3, 2, 5, 4},{0,
4, 3, 2, 1, 5},
{4, 5, 3, 2, 0, 1},{3, 4, 5, 0, 1, 2},{3, 5, 1, 4, 0, 2},{3, 1, 0, 5, 4, 2},{3,
0, 4, 1, 5, 2},
{1, 3, 5, 0, 2, 4},{0, 3, 1, 4, 2, 5},{4, 3, 0, 5, 2, 1},{5, 3, 4, 1, 2, 0},
};

#include<cstdio>
#include<cstring>
```

```
#include<string>
#include<vector>
#include<algorithm>
using namespace std;

const int maxn = 4;
int n, dice[maxn][6], ans;

vector<string> names;
int ID(const char* name) {
  string s(name);
  int n = names.size();
  for(int i = 0; i < n; i++)
    if(names[i] == s) return i;
  names.push_back(s);
  return n;
}

int r[maxn], color[maxn][6];    //每个立方体的旋转方式和旋转后各个面的颜色

void check() {
  for(int i = 0; i < n; i++)
    for(int j = 0; j < 6; j++) color[i][dice24[r[i]][j]] = dice[i][j];

  int tot = 0;                  //需要重新涂色的面数
  for(int j = 0; j < 6; j++) {  //考虑每个面
    int cnt[maxn*6];            //每种颜色出现的次数
    memset(cnt, 0, sizeof(cnt));
    int maxface = 0;
    for(int i = 0; i < n; i++)
      maxface = max(maxface, ++cnt[color[i][j]]);
    tot += n - maxface;
  }
  ans = min(ans, tot);
}

void dfs(int d) {
  if(d == n) check();
  else for(int i = 0; i < 24; i++) {
    r[d] = i;
    dfs(d+1);
  }
```

```
}

int main() {
  while(scanf("%d", &n) == 1 && n) {
    names.clear();
    for(int i = 0; i < n; i++)
      for(int j = 0; j < 6; j++) {
        char name[30];
        scanf("%s", name);
        dice[i][j] = ID(name);
      }
    ans = n*6;   //上界：所有面都重涂色
    r[0] = 0;    //第一个立方体不旋转
    dfs(1);
    printf("%d\n", ans);
  }
  return 0;
}
```

例题 9　中国麻将（Chinese Mahjong, UVa 11210）

麻将是一个中国原创的 4 人玩的游戏。这个游戏有很多变种，但本题只考虑一种有 136 张牌的玩法。这 136 张牌所包含的内容如下。

饼（筒）牌：每张牌包括一系列点，每个点代表一个铜钱，如图 1-12 所示。本题中用 1T、2T、3T、4T、5T、6T、7T、8T、9T 表示。

图　1-12

索（条）牌：每张牌由一系列竹棍组成，每根棍代表一挂铜钱，如图 1-13 所示。本题中用 1S、2S、3S、4S、5S、6S、7S、8S、9S 表示。

图　1-13

万牌：每张牌代表一万枚铜钱，如图 1-14 所示。本题中用 1W、2W、3W、4W、5W、6W、7W、8W、9W 表示。

图　1-14

风牌：东、南、西、北风，如图 1-15 所示。本题中用 DONG、NAN、XI、BEI 表示。

箭牌：中、发、白，如图 1-16 所示。本题中用 ZHONG、FA、BAI 表示。

图　1-15

图　1-16

总共有 9×3+4+3=34 种牌，每种 4 张，一共有 136 张牌。

其实麻将中还有如图 1-17 所示的 8 张花牌，所以共有 136 + 8 = 144 张牌，但是本题中不予考虑。

图　1-17

中国麻将的规则十分复杂，本题中只需考虑部分规则。在本题中，手牌（即每个人手里的牌）总是有 13 张。如果多了某张牌以后，整副牌可以拆成一个将（两张相同的牌）、0 个或多个刻子（3 张相同的牌）和 0 个或多个顺子（3 张同花相连的牌。注意，风牌和箭牌不能形成顺子），我们就说这手牌“听”这张牌，即拿到那张牌以后就赢了，称为“和”（实战中还要考虑番数和特殊和法，在本题中可以忽略）。

比如，如图 1-18 所示的这手牌：

图　1-18

听牌  、  和  ，即 1S、FA 和 4S。听牌  的原因是：“发”做将，另有 3 个顺子（1S2S3S, 1S2S3S, 2S3S4S）。

【输入格式】

输入数据最多 50 组。每组数据由一行 13 张牌给出，输入保证给出的牌是合法的。输入结束标记为一行单个 0。

【输出格式】

对于每组数据，输出所有“听”的牌，按照描述中的顺序列出（1T-9T，1S-9S，1W-9W，DONG，NAN，XI，BEI，ZHONG，FA，BAI）。每张牌最多被列出一次。如果没有“听”牌，输出 Not ready。

【分析】

如果您和笔者一样对麻将很熟悉，不妨回忆一下自己平时打麻将时，是如何知道自己有没有听牌的。虽然多数情况都容易判断，但对于一些复杂的情况，新手容易看不出自己“听”牌了，或者看不全所有“听”的牌，而麻将老手却可以。原因在于，麻将老手擅长把手里的牌按照不同的方式进行组合。在程序里，我们也需要用一点“暴力”来枚举所有

可能的组合方式。

一共只有 34 种牌，因此可以依次判断是否“听”这些牌。比如，为了判断是否“听”一万，只需要判断自己拿到这张一万后是否可以和牌。这样，问题就转化为了：给定 14 张牌，判断是否可以和牌。为此，我们可以递归求解：首先选两张牌作为“将”，然后每次选 3 张作为刻子或者顺子。如图 1-19 所示，即为一次递归求解的过程。

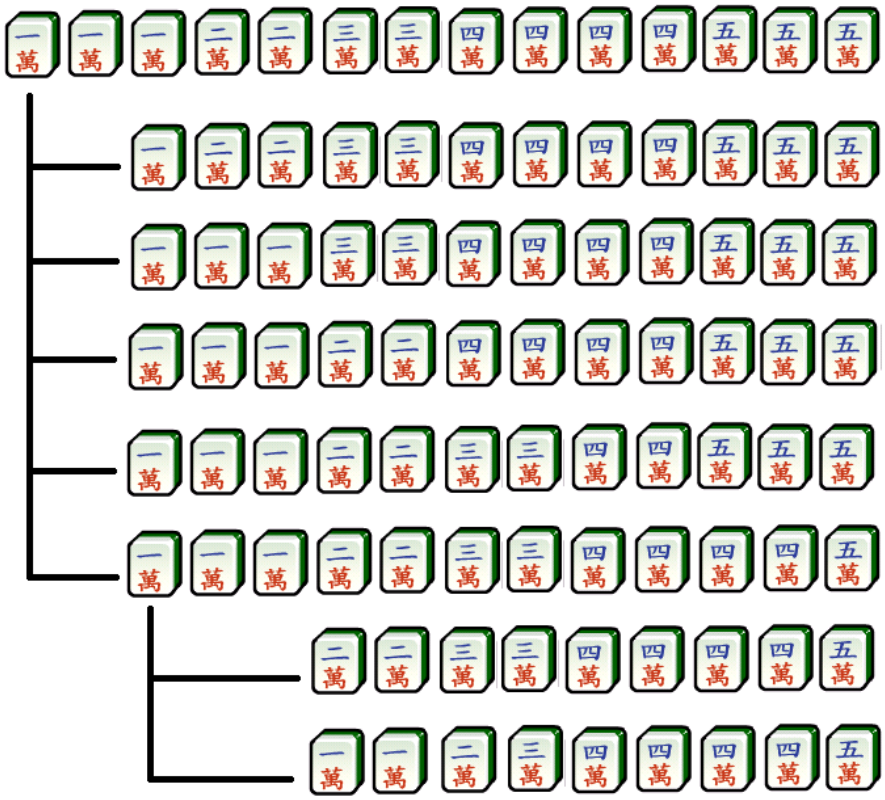


图　1-19

选将有 5 种方法（一二三四五万都可以做将）。如果选五万做将，一万要么属于一个刻子，要么属于一个顺子（二三四）。注意，这时不必考虑其他牌是如何形成刻子或者顺子的，否则会出现重复枚举（想一想，为什么）。

为了快速选出将、刻子和顺子，我们用一个 34 维向量来表示状态，即每种牌所剩的张数。除了第一次直接枚举将牌之外，每次只需要考虑编号最小的牌，看它能否形成刻子或者顺子（一定是以它作为最小牌。想一想，为什么），并且递归判断。本题唯一的陷阱是：每一种牌都只有 4 张，所以 1S1S1S1S 是不“听”任何牌的。

完整代码如下。

```
#include<stdio.h>
#include<string.h>

const char* mahjong[] = {
```

```
    "1T","2T","3T","4T","5T","6T","7T","8T","9T",
    "1S","2S","3S","4S","5S","6S","7S","8S","9S",
    "1W","2W","3W","4W","5W","6W","7W","8W","9W",
    "DONG","NAN","XI","BEI",
    "ZHONG","FA","BAI"
    };

    int convert(char *s){                          //只在预处理时调用，因此速度无关紧要
      for(int i = 0; i < 34; i++)
        if(strcmp(mahjong[i], s) == 0) return i;
      return -1;
    }

    int c[34];
    bool search(int dep){                          //回溯法递归过程
      int i;
      for(i = 0; i < 34; i++) if (c[i] >= 3){   //刻子
        if(dep == 3) return true;
        c[i] -= 3;
        if(search(dep+1)) return true;
        c[i] += 3;
      }
      for(i = 0; i <= 24; i++) if (i % 9 <= 6 && c[i] >= 1 && c[i+1] >= 1 && c[i+2] >=
1){                                                //顺子
        if(dep == 3) return true;
        c[i]--; c[i+1]--; c[i+2]--;
        if(search(dep+1)) return true;
        c[i]++; c[i+1]++; c[i+2]++;
      }
      return false;
    }

    bool check(){
      int i;
      for(i = 0; i < 34; i++)
        if(c[i] >= 2){                             //将牌
          c[i] -= 2;
          if(search(0)) return true;
          c[i] += 2;
      }
      return false;
    }
```

```
int main(){
  int caseno = 0, i, j;
  bool ok;
  char s[100];
  int mj[15];

  while(scanf("%s", &s) == 1){
    if(s[0] == '0') break;
    printf("Case %d:", ++caseno);
    mj[0] = convert(s);
    for(i = 1; i < 13; i++){
      scanf("%s", &s);
      mj[i] = convert(s);
    }
    ok = false;
    for(i = 0; i < 34; i++){
      memset(c, 0, sizeof(c));
      for(j = 0; j < 13; j++) c[mj[j]]++;
      if(c[i] >= 4) continue;    //每种牌最多只有 4 张
      c[i]++;                    //假设拥有这张牌
      if(check()){               //如果“和”了
        ok = true;               //说明听这张牌
        printf(" %s", mahjong[i]);
      }
      c[i]--;
    }
    if(!ok) printf(" Not ready");
    printf("\n");
  }
  return 0;
}
```

例题 10　正整数序列（Help is needed for Dexter, UVa 11384）

给定正整数 n，你的任务是用最少的操作次数把序列 1, 2, …, n 中的所有数都变成 0。每次操作可从序列中选择一个或多个整数，同时减去一个相同的正整数。比如，1,2,3 可以把 2 和 3 同时减小 2，得到 1,0,1。

【输入格式】

输入包含多组数据。每组仅一行，为正整数 n（$n \leqslant 10^9$）。输入结束标志为文件结束符（EOF）。

【输出格式】

对于每组数据，输出最少操作次数。

【分析】

拿到这道题目之后，最好的方式是自己试一试。经过若干次尝试和总结后，不难发现第一步的最好方式如图 1-20 所示。

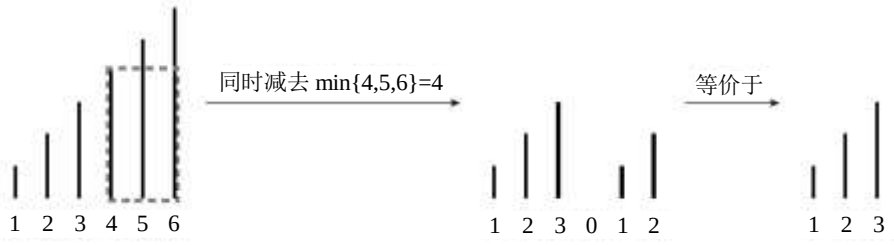


图 1-20

换句话说，当 n=6 的时候留下 1, 2, 3，而把 4,5,6 同时减去 min{4,5,6}=4 得到序列 1, 2, 3, 0, 1, 2，它等价于 1, 2, 3（想一想，为什么）。换句话说，$f(6)=f(3)+1$。

一般地，为了平衡，我们保留 1~n/2，把剩下的数同时减去 n/2+1，得到序列 1, 2, …,n/2, 0,1,…,(n−1)/2，它等价于 1, 2, …, n/2，因此 $f(n)=f(n/2)+1$。边界是 $f(1)=1$。代码如下。

```
#include<cstdio>
int f(int n) {
  return n == 1 ? 1 : f(n/2) + 1;
}

int main() {
  int n;
  while(scanf("%d", &n) == 1)
    printf("%d\n", f(n));
  return 0;
}
```

例题 11　新汉诺塔问题（A Different Task, UVa 10795）

标准的汉诺塔上有 n 个大小各异的盘子。给定一个初始局面（如图 1-21 所示），求它到给定目标局面（如图 1-22 所示）至少需要多少步。移动规则如下：一次只能移动一个盘子；在移动一个盘子之前，必须把压在上面的其他盘子先移走；编号大的盘子不得压在编号小的盘子上。

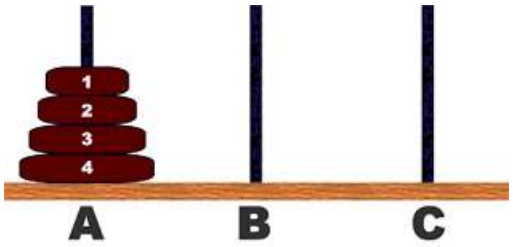


图 1-21

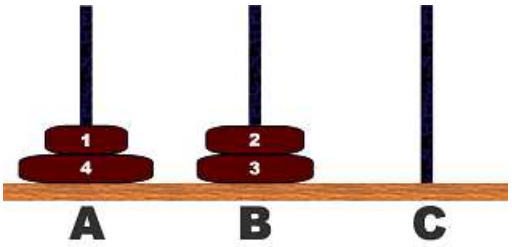


图 1-22

【输入格式】

输入包含不超过 100 组数据。每组数据的第一行为正整数 n（$1 \leqslant n \leqslant 60$）；第二行包含 n 个 1~3 的整数，即初始局面中每个盘子所在的柱子编号；第三行和第二行格式相同，为目标局面。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出最少步数。

【分析】

考虑编号最大的盘子。如果这个盘子在初始局面和目标局面中位于同一根柱子上，那么根本不需要移动它，而如果移动了，反而不可能是最优解（想一想，为什么）。这样，我们可以在初始局面和目标局面中，找出所在柱子不同的盘子中编号最大的一个，设为 k，那么 k 必须移动。

让我们设想一下，移动 k 之前的一瞬间，柱子上的情况吧。假设盘子 k 需要从柱子 1 移动到柱子 2。由于编号比 k 大的盘子不需要移动，而且也不会碍事，所以我们直接把它们看成不存在；编号比 k 小的盘子既不能在柱子 1 上，也不能在柱子 2 上，因此只能在柱子 3 上。换句话说，这时柱子 1 只有盘子 k，柱子 2 为空，柱子 3 从上到下依次是盘子 1, 2, 3, …, k−1（再次提醒：我们已经忽略了编号大于 k 的盘子）。我们把这个局面称为参考局面。

由于盘子的移动是可逆的，根据对称性，我们只需要求出初始局面和目标局面移动成参考局面的步数之和，然后加 1（移动盘子 k）即可。换句话说，我们需要写一个函数 $f(P, i,$ final)，表示已知各盘子的初始柱子编号数组为 P（具体来说，$P[i]$代表盘子 i 的柱子编号），把盘子 1, 2, 3, …, i 全部移到柱子 final 所需的步数，则本题的答案就是 f(start, k−1, 6−start[k]−finish[k]) + f(finish, k−1, 6−start[k]−finish[k]) + 1。其中，start[i]和 finish[i]是本题输入中盘子 i 的初始柱子和目标柱子，k 是上面所说的“必须移动的编号最大的盘子”的编号。我们把柱子编号为 1, 2, 3，所以“除了柱子 x 和柱子 y 之外的那个柱子”编号为 6−x−y。

如何计算 $f(P, i,$ final)呢？推理和刚才类似。假设 $P[i]$=final，那么 $f(P,i,$final)=$f(P,i-1,$final)；否则需要先把前 i−1 个盘子挪到 6−$P[i]$−final 这个盘子做中转，然后把盘子 i 移动到柱子 final，最后把前 i−1 个盘子从中转的柱子移到目标柱子 final。注意，最后一个步骤是把 i−1 个盘子从一个柱子整体移到另一个柱子，根据汉诺塔问题的经典结论，这个步骤需要 $2^{i-1}-1$ 步，加上移动盘子 i 的那一步，一共需要 2^{i-1} 步。换句话说，当 $P[i]$不等于 final 的时候，$f(P,i,$final)=$f(P,$i−1,6−$P[i]$−final)+2^{i-1}。

最后，注意答案需要用 long long 保存（想一想，为什么）。代码如下。

```
#include<cstdio>

long long f(int* P, int i, int final) {
  if(i == 0) return 0;
  if(P[i] == final) return f(P, i-1, final);
  return f(P, i-1, 6-P[i]-final) + (1LL << (i-1));
}
```

```
const int maxn = 60 + 10;
int n, start[maxn], finish[maxn];

int main() {
  int kase = 0;
  while(scanf("%d", &n) == 1 && n) {
    for(int i = 1; i <= n; i++) scanf("%d", &start[i]);
    for(int i = 1; i <= n; i++) scanf("%d", &finish[i]);
    int k = n;
    while(k >= 1 && start[k] == finish[k]) k--;

    long long ans = 0;
    if(k >= 1) {
      int other = 6-start[k]-finish[k];
      ans = f(start, k-1, other) + f(finish, k-1, other) + 1;
    }
    printf("Case %d: %lld\n", ++kase, ans);
  }
  return 0;
}
```

例题 12　组装电脑（Assemble, NWERC 2007, LA 3971）

你有 b 块钱，想要组装一台电脑。给出 n 个配件各自的种类、品质因子和价格，要求每种类型的配件各买一个，总价格不超过 b，且“品质最差配件”的品质因子应尽量大。

【输入格式】

输入的第一行为测试数据组数 T（$T \leqslant 100$）。每组数据的第一行为两个正整数 n（$1 \leqslant n \leqslant 1\,000$）和 b（$1 \leqslant b \leqslant 10^9$），即配件的数目和预算；以下 n 行每行描述一个配件，依次为种类、名称、价格和品质因子。其中，价格为不超过 10^6 的非负整数；品质因子是不超过 10^9 的非负整数（越大越好）；种类和名称则由不超过 20 个字母、数字和下划线组成。输入保证总是有解。

【输出格式】

对于每组数据，输出配件最小品质因子的最大值。

【分析】

在《入门经典》一书中，我们曾提到过，解决“最小值最大”的常用方法是二分答案。假设答案为 x，如何判断这个 x 是最小还是最大呢？删除品质因子小于 x 的所有配件，如果可以组装出一台不超过 b 元的电脑，那么标准答案 ans $\geqslant x$，否则 ans $< x$。

如何判断是否可以组装出满足预算约束的电脑呢？很简单，每一类配件选择最便宜的一个即可。如果这样选都还超预算的话，就不可能有解了。代码如下。

```
#include<cstdio>
#include<string>
```

```
#include<vector>
#include<map>
using namespace std;

int cnt;        //组件的类型数
map<string,int> id;
int ID(string s) {
  if(!id.count(s)) id[s] = cnt++;
  return id[s];
}

const int maxn = 1000 + 5;

struct Component {
  int price;
  int quality;
};
int n, b;       //组件的数目，预算
vector<Component> comp[maxn];

//品质因子不小于 q 的组件能否组装成一个不超过 b 元的电脑
bool ok(int q) {
  int sum = 0;
  for(int i = 0; i < cnt; i++) {
    int cheapest = b+1, m = comp[i].size();
    for(int j = 0; j < m; j++)
      if(comp[i][j].quality >= q) cheapest = min(cheapest, comp[i][j].price);
    if(cheapest == b+1) return false;
    sum += cheapest;
    if(sum > b) return false;
  }
  return true;
}

int main() {
  int T;
  scanf("%d", &T);
  while(T--) {
    scanf("%d%d", &n, &b);

    cnt = 0;
    for(int i = 0; i < n; i++) comp[i].clear();
```

```
    id.clear();

    int maxq = 0;
    for(int i = 0; i < n; i++) {
      char type[30], name[30];
      int p, q;
      scanf("%s%s%d%d", type, name, &p, &q);
      maxq = max(maxq, q);
      comp[ID(type)].push_back((Component){p, q});
    }

    int L = 0, R = maxq;
    while(L < R) {
      int M = L + (R-L+1)/2;
      if(ok(M)) L = M; else R = M-1;
    }
    printf("%d\n", L);
  }
  return 0;
}
```

例题 13　派（Pie, NWERC 2006, LA 3635）

有 F+1 个人来分 N 个圆形派，每个人得到的必须是一整块派，而不是几块拼在一起，且面积要相同。求每个人最多能得到多大面积的派（不必是圆形）。

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为两个整数 N 和 F（$1 \leqslant N, F \leqslant 10\,000$）；第二行为 N 个整数 r_i（$1 \leqslant r_i \leqslant 10\,000$），即各个派的半径。

【输出格式】

对于每组数据，输出每人得到的派的面积的最大值，精确到 10^{-3}。

【分析】

这个问题并不是“最小值最大”问题，但仍然可以采用二分答案方法，把问题转化为“是否可以让每人得到一块面积为 x 的派”。这样的转化相当于多了一个条件，然后求解目标变成了“看看这些条件是否相互矛盾”。

会有怎样的矛盾呢？只有一种矛盾：x 太大，满足不了所有的 F+1 个人。这样，我们只需要算算一共可以切多少份面积为 x 的派，然后看看这个数目够不够 F+1 即可。因为派是不可以拼起来的，所以一个半径为 r 的派只能切出$[\pi r^2/x]$个派（其他部分就浪费了），把所有圆形派能切出的份数加起来即可。代码如下。

```
#include<cstdio>
#include<cmath>
#include<algorithm>
```

```
using namespace std;

const double PI = acos(-1.0);
const int maxn = 10000 + 5;

int n, f;
double A[maxn];

bool ok(double area) {
  int sum = 0;
  for(int i = 0; i < n; i++) sum += floor(A[i] / area);
  return sum >= f+1;
}

int main() {
  int T;
  scanf("%d", &T);
  while(T--) {
    scanf("%d%d", &n, &f);
    double maxa = -1;
    for(int i = 0; i < n; i++) {
      int r;
      scanf("%d", &r);
      A[i] = PI*r*r; maxa = max(maxa, A[i]);
    }
    double L = 0, R = maxa;
    while(R-L > 1e-5) {
      double M = (L+R)/2;
      if(ok(M)) L = M; else R = M;
    }
    printf("%.5lf\n", L);
  }
  return 0;
}
```

例题 14　填充正方形（Fill the Square, UVa 11520）

在一个 $n \times n$ 网格中填了一些大写字母，你的任务是把剩下的格子中也填满大写字母，使得任意两个相邻格子（即有公共边的格子）中的字母不同。如果有多种填法，则要求按照从上到下、从左到右的顺序把所有格子连接起来得到的字符串的字典序应该尽量小。

【输入格式】

输入的第一行为测试数据组数 T。每组数据的第一行为整数 n（$n \leqslant 10$），即网格的行数和列数；以下 n 行每行 n 个字符，表示整个网格。为了清晰起见，本题用小数点表示没

有填字母的格子。

【输出格式】

对于每组数据，输出填满字母后的网格。

【样例输入】

```
2
3
...
...
...
3
...
A..
...
```

【样例输出】

```
Case 1:
ABA
BAB
ABA
Case 2:
BAB
ABA
BAB
```

【分析】

首先说点题外话。一道题当可能有多个解时，为了确保答案唯一（比如，命题者不想写“输出检查器”，或者为了加大难度），题目通常会加上一些限制条件，其中“字典序最小”就是一个很常见的要求。

所谓“字典序”，就是“在字典中的顺序”。字典中的单词是如何排列的呢？首先按照第一个字母排序，即所有以 a 开头的单词排在以 b 开头的单词前面，而以 b 开头的单词排在以 c 开头的单词前面，以此类推。把这种方法扩展一下：对于任意两个序列，我们先比较第一个元素，再比较第二个元素……直到有一个元素不同，那么此元素小的序列，其字典序也小。剩下的元素全部不比较。注意，如果比较过程中恰好有一个序列结束，那么该序列较小。如果两个序列同时结束，说明两个序列完全相等，字典序自然也相等。下面是比较两个整数序列字典序的代码。

```
bool lexicographicallySmaller(vector<int> a, vector<int> b) {
  int n = a.size();
  int m = b.size();
  int i;
  for(i = 0; i < n && i < m; i++)
```

```
    if(a[i] < b[i]) return true;
    else if(b[i] < a[i]) return false;
  return (i == n && i < m);
}
```

不难发现，对于定义了“小于”运算符的任意数据类型，由该类型元素组成的序列的字典序的比较方法是完全一样的。这样，我们可以把上述函数模板化。

```
template<class T>
bool lexicographicallySmaller(vector<T> a, vector<T> b) {
  int n = a.size();
  int m = b.size();
  int i;
  for(i = 0; i < n && i < m; i++)
    if(a[i] < b[i]) return true;
    else if(b[i] < a[i]) return false;
  return (i == n && i < m);
}
```

除了阴影部分之外，这份代码和前面的代码完全一样。有了模板函数，不管你定义的是 vector<int>a, b 还是 vector<string>a, b，甚至是 vector<vector<int> >x, y，全部都可以用 if(lexicographicallySmaller(a,b)) … 的方式直接使用上述函数，而不必针对各种类型各写一个函数。

既然只有序列才有字典序，题目中的这句“从上到下、从左到右”就不难理解了。它的意思是首先把每行看成一个字符串，然后从上到下顺次连接，要求得到的这个长长的字符串的字典序最小。

根据字典序的定义，我们可以从上到下、从左到右一位一位地求：先满足第一个元素最小，再满足第二个元素最小，以此类推。落实到本题中，我们只需从左到右、从上到下依次给所有的空格填上最小可能的字母即可，代码如下所示。

```
#include<cstdio>
#include<cstring>
const int maxn = 10 + 5;
char grid[maxn][maxn];
int n;
int main() {
  int T;
  scanf("%d", &T);
  for(int kase = 1; kase <= T; kase++) {
    scanf("%d", &n);
    for(int i = 0; i < n; i++) scanf("%s", grid[i]);
    for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++) if(grid[i][j] == '.') {//没填过的字母才需要填
```

```
        for(char ch = 'A'; ch <= 'Z'; ch++) {        //按照字典序依次尝试
          bool ok = true;
          if(i>0 && grid[i-1][j] == ch) ok = false; //和上面的字母冲突
          if(i<n-1 && grid[i+1][j] == ch) ok = false;
          if(j>0 && grid[i][j-1] == ch) ok = false;
          if(j<n-1 && grid[i][j+1] == ch) ok = false;
          if(ok) { grid[i][j] = ch; break; } //没有冲突，填进网格，停止继续尝试
        }
      }
     printf("Case %d:\n", kase);
     for(int i = 0; i < n; i++) printf("%s\n", grid[i]);
   }
   return 0;
}
```

严谨的读者可能又要发问了：如果上述代码顺利执行完毕，即每个格子都有得填，得到的解自然是字典序最小的，但如果某个格子把 A~Z 的所有字母都尝试完，一个都填不了，该怎么办？这意味着必须推翻以前的决策，一下子让情况变得复杂起来。

幸运的是，这种情况不会发生，因为一个格子的上下左右只有 4 个格子，不可能包含 A~Z 这 26 个字母。因此，每个空格都能填上字母。

例题 15　网络（Network, Seoul 2007, LA 3902）

n 台机器连成一个树状网络，其中叶结点是客户端，其他结点是服务器。目前有一台服务器正在提供 VOD（Video On Demand）服务，虽然视频质量本身很不错，但对于那些离它很远的客户端来说，网络延迟却难以忍受。你的任务是在一些其他服务器上也安装同样的服务，使得每台客户端到最近服务器的距离不超过一个给定的整数 *k*。为了节约成本，安装服务的服务器台数应尽量少，如图 1-23 所示，当 *k*=2 时还要在结点 4 处放一台服务器。

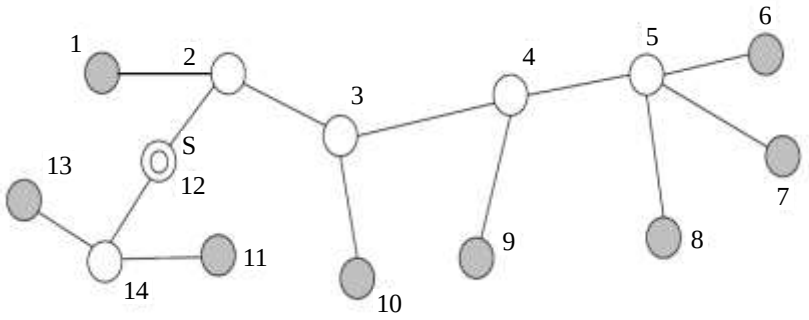


图　1-23

【输入格式】

输入的第一行为数据组数 *T*。每组数据的第一行为树中的结点数 *n*（3≤*n*≤1 000）；下一行包含两个整数 *s* 和 *k*（1≤*s*≤*n*, 1≤*k*≤*n*），其中 *s* 是已经放置好的 VOD 服务器的结点编号，*k* 是叶子和服务器的距离下限；以下 *n*–1 行每行包含两个整数，即树中的一条边。

【输出格式】

对于每组数据，输出一个整数，即还需要放置的 VOD 服务器的个数的最小值。

【分析】

通常来说，把无根树变成有根树会有助于解题。何况在本题中，已经有了一个天然的根结点：原始 VOD 服务器。对于那些已经满足条件（即到原始 VOD 服务器的距离不超过 k）的客户端，直接当它们不存在就可以了，如图 1-24 所示。

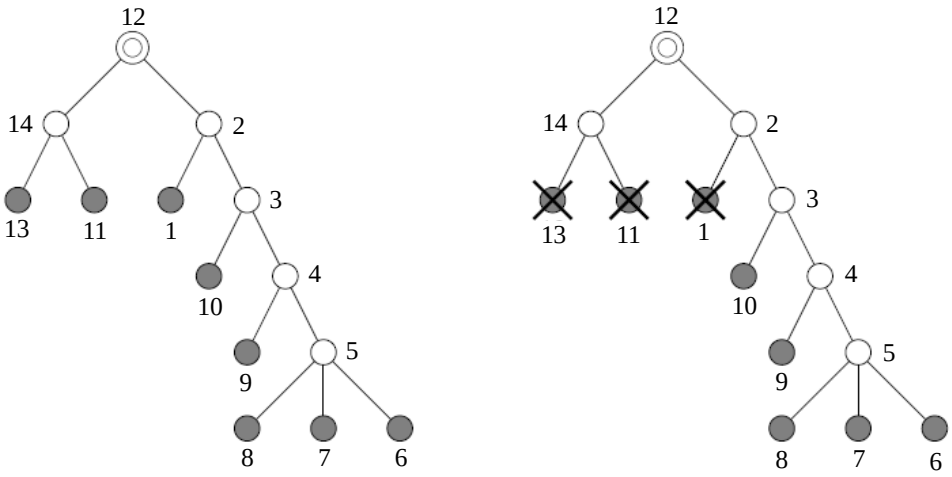


图　1-24

接下来，我们考虑深度最大的结点。比如结点 8，应该在哪里放新的服务器来覆盖（“覆盖”一个叶子是指到该叶子的距离不超过 k）它呢？只有结点 5 和结点 4 满足条件。显然，结点 4 比结点 5 划算，因为结点 5 所覆盖的叶子（6, 7, 8）都能被结点 4 所覆盖。一般的，对于深度最大的结点 u，选择 u 的 k 级祖先是最划算的（父亲是 1 级祖先，父亲的父亲是 2 级祖先，以此类推）。证明过程留给读者自行思考。

下面给出上述算法的一种实现方法：每放一个新服务器，进行一次 DFS，覆盖与它距离不超过 k 的所有结点。注意，本题只需要覆盖叶子，而不需要覆盖中间结点，而且深度不超过 k 的叶子已经被原始服务器覆盖，所以我们只需要处理深度大于 k 的叶结点即可。为了让程序更简单，我们可用 nodes 表避开“按深度排序”的操作，代码如下。

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

const int maxn = 1000 + 10;
vector<int> gr[maxn], nodes[maxn];
int n, s, k, fa[maxn];
bool covered[maxn];
```

```
//无根树转有根树，计算 fa 数组，根据深度把叶子结点插入 nodes 表里
void dfs(int u, int f, int d) {
  fa[u] = f;
  int nc = gr[u].size();
  if(nc == 1 && d > k) nodes[d].push_back(u);
  for(int i = 0; i < nc; i++) {
    int v = gr[u][i];
    if(v != f) dfs(v, u, d+1);
  }
}

void dfs2(int u, int f, int d) {
  covered[u] = true;
  int nc = gr[u].size();
  for(int i = 0; i < nc; i++) {
    int v = gr[u][i];
    if(v != f && d < k) dfs2(v, u, d+1);    //只覆盖到新服务器距离不超过 k 的结点
  }
}

int solve() {
  int ans = 0;
  memset(covered, 0, sizeof(covered));
  for(int d = n-1; d > k; d--)
    for(int i = 0; i < nodes[d].size(); i++) {
      int u = nodes[d][i];
      if(covered[u]) continue;              //不考虑已覆盖的结点

      int v = u;
      for(int j = 0; j < k; j++) v = fa[v]; //v 是 u 的 k 级祖先
      dfs2(v, -1, 0);                       //在结点 v 放服务器
      ans++;
    }
  return ans;
}

int main() {
  int T;
  scanf("%d", &T);
  while(T--) {
    scanf("%d%d%d", &n, &s, &k);
```

```
    for(int i = 1; i <= n; i++) { gr[i].clear(); nodes[i].clear(); }
    for(int i = 0; i < n-1; i++) {
      int a, b;
      scanf("%d%d", &a, &b);
      gr[a].push_back(b);
      gr[b].push_back(a);
    }
    dfs(s, -1, 0);
    printf("%d\n", solve());
  }
  return 0;
}
```

例题 16　长城守卫（Beijing Guards, CERC 2004, LA 3177）

有 n 个人围成一个圈，其中第 i 个人想要 r_i 个不同的礼物。相邻的两个人可以聊天，炫耀自己的礼物。如果两个相邻的人拥有同一种礼物，则双方都会很不高兴。问：一共需要多少种礼物才能满足所有人的需要？假设每种礼物有无穷多个，不相邻的两个人不会一起聊天，所以即使拿到相同的礼物也没关系。

比如，一共有 5 个人，每个人都要一个礼物，则至少要 3 种礼物。如果把这 3 种礼物编号为 1, 2, 3，则 5 个人拿到的礼物应分别是：1,2,1,2,3。如果每个人要两个礼物，则至少要 5 种礼物，且 5 个人拿到的礼物集合应该是：{1,2},{3,4},{1,5},{2,3},{4,5}。

【输入格式】

输入包含多组数据。每组数据的第一行为一个整数 n（$1 \leqslant n \leqslant 100\,000$）；以下 n 行按照圈上的顺序描述每个人的需求，其中每行为一个整数 r_i（$1 \leqslant r_i \leqslant 100\,000$），表示第 i 个人想要 r_i 个不同的礼物。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出所需礼物的种类数。

【分析】

如果 n 为偶数，那么答案为相邻的两个人的 r 值之和的最大值，即 $p=\max\{r_i+r_{i+1}\}$（i=1, 2, …, n），规定 $r_{n+1}=r_1$。不难看出，这个数值是答案的下限，而且还可以构造出只用 p 种礼物的方案：对于一个编号为 i 的人，如果 i 为奇数，发编号为 1~r 的礼物 r_i；如果 i 为偶数，发礼物 $p-r_i+1$~p，请读者自己验证它是否符合要求。

n 为奇数的情况比较棘手，因为上述方法不再奏效。这个时候需要二分答案：假设已知共有 p 种礼物，该如何分配呢？设第 1 个人的礼物是 1~r_1，不难发现最优的分配策略一定是这样的：编号为偶数的人尽量往前取，编号为奇数的人尽量往后取。这样，编号为 n 的人在不冲突的前提下，尽可能地往后取了 r_n 样东西，最后判定编号为 1 的人和编号为 n 的人是否冲突即可。比如，n = 5，A = {2, 2, 5, 2, 5}，p = 8 时，则第 1 个人取{1, 2}，第 2 个人取{3, 4}，第 3 个人取{8, 7, 6, 5, 2}，第 4 个人取{1, 3}，第 5 个人取{8, 7, 6, 5, 4}，由于第 1 个人与第 5 个人不冲突，所以 p = 8 是可行的。

程序实现上，由于题目并不要求输出方案，因此，只需记录每个人在[1~r_1]的范围内取了几个，在[r_1+1~n]的范围里取了几个（在程序中分别用 left[i]和 right[i]表示），最后判断出第 n 个人在[1~r_1]里面是否有取东西即可。代码如下。

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

const int maxn = 100000 + 10;
int n, r[maxn], left[maxn], right[maxn];

//测试 p 个礼物是否足够。
//left[i]是第 i 个人拿到的“左边的礼物”总数，right 类似
bool test(int p) {
  int x = r[1], y = p - r[1];
  left[1] = x; right[1] = 0;
  for(int i = 2; i <= n; i++) {
    if(i % 2 == 0) {
      right[i] = min(y - right[i-1], r[i]);     //尽量拿右边的礼物
      left[i] = r[i] - right[i];
    }
    else {
      left[i] = min(x - left[i-1], r[i]);       //尽量拿左边的礼物
      right[i] = r[i] - left[i];
    }
  }
  return left[n] == 0;
}

int main() {
  int n;
  while(scanf("%d", &n) == 1 && n) {
    for(int i = 1; i <= n; i++) scanf("%d", &r[i]);
    r[n+1] = r[1];

    int L = 0, R = 0;
    for(int i = 1; i <= n; i++) L = max(L, r[i] + r[i+1]);
    if(n % 2 == 1) {
      for(int i = 1; i <= n; i++) R = max(R, r[i]*3);
      while(L < R) {
        int M = L + (R-L)/2;
        if(test(M)) R = M; else L = M+1;
```

```
      }
    }
    printf("%d\n", L);
  }
  return 0;
}
```

1.3　高效算法设计举例

例题 17　年龄排序（Age Sort, UVa 11462）

给定若干居民的年龄（都是 1~100 之间的整数），把它们按照从小到大的顺序输出。

【输入格式】

输入包含多组测试数据。每组数据的第一行为整数 n（$0<n\leqslant 2\,000\,000$），即居民总数；下一行包含 n 个不小于 1、不大于 100 的整数，即各居民的年龄。输入结束标志为 n=0。

输入文件约有 25MB，而内存限制只有 2MB。

【输出格式】

对于每组数据，按照从小到大的顺序输出各居民的年龄，相邻年龄用单个空格隔开。

【分析】

由于数据太大，内存限制太紧（甚至都不能把它们全读进内存），因此无法使用快速排序方法。但整数范围很小，可以用计数排序方法。下面是程序代码。

```
#include<cstdio>
#include<cstring>                    //为了使用 memset 函数
int main() {
  int n, x, c[101];
  while(scanf("%d", &n) == 1 && n) {
    memset(c, 0, sizeof(c));
    for(int i = 0; i < n; i++) {
      scanf("%d", &x);
      c[x]++;
    }
    int first = 1;                   //标志 first=1 表示还没有输出过整数
    for(int i = 1; i <= 100; i++)
      for(int j = 0; j < c[i]; j++) {
        if(!first) printf(" ");  //从第二个数开始，每输出一个数之前先输出一个空格
        first = 0;
        printf("%d", i);
      }
    printf("\n");
  }
```

```
  return 0;
}
```

如果还要精益求精，可以优化输入输出，进一步降低运行时间。程序如下。

```
#include<cstdio>
#include<cstring>
#include<cctype>    //为了使用 isdigit 宏

inline int readint() {
  char c = getchar();
  while(!isdigit(c)) c = getchar();

  int x = 0;
  while(isdigit(c)) {
    x = x * 10 + c - '0';
    c = getchar();
  }
  return x;
}

int buf[10];        //声明成全局变量可以减小开销
inline void writeint(int i) {
  int p = 0;
  if(i == 0) p++;   //特殊情况：i 等于 0 的时候需要输出 0，而不是什么也不输出
  else while(i) {
    buf[p++] = i % 10;
    i /= 10;
  }
  for(int j = p-1; j >=0; j--) putchar('0' + buf[j]); //逆序输出
}

int main() {
  int n, x, c[101];
  while(n = readint()) {
    memset(c, 0, sizeof(c));
    for(int i = 0; i < n; i++) c[readint()]++;
    int first = 1;
    for(int i = 1; i <= 100; i++)
      for(int j = 0; j < c[i]; j++) {
        if(!first) putchar(' ');
        first = 0;
        writeint(i);
```

```
    }
    putchar('\n');
  }
  return 0;
}
```

上述优化使得运行时间缩短了约 2/3。一般情况下，当输入输出数据量很大时，应尽量用 scanf 和 printf 函数；如果时间效率还不够高，应逐字符输入输出，就像上面的 readint 和 writeint 函数[①]。不管怎样，在确信 I/O 时间成为整个程序性能瓶颈之前，不要盲目优化。测试方法也很简单：输入之后不执行主算法，直接输出一个任意的结果，看看运行时间是否过长。

例题 18　开放式学分制（Open Credit System, UVa 11078）

给一个长度为 n 的整数序列 $A_0, A_1, \ldots, A_{n-1}$，找出两个整数 A_i 和 A_j（$i<j$），使得 A_i-A_j 尽量大。

【输入格式】

输入第一行为数据组数 T（$T \leqslant 20$）。每组数据的第一行为整数的个数 n（$2 \leqslant n \leqslant 100\,000$）；以下 n 行，每行为一个绝对值不超过 150 000 的整数。

【输出格式】

对于每组数据，输出 A_i-A_j 的最大值。

【分析】

最简单的一种方法是用二重循环，代码如下。

```
#include<cstdio>
#include<algorithm>
using namespace std;
int A[100000], n;
int main() {
  int T;
  scanf("%d", &T);
  while(T--) {
    scanf("%d", &n);
    for(int i = 0; i < n; i++) scanf("%d", &A[i]);
    int ans = A[0]-A[1]; //初始值。注意不要初始化为 0，因为最终答案可能小于 0
    for(int i = 0; i < n; i++)
      for(int j = i+1; j < n; j++)
        ans = max(ans, A[i]-A[j]);
    //ans >?= A[i]-A[j] 这种写法已经被新版 g++抛弃
    printf("%d\n", ans);
  }
```

[①] 注意：上述 readint 和 writeint 只能处理非负整数，请读者自行编写适用于负整数的函数。

```
  return 0;
}
```

可惜上述算法的时间复杂度是 $O(n^2)$，在 n=100 000 的规模面前无能为力。怎么办呢？对于每个固定的 j，我们应该选择的是小于 j 且 A_i 最大的 i，而和 A_j 的具体数值无关。这样，我们从小到大枚举 j，顺便维护 A_i 的最大值即可。代码如下。

```
#include<cstdio>
#include<algorithm>
using namespace std;
int A[100000], n;
int main() {
  int T;
  scanf("%d", &T);
  while(T--) {
    scanf("%d", &n);
    for(int i = 0; i < n; i++) scanf("%d", &A[i]);
    int ans = A[0]-A[1];
    int MaxAi = A[0];            //MaxAi 动态维护 A[0]，A[1]，…，A[j-1]的最大值
    for(int j = 1; j < n; j++) {  //j 从 1 而不是 0 开始枚举，因为 j=0 时，不存在 i
      ans = max(ans, MaxAi-A[j]);
      MaxAi = max(A[j], MaxAi);  //MaxAi 晚于 ans 更新。想一想，为什么
    }
    printf("%d\n", ans);
  }
  return 0;
}
```

不难发现，上述程序的时间复杂度为 $O(n)$。和刚才的平方算法相比，这个算法快就快在每次用 $O(1)$时间更新了 MaxAi，而不是重新计算。

如果你已经理解了这个算法，不妨思考一下，如果题目要求输出对应的 i 和 j，应该怎么办？另外，你能不用 A 数组实现边读边计算么？这样可以让附加空间从 $O(n)$降低到 $O(1)$。

例题 19　计算器谜题（Calculator Conundrum, UVa 11549）

有一个老式计算器，只能显示 n 位数字。有一天，你无聊了，于是输入一个整数 k，然后反复平方，直到溢出。每次溢出时，计算器会显示出结果的最高 n 位和一个错误标记。然后清除错误标记，继续平方。如果一直这样做下去，能得到的最大数是多少？比如，当 n=1, k=6 时，计算器将依次显示 6、3（36 的最高位），9、8（81 的最高位），6（64 的最高位），3……

【输入格式】

输入的第一行为一个整数 T（1≤T≤200），即测试数据的数量。以下 T 行，每行包含两个整数 n 和 k（1≤n≤9，0≤k<10^n）。

【输出格式】

对于每组数据，输出你能得到的最大数。

【分析】

题目已经暗示了计算器显示出的数将出现循环（想一想，为什么），所以不妨一个一个地模拟，每次判断新得到的数是否以前出现过。如何判断呢？一种方法是把所有计算出来的数放到一个数组里，然后一一进行比较。不难发现，这样每次判断需要花费非常多的时间，相当慢。能否开一个数组 *vis*，直接读 *vis*[*k*]判断整数 *k* 是否出现过呢？很遗憾，*k* 的范围太大，开不下。在这种情况下，一个简便的方法是利用 STL 的集合，代码如下。

```
#include<set>
#include<iostream>
#include<sstream>
using namespace std;

int next(int n, int k) {
  stringstream ss;
  ss << (long long)k * k; //注意，k*k 可能会溢出。必须先转化为 long long 再相乘
  string s = ss.str();
  if(s.length() > n) s = s.substr(0, n);    //结果太长，只取前 n 位
  int ans;
  stringstream ss2(s);
  ss2 >> ans;
  return ans;
}

int main() {
  int T;
  cin >> T;
  while(T--) {
    int n, k;
    cin >> n >> k;
    set<int> s;
    int ans = k;
    while(!s.count(k)) {                    //以前没有出现过
      s.insert(k);
      if(k > ans) ans = k;
      k = next(n, k);
    }
    cout << ans << endl;
  }
  return 0;
}
```

上述程序在 UVa OJ 上的运行时间为 4.5 秒。有经验的读者应该知道，STL 的 string 很

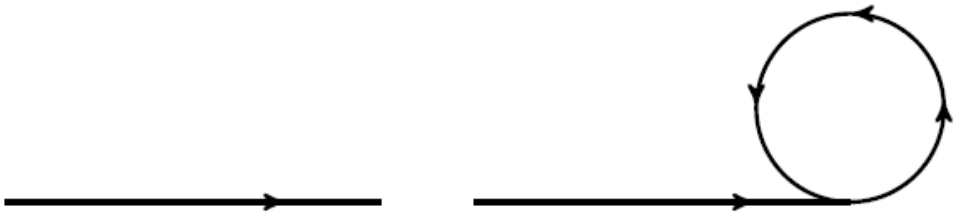
慢，stringstream 更慢，所以需要考虑把它们换掉。

```
int buf[10];
int next(int n, int k) {
  if(!k) return 0;
  long long k2 = (long long)k * k;
  int L = 0;
  while(k2 > 0) { buf[L++] = k2 % 10; k2 /= 10; }  //分离并保存 k² 的各个数字
  if(n > L) n = L;
  int ans = 0;
  for(int i = 0; i < n; i++)                        //把前 min{n,L}位重新组合
    ans = ans * 10 + buf[--L];
  return ans;
}
```

上述程序的运行时间降为 1 秒。

当然，也可以用哈希表（详见《入门经典》的相关部分），但和 set 一样，空间开销比较大。有没有空间开销比较小且速度也不错的方法呢？答案是肯定的。

想象一下，假设有两个小孩子在一个“可以无限向前跑”的跑道上赛跑，同时出发，但其中一个小孩的速度是另一个的两倍。如果跑道是直的（如图 1-25（a）所示），跑得快的小孩永远在前面；但如果跑道有环（如图 1-25（b）所示），则跑得快的小孩将“追上”跑得慢的小孩。



（a）　　（b）

图　1-25

这个算法称为 Floyd 判圈算法，不仅空间复杂度将降为 *O*(1)，运行时间也将缩短到 0.5 秒。主程序如下。

```
int main() {
  int T;
  cin >> T;
  while(T--) {
    int n, k;
    cin >> n >> k;
    int ans = k;
    int k1 = k, k2 = k;
    do {
```

```
        k1 = next(n, k1); //小孩 1
        k2 = next(n, k2); if(k2 > ans) ans = k2; //小孩 2，第一步
        k2 = next(n, k2); if(k2 > ans) ans = k2; //小孩 2，第二步
      } while(k1 != k2); //追上以后才停止
      cout << ans << endl;
    }
    return 0;
  }
```

例题 20　流星（Meteor, Seoul 2007, LA 3905）

给你一个矩形照相机，还有 n 个流星的初始位置和速度，求能照到流星最多的时刻。注意，在相机边界上的点不会被照到。如图 1-26 所示，流星 2、3、4、5 将不会被照到，因为它们从来没有经过图中矩形的内部。

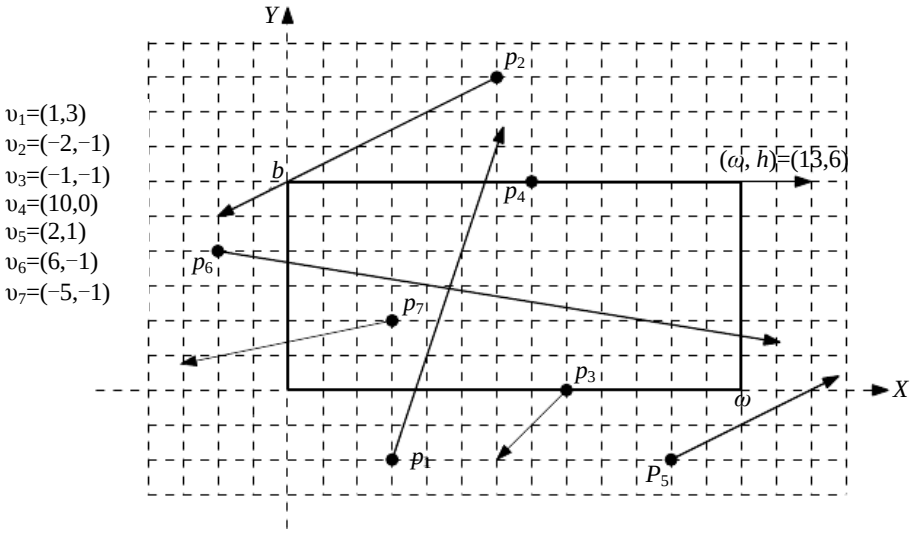


图　1-26

相机的左下角为(0,0)，右上角为(w,h)。每个流星用两个向量 p 和 v 表示，其中，p 为初始（t=0 时）位置，v 为速度。在时刻 t（t>=0）的位置是 p+tv。比如，若 p=(1,3)，v=(−2,5)，则 t=0.5 时该流星的位置为(1,3) + 0.5×(−2,5) = (0, 5.5)。

【输入格式】

输入的第一行为测试数据组数 T。每组数据的第一行为两个整数 w 和 h（1≤w，h≤100 000）；第二行为流星个数 n（1≤n≤100 000）；以下 n 行每行用 4 个整数 x_i, y_i, a_i, b_i（−200 000≤x_i,y_i≤200 000，−10≤a_i,b_i≤10）描述一个流星，其中(x_i,y_i)是初始位置，(a_i,b_i)是速度。a_i 和 b_i 不同时为 0。不同流星的初始位置不同。

【输出格式】

对于每组数据，输出能照到的流星个数的最大值。

【分析】

不难发现，流星的轨迹是没有直接意义的，有意义的只是每个流星在照相机视野内出

现的时间段。换句话说，我们把本题抽象为这样一个问题：给出 n 个开区间(L_i, R_i)，你的任务是求出一个数 t，使得包含它的区间数最多（为什么是开区间呢？请读者思考）。开区间(L_i, R_i)是指所有满足 $L_i < x < R_i$ 的实数 x 的集合。

把所有区间画到平行于数轴的直线上（免得相互遮挡，看不清楚），然后想象有一条竖直线从左到右进行扫描，则问题可以转化为：求扫描线在哪个位置时与最多的开区间相交，如图 1-27 所示。

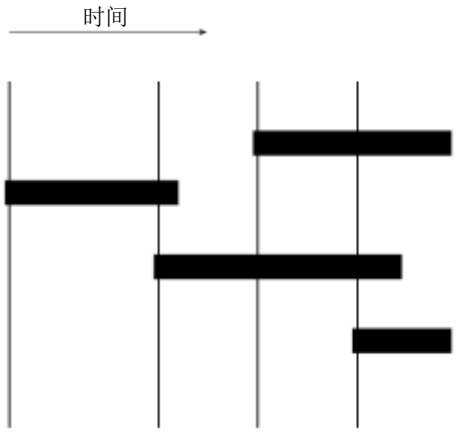


图 1-27

不难发现，当扫描线移动到某个区间左端点的“右边一点点”时最有希望和最多的开区间相交（想一想，为什么）。为了快速得知在这些位置时扫描线与多少条线段相交，我们再一次使用前面提到的技巧：维护信息，而不是重新计算。

我们把“扫描线碰到一个左端点”和“扫描线碰到一个右端点”看成是事件（event），则扫描线移动的过程就是从左到右处理各个事件的过程。每遇到一个“左端点事件”，计数器加 1；每遇到一个“右端点事件”，计数器减 1。这里的计数器保存的正是我们要维护的信息：扫描线和多少个开区间相交，如图 1-28 所示。

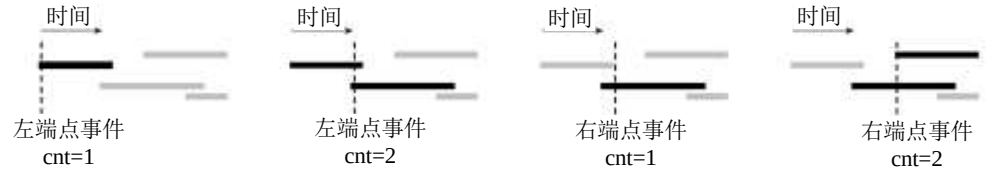


图 1-28

这样，我们可以写出这样一段伪代码。

```
将所有事件按照从左到右排序
while(还有未处理的事件) {
  选择最左边的事件 E
  if(E 是“左端点事件”) { cnt++; if(cnt > ans) ans = cnt; } //更新计数器和答案
  else cnt--; //一定是“右端点事件”
}
```

这段伪代码看上去挺有道理，但实际上暗藏危险：如果不同事件的端点相同，那么哪个排在前面呢？考虑这样一种情况——输入是两个没有公共元素的开区间，且左边那个区间的右端点和右边那个区间的左端点重合。在这种情况下，两种排法的结果截然不同：如果先处理左端点事件，执行结果是 2；如果先处理右端点事件，执行结果是 1。这才是正确答案。

这样，我们得到了一个完整的扫描算法：先按照从左到右的顺序给事件排序，对于位置相同的事件，把右端点事件排在前面，然后执行上述伪代码的循环部分。如果你对这个冲突解决方法心存疑虑，不妨把它理解成把所有区间的右端点往左移动了一个极小（但大于 0）的距离。代码如下。

```
#include<cstdio>
#include<algorithm>
using namespace std;
//0<x+at<w
void update(int x, int a, int w, double& L, double& R) {
  if(a == 0) {
    if(x <= 0 || x >= w) R = L-1; //无解
  } else if(a > 0) {
    L = max(L, -(double)x/a);
    R = min(R, (double)(w-x)/a);
  } else {
    L = max(L, (double)(w-x)/a);
    R = min(R, -(double)x/a);
  }
}

const int maxn = 100000 + 10;

struct Event {
  double x;
  int type;
  bool operator < (const Event& a) const {
    return x < a.x || (x == a.x && type > a.type); //先处理右端点
  }
} events[maxn*2];

int main() {
  int T;
  scanf("%d", &T);
  while(T--) {
    int w, h, n, e = 0;
    scanf("%d%d%d", &w, &h, &n);
```

```
    for(int i = 0; i < n; i++) {
      int x, y, a, b;
      scanf("%d%d%d%d", &x, &y, &a, &b);
      //0<x+at<w, 0<y+bt<h, t>=0
      double L = 0, R = 1e9;
      update(x, a, w, L, R);
      update(y, b, h, L, R);
      if(R > L) {
        events[e++] = (Event){L, 0};
        events[e++] = (Event){R, 1};
      }
    }
    sort(events, events+e);
    int cnt = 0, ans = 0;
    for(int i = 0; i < e; i++) {
      if(events[i].type == 0) ans = max(ans, ++cnt);
      else cnt--;
    }
    printf("%d\n", ans);
  }
  return 0;
}
```

另外，本题还可以完全避免实数运算，全部采用整数：只需要把代码中的 double 全部改成 int，然后在 update 函数中把所有返回值乘以 lcm(1,2,…,10)=2 520 即可（想一想，为什么）。

```
void update(int x, int a, int w, int& L, int& R) {
  if(a == 0) {
    if(x <= 0 || x >= w) R = L-1; //无解
  } else if(a > 0) {
    L = max(L, -x*2520/a);
    R = min(R, (w-x)*2520/a);
  } else {
    L = max(L, (w-x)*2520/a);
    R = min(R, -x*2520/a);
  }
}
```

例题 21　子序列（Subsequence, SEERC 2006, LA 2678）

有 n 个正整数组成一个序列。给定整数 S，求长度最短的连续序列，使它们的和大于或等于 S。

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n 和 S（$10<n\leqslant 100\,000$，$S<10^9$）；第

二行为 n 个正整数，均不超过 10 000。输入结束标志为文件结束符（EOF）。

【输出格式】

对于每组数据，输出满足条件的最短序列的长度。如果不存在，输出 0。

【分析】

和《开放式学分制》一样，本题最直接的思路是二重循环，枚举子序列的起点和终点。代码如下（输入数据已存入数组 $A[1]$~$A[n]$）。

```
int ans = n+1;
for(int i = 1; i <= n; i++)
  for(int j = i; j <= n; j++) {
    int sum = 0;
    for(int k = i; k <= j; k++) sum += A[k];
    if(sum >= S) ans = min(ans, j-i+1);
  }
printf("%d\n", ans == n+1 ? 0 : ans);
```

很可惜，上述程序的时间复杂度是 $O(n^3)$的，因此，当 n 达到 100 000 的规模后，程序将无能为力。有一个方法可以降低时间复杂度，即常见的前缀和技巧。令 $B_i=A_1+A_2+\cdots+A_i$，规定 $B_0=0$，则可以在 $O(1)$时间内求出子序列的值：$A_i+A_{i+1}+\ldots+A_j=B_j-B_{i-1}$。这样，时间复杂度降为 $O(n^2)$，代码如下。

```
B[0] = 0;
for(int i = 1; i <= n; i++) B[i] = B[i-1] + A[i];
int ans = n+1;
  for(int i = 1; i <= n; i++)
    for(int j = i; j <= n; j++)
      if(B[j] - B[i-1] >= S) ans = min(ans, j-i+1);
printf("%d\n", ans == n+1 ? 0 : ans);
```

遗憾的是，本题的数据规模太大，$O(n^2)$时间复杂度的算法也太慢。不难发现，只要同时枚举起点和终点，时间复杂度不可能比 $O(n^2)$更低，所以必须另谋他路。比如，是否可以不枚举终点，只枚举起点，或者不枚举起点，只枚举终点呢？

我们首先试试只枚举终点。对于终点 j，我们的目标是要找到一个让 $B_j-B_{i-1}\geqslant S$，且 i 尽量大（i 越大，序列长度 $j-i+1$ 就越小）的 i 值，也就是找一个让 $B_{i-1}\leqslant B_j-S$ 最大的 i。考虑图 1-29 所示的序列。

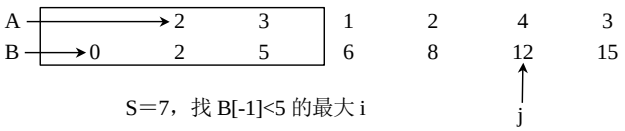


图　1-29

当 j=5 时，B_5=12，因此目标是找一个 $B_{i-1}\leqslant 12-7=5$ 的最大 i。注意到 B 是递增的（别忘了，本题中所有 A_i 均为整数），所以可以用二分查找。如果使用 STL 的话，这里的 i 就是

lower_bound(*B*, *B*+*j*, *B*[*j*]−*S*)。代码如下。

```
B[0] = 0;
for(int i = 1; i <= n; i++) B[i] = B[i-1] + A[i];
int ans = n+1;
for(int j = 1; j <= n; j++) {
  int i = lower_bound(B, B+j, B[j]-S) - B;
  if(i > 0) ans = min(ans, j-i+1);
}
printf("%d\n", ans == n+1 ? 0 : ans);
```

上面代码的时间复杂度是 $O(n\log n)$。可以将其继续优化到 $O(n)$。由于 j 是递增的，B_j 也是递增的，所以 $B_{i-1} \leqslant B_j - S$ 的右边也是递增的。换句话说，满足条件的 i 的位置也是递增的。因此我们可以写出这样的程序。

```
B[0] = 0;
for(int i = 1; i <= n; i++) B[i] = B[i-1] + A[i];
int ans = n+1;
int i = 1;
for(int j = 1; j <= n; j++) {
  if(B[i-1] > B[j]-S) continue;     //(1)没有满足条件的 i，换下一个 j
  while(B[i] <= B[j]-S) i++;        //(2)求满足 B[i-1]<=B[j]-S 的最大 i
  ans = min(ans, j-i+1);
}
printf("%d\n", ans == n+1 ? 0 : ans);
```

这段程序的时间复杂度如何？似乎答案并不那么明显，因为它是一个二重循环：外层循环 j，内层循环 i。这时我们需要一点技巧，用不同方式统计不同语句的执行次数。语句（1）和（2）的执行次数为 n，因为每个不同的 j 都执行了一次；语句（2）的执行次数有些复杂，因为不同的 j 对应的执行次数不一样。但我们可以从另外一个角度考虑：i 从未减小，一直递增，所以递增次数一定不超过 n。换句话说，整个程序的时间复杂度为 $O(n)$。

例题 22　最大子矩阵（City Game, SEERC 2004, LA 3029）

给定一个 $m \times n$ 的矩阵，其中一些格子是空地（F），其他是障碍（R）。找出一个全部由 F 组成的面积最大的子矩阵，输出其面积乘以 3 后的结果。

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为整数 m 和 n（$1 \leqslant m,n \leqslant 1\,000$）；以下 m 行每行 n 个字符（保证为 F 或者 R），即输入矩阵。

【输出格式】

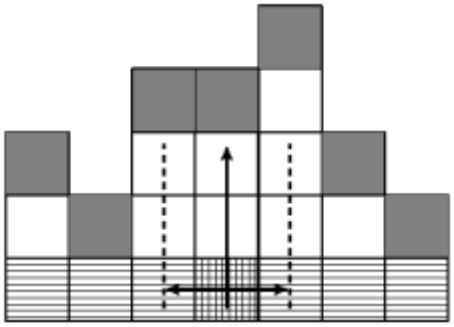
对于每组数据，输出面积最大的、全由 F 组成的矩阵的面积乘以 3 后的结果。

【分析】

最容易想到的算法便是：枚举左上角坐标和长、宽，然后判断这个矩形是否全为空地。这样做需要枚举 $O(m^2n^2)$个矩形，判断需要 $O(mn)$时间，总时间复杂度为 $O(m^3n^3)$，实在是太

高了。本题虽然是矩形，但仍然可以用扫描法：从上到下扫描。

我们把每个格子向上延伸的连续空格看成一条悬线，并且用 up(*i*,*j*)、left(*i*,*j*)、right(*i*,*j*)表示格子(*i*,*j*)的悬线长度以及该悬线向左、向右运动的“运动极限”，如图 1-30 所示。列 3 的悬线长度为 3，向左向右各能运动一列，因此左右的运动极限分别为列 2 和列 4。



j:	0	1	2	3	4	5	6
up:	2	1	3	3	4	2	1
left:	0	0	2	2	4	2	0
right:	0	6	4	4	4	5	6

图　1-30

这样，每个格子(*i*,*j*)对应着一个以第 *i* 行为下边界、高度为 up(*i*,*j*)，左右边界分别为 left(*i*,*j*)和 right(*i*,*j*)的矩形。不难发现，所有这些矩形中面积最大的就是题目所求（想一想，为什么）。这样，我们只需思考如何快速计算出上述 3 种信息即可。

当第 *i* 行第 *j* 列不是空格时，3 个数组的值均为 0，否则 up(*i*,*j*)=up(*i*−1,*j*)+1。那么，left 和 right 呢？深入思考后，可以发现：

$$\text{left}(i,j) = \max\{\text{left}(i-1,j), \text{lo}+1\}$$

其中 lo 是第 *i* 行中，第 *j* 列左边的最近障碍格的列编号。如果从左到右计算 left(*i*,*j*)，则很容易维护 lo。right 也可以同理计算，但需要从右往左计算，因为要维护第 *j* 列右边最近的障碍格的列编号 ro。为了节约空间，下面的程序用 up[*j*]，left[*j*]和 right[*j*]来保存当前扫描行上的信息。

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

const int maxn = 1000;
int mat[maxn][maxn], up[maxn][maxn], left[maxn][maxn], right[maxn][maxn];
int main() {
  int T;
  scanf("%d", &T);
  while(T--) {
int m, n;
//读入数据
```

```
    scanf("%d%d", &m, &n);
    for(int i = 0; i < m; i++)
      for(int j = 0; j < n; j++) {
        int ch = getchar();
        while(ch != 'F' && ch != 'R') ch = getchar();
        mat[i][j] = ch == 'F' ? 0 : 1;
      }

    int ans = 0;
    for(int i = 0; i < m; i++) {    //从上到下逐行处理
      int lo = -1, ro = n;
      for(int j = 0; j < n; j++)    //从左到右扫描，维护 up 和 left
        if(mat[i][j] == 1) { up[i][j] = left[i][j] = 0; lo = j; }
        else {
          up[i][j] = i == 0 ? 1 : up[i-1][j] + 1;
          left[i][j] = i == 0 ? lo+1 : max(left[i-1][j], lo+1);
        }
      for(int j = n-1; j >= 0; j--) //从右到左扫描，维护 right 并更新答案
        if(mat[i][j] == 1) { right[i][j] = n; ro = j; }
        else {
          right[i][j] = i == 0 ? ro-1 : min(right[i-1][j], ro-1);
          ans = max(ans, up[i][j]*(right[i][j]-left[i][j]+1));
        }
    }
    printf("%d\n", ans*3);          //题目要求输出最大面积乘以 3 后的结果
  }
  return 0;
}
```

程序的时空复杂度均为 $O(mn)$。另外，本题可以用一个栈来代替 left 和 right 数组，有兴趣的读者可以自行研究。但不管采用怎样的程序实现，上述的递推、扫描思想都是解决问题的关键。

例题 23　遥远的银河（Distant Galaxy, Shanghai 2006, LA 3695）

给出平面上的 n 个点，找一个矩形，使得边界上包含尽量多的点。

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 100$）；以下 n 行每行两个整数，即各个点的坐标（坐标均为绝对值不超过 10^9 的整数）。输入结束标志为 $n=0$。

【输出格式】

对于每组数据，输出边界点个数的最大值。

【分析】

不难发现，除非所有输入点都在同一行或者同一列上（此时答案为 n），最优矩形的 4

条边都至少有一个点（一个角上的点同时算在两条边上）。这样，我们可以枚举 4 条边界所穿过的点，然后统计点数。这样做的时间复杂度为 $O(n^5)$（统计点数还需要 $O(n)$时间），无法承受。

和《子序列》一题类似，可以考虑部分枚举，即只枚举矩形的上下边界，用其他方法确定左右边界，过程如图 1-31 所示。

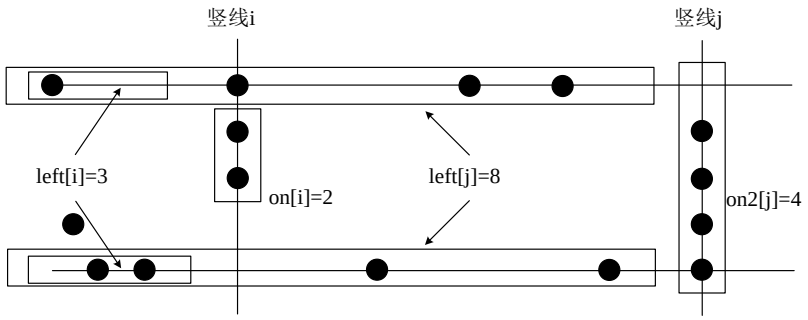


图　1-31

对于竖线 i，我们用 left[i]表示竖线左边位于上下边界上的点数（注意，不统计位于该竖线上的点），on[i]和 on2[i]表示竖线上位于上下边界之间的点数（区别在于 on[i]不统计位于上下边界上的点数，而 on2[i]要统计）。这样，给定左右边界 i 和 j 时，矩形边界上的点数为 left[j]−left[i]+on[i]+on2[j]。当右边界 j 确定时，on[i]-left[i]应最大。

枚举完上下边界后，我们先花 $O(n)$时间按照从左到右的顺序扫描一遍所有点，计算 left、on[i]和 on2[i]数组，然后枚举右边界 j，同时维护 on[i]−left[i]（$i<j$）的最大值。这一步本质上等价于例题《开放式学分制》。代码如下。

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

struct Point {
  int x, y;
  bool operator < (const Point& rhs) const {
    return x < rhs.x;
  }
};

const int maxn = 100 + 10;
Point P[maxn];
int n, m, y[maxn], on[maxn], on2[maxn], left[maxn];

int solve() {
  sort(P, P+n);
  sort(y, y+n);
```

```
  m = unique(y, y+n) - y;            //所有不同的 y 坐标的个数
  if(m <= 2) return n;               //最多两种不同的 y

  int ans = 0;
  for(int a = 0; a < m; a++)
    for(int b = a+1; b < m; b++) {
      int ymin = y[a], ymax = y[b]; //计算上下边界分别为 ymin 和 ymax 时的解

      //计算 left, on, on2
      int k = 0;
      for(int i = 0; i < n; i++) {
        if(i == 0 || P[i].x != P[i-1].x) { //一条新的竖线
          k++;
          on[k] = on2[k] = 0;
          left[k] = k == 0 ? 0 : left[k-1] + on2[k-1] - on[k-1];
        }
        if(P[i].y > ymin && P[i].y < ymax) on[k]++;
        if(P[i].y >= ymin && P[i].y <= ymax) on2[k]++;
      }
      if(k <= 2) return n;           //最多两种不同的 x

      int M = 0;
      for(int j = 1; j <= k; j++) {
        ans = max(ans, left[j]+on2[j]+M);
        M = max(M, on[j]-left[j]);
      }
    }
  return ans;
}

int main() {
  int kase = 0;
  while(scanf("%d", &n) == 1 && n) {
    for(int i = 0; i < n; i++) { scanf("%d%d", &P[i].x, &P[i].y); y[i] = P[i].y; }
    printf("Case %d: %d\n", ++kase, solve());
  }
  return 0;
}
```

例题 24　废料堆（Garbage Heap, UVa 10755）

有个长方体形状的废料堆，由 $A \times B \times C$ 个废料块组成，每个废料块都有一个价值，可正可负。现在要在这个长方体上选择一个子长方体，使组成这个子长方体的废料块的价值之

和最大。

【输入格式】

输入的第一行为数据组数 T（T≤15）。每组数据的第一行为 3 个整数 A, B, C（1≤A,B, C≤20）。接下来有 $A\times B\times C$ 个整数，即各个废料块的价值，每个废料块的价值的绝对值不超过 2^{31}。如果给每个废料块赋予一个空间坐标（一个角为(1,1,1)，对角线的另一端为(A,B,C)），则这些废料块在输入文件中的出现顺序为：(1,1,1), (1,1,2), …, (1,1,C), (1,2,1), …, (1,2,C), …, (1,B,C), …, (2,1,1), …, (2,B,C), …, (A,B,C)。

【输出格式】

对于每组数据，输出最大子长方体的价值和。

【分析】

还是老规矩，先想一个正确但低效的方法。枚举 x，y，z 的上下界 $x1, x2, y1, y2, z1, z2$，然后比较这 $O(n^6)$个长方体的价值和，而每个长方体还需要 $O(n^3)$时间累加出价值和，所以总时间复杂度为 $O(n^9)$，即使对于 n≤20 这样的规模，也太大了。

解决高维问题的常见思路是降维。让我们先来看看本题的二维情况：给定一个数字矩阵，求一个和最大的连续子矩阵。借用上题的思路，我们枚举上下边界 $y1$ 和 $y2$（规定 x 从左到右递增，y 从上到下递增），则问题转化为了一维问题，如图 1-32 所示。

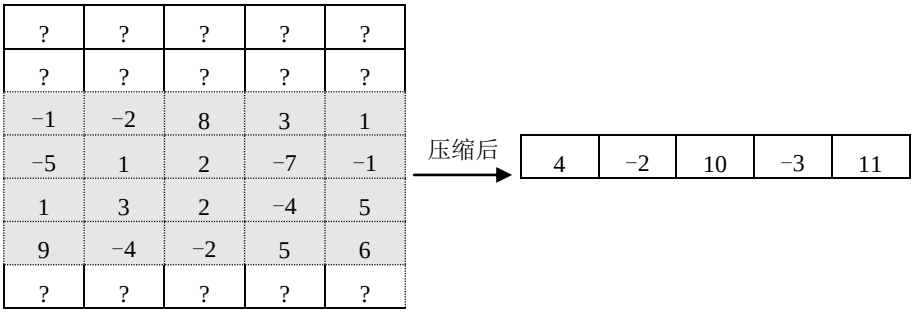


图　1-32

注意，右图这个一维问题中的一个元素对应左图 4 个灰色格子的数之和。比如，(−1)+(−5)+1+9=4，(−2)+1+3+(−4)=−2 等。

为了节省时间，这 4 个元素不能再用一重循环来累加得到，否则时间复杂度会变成 $O(n^4)$。我们得想办法让这些元素可以在 $O(1)$时间内得到，这样，二维问题才能在 $O(n^3)$时间内解决。

解决方法仍然是前面曾多次使用的递推法：设 sum(x,$y1$,$y2$)表示满足 $y1\leq y\leq y2$ 的所有格子(x,y)里的数之和，则当 $y1<y2$ 时，sum(x,$y1$,$y2$)=sum(x,$y1$,$y2-1$)+A[x][$y2$]。这样，可以事先在 $O(n^3)$时间内算出整个 sum 数组，则所有一维问题中的元素都可以在 $O(1)$时间内得到，完整的二维问题在 $O(n^3)$时间内得到了解决。尽管这个方法在时间效率上不错，但却占据了较大空间。

有没有一种办法可在保持 $O(n^3)$时间复杂度的同时，降低空间开销呢？办法之一就是使用二维前缀和。设 $S(x,y)$为满足 $x'\leq x$, $y'\leq y$ 的所有 A[x'][y']之和，即以(x,y)为右下角的矩形

中的所有元素之和，这样所有子矩形的元素之和等于 4 个“前缀矩形”的元素之和经过加减之后得到。如图 1-33 所示，黑色部分的元素之和等于以 1 号、4 号为右下角的前缀矩形的元素和减去以 2 号、3 号为右下角的前缀矩形的元素和。

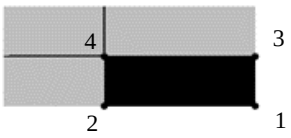


图 1-33

这个关系也可以用来递推出整个 S 数组（注意，x,y 都从 1 开始编号）。

$$S(0,y)=S(x,0)=0$$

$$S(x,y)=S(x-1,y)+S(x,y-1)-S(x-1,y-1)+A[x][y]$$

第二个方法是边枚举边递推。此时，需要用到一个辅助数组 C。先按照升序枚举 $y1$，对于每个 $y1$，先清空 C，再按照升序枚举 $y2$；每枚举一个新的 $y2$，先把所有 $C[x]$都累加 $A[x][y2]$，然后计算数组 C 的最大连续和。对于给定的$(y1,y2)$，这个 $C[x]$实际上就是 $\text{sum}(x,y1,y2)$，但是因为及时用新数据覆盖了旧数据（那些数据再也用不到了），所以辅助空间占用仅为 $O(n)$。

上述两种方法都可以很方便地推广到三维情形，时间复杂度为 $O(n^5)$。因为三维情况下的 n 很小，因此前面所说的空间问题并不严重。下面是算法一的完整代码，它用三维数组 S 保存以(x,y,z)为“右下角”的长方体的元素和。代码效率不算高，但读者很容易把它推广到四维或更高维的情形。

```
#include<cstdio>
#include<cstring>
#include<algorithm>
#define FOR(i,s,t)  for(int i = (s); i <= (t); ++i)
using namespace std;

void expand(int i, int& b0, int& b1, int& b2) {
  b0 = i&1; i >>= 1;
  b1 = i&1; i >>= 1;
  b2 = i&1;
}

int sign(int b0, int b1, int b2) {
  return (b0 + b1 + b2) % 2 == 1 ? 1 : -1;
}

const int maxn = 30;
const long long INF = 1LL << 60;
```

```
long long S[maxn][maxn][maxn];

long long sum(int x1, int x2, int y1, int y2, int z1, int z2) {
  int dx = x2-x1+1, dy = y2-y1+1, dz = z2-z1+1;
  long long s = 0;
  for(int i = 0; i < 8; i++) {
    int b0, b1, b2;
    expand(i, b0, b1, b2);
    s -= S[x2-b0*dx][y2-b1*dy][z2-b2*dz] * sign(b0, b1, b2);
  }
  return s;
}

int main() {
  int T;
  scanf("%d", &T);
  while(T--) {
    int a, b, c, b0, b1, b2;
    scanf("%d%d%d", &a, &b, &c);
    memset(S, 0, sizeof(S));
    FOR(x,1,a) FOR(y,1,b) FOR(z,1,c) scanf("%lld", &S[x][y][z]);
    FOR(x,1,a) FOR(y,1,b) FOR(z,1,c) FOR(i,1,7){
      expand(i, b0, b1, b2);
      S[x][y][z] += S[x-b0][y-b1][z-b2] * sign(b0, b1, b2);
    }
    long long ans = -INF;
    FOR(x1,1,a) FOR(x2,x1,a) FOR(y1,1,b) FOR(y2,y1,b) {
      long long M = 0;
      FOR(z,1,c) {
        long long s = sum(x1,x2,y1,y2,1,z);
        ans = max(ans, s - M);
        M = min(M, s);
      }
    }
    printf("%lld\n", ans);
    if(T) printf("\n");
  }
  return 0;
}
```

例题 25　侏罗纪（Jurassic Remains, NEERC 2003, LA 2965）

给定 n 个大写字母组成的字符串。选择尽量多的串，使得每个大写字母都能出现偶数次。

【输入格式】

输入包含多组数据。每组数据的第一行为正整数 n（$1 \leqslant n \leqslant 24$），以下 n 行每行包含一个大写字母组成的字符串。

【输出格式】

对于每组数据，第一行输出整数 k，即字符串个数的最大值。第二行按照从小到大的顺序输出选中的 k 个字符串的编号（字符串按照输入顺序编号为 1~n）。

【样例输入】

```
6
ABD
EG
GE
ABE
AC
BCD
```

【样例输出】

```
5
1 2 3 5 6
```

【分析】

在一个字符串中，每个字符出现的次数本身是无关紧要的，重要的只是这些次数的奇偶性，因此想到用一个二进制的位表示一个字母（1 表示出现奇数次，0 表示出现偶数次）。比如样例的 6 个数，写成二进制后如图 1-34 所示。

A	B	C	D	E	F	G	H	…			
1	1	0	1	0	0	0	0	…	A	B	D
0	0	0	0	1	0	1	0	…	E	G	
0	0	0	0	1	0	1	0	…	G	E	
1	1	0	0	1	0	0	0	…	A	B	E
1	0	1	0	0	0	0	0	…	A	C	
0	1	1	1	0	0	0	0	…	B	C	D

图 1-34

此时，问题转化为求尽量多的数，使得它们的 xor（异或）值为 0。

最容易想到的方法是直接穷举，时间复杂度为 $O(2^n)$，有些偏大。注意到 xor 值为 0 的两个整数必须完全相等，我们可以把字符串分成两个部分：首先计算前 $n/2$ 个字符串所能得到的所有 xor 值，并将其保存到一个映射 S（xor 值→前 $n/2$ 个字符串的一个子集）中；然后枚举后 $n/2$ 个字符串所能得到的所有 xor 值，并每次都在 S 中查找。

如果映射用 STL 的 map 实现，总时间复杂度为 $O(2^{n/2}\log n)$，即 $O(1.44^n \log n)$，比第一种方法好了很多。这样的策略称为中途相遇法（Meet-in-the-Middle）。密码学中著名的中途相遇攻击（Meet-in-the-Middle attack）就是基于这个原理。

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

const int maxn = 24;
map<int,int> table;

int bitcount(int x) { return x == 0 ? 0 : bitcount(x/2) + (x&1); }

int main() {
  int n, A[maxn];
  char s[1000];

  while(scanf("%d", &n) == 1 && n) {
    //输入并计算每个字符串对应的位向量
    for(int i = 0; i < n; i++) {
      scanf("%s", s);
      A[i] = 0;
      for(int j = 0; s[j] != '\0'; j++) A[i] ^= (1<<(s[j]-'A'));
    }
    //计算前 n1 个元素的所有子集的 xor 值
    //table[x]保存的是 xor 值为 x 的，bitcount 尽量大的子集
    table.clear();
    int n1 = n/2, n2 = n-n1;
    for(int i = 0; i < (1<<n1); i++) {
      int x = 0;
      for(int j = 0; j < n1; j++) if(i & (1<<j)) x ^= A[j];
      if(!table.count(x) || bitcount(table[x]) < bitcount(i)) table[x] = i;
    }
    //枚举后 n2 个元素的所有子集，并在 table 中查找
    int ans = 0;
    for(int i = 0; i < (1<<n2); i++) {
      int x = 0;
      for(int j = 0; j < n2; j++) if(i & (1<<j)) x ^= A[n1+j];
      if(table.count(x)&&bitcount(ans)<bitcount(table[x])+bitcount(i)) ans
= (i<<n1)^table[x];
    }
    //输出结果
    printf("%d\n", bitcount(ans));
    for(int i = 0; i < n; i++) if(ans & (1<<i)) printf("%d ", i+1);
    printf("\n");
  }
```

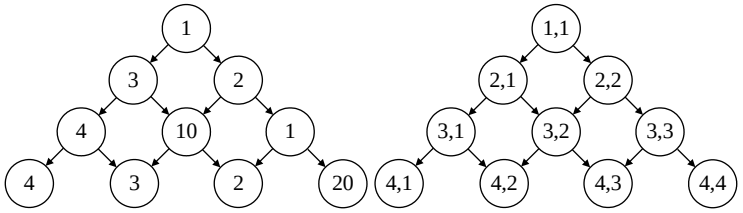
```
  return 0;
}
```

1.4 动态规划专题

在《算法竞赛入门经典》中，我们已经接触过了不少动态规划题目，下面简单回顾一下。如果还没有系统地学习过动态规划，建议先熟读《算法竞赛入门经典》的第 9 章。本节是在该章基础之上的复习、拓宽与加深。

问题 1：数字三角形。如图 1-35（a）所示，有一个由非负整数组成的三角形，第一行只有一个数，除了最下行之外，每个数的左下方和右下方各有一个数。从第一行的数开始，每次可以往左下或右下走一格，直到走到最下行，把沿途经过的数全部加起来。如何走，可使得这个和最大？

分析：这是一个多段图上的最短路径问题，其中每行是一个阶段。设 $d(i,j)$为从格子(i,j)出发能得到的最大和，则 $d(i,j)=a(i,j)+\max\{d(i+1,j),d(i+1,j+1)\}$，边界是 $d(n+1,j)=0$，各个格子的编号如图 1-35（b）所示。



（a）　　（b）

图 1-35

问题 2：嵌套矩形。有 n 个矩形，每个矩形可以用两个整数 a, b 描述，表示它的长和宽。矩形 $X(a,b)$可以嵌套在矩形 $Y(c,d)$中的条件为：当且仅当 $a<c, b<d$，或者 $b<c, a<d$（相当于把矩形 X 旋转 90°）。例如，矩形(1,5)可以嵌套在矩形(6,2)内，但不能嵌套在矩形(3,4)内。选出尽量多的矩形排成一行，使得除了最后一个之外，每一个矩形都可以嵌套在下一个矩形内。

分析：本题是 DAG 最长路问题。设 $d(i)$为以矩形 i 结尾的最长链的长度，则 $d(i)=\max\{0,d(j)|$矩形 j 可以嵌套在矩形 i 中$\}+1$。

问题 3：硬币问题。有 n 种硬币，面值分别为 $V_1, V_2,\ldots, V_n$，每种都有无限多。给定非负整数 S，可以选用多少个硬币，使得面值之和恰好为 S？输出硬币数目的最小值和最大值。其中，$1\leqslant n\leqslant 100$，$0\leqslant S\leqslant 10\,000$，$1\leqslant V_i\leqslant S$。

分析：本题是 DAG 最长路和最短路问题。设 $f(i)$和 $g(i)$分别为面值之和恰好为 i 时，硬币数目的最小值和最大值，则 $f(i)=\min\{\infty, f(i-V_j)+1|V_j\leqslant i\}$，$g(i)=\max\{-\infty,g(i-V_j)+1|V_j\leqslant i\}$，边界条件是 $f(0)=g(0)=0$。

问题 4：01 背包问题。有 n 种物品，每种只有一个。第 i 种物品的体积为 V_i，重量为

W_i。选一些物品装到一个容量为 C 的背包，使得背包内物品在总体积不超过 C 的前提下重量尽量大。其中，$1 \leqslant n \leqslant 100$，$1 \leqslant V_i \leqslant C \leqslant 10\,000$，$1 \leqslant W_i \leqslant 10^6$。

分析：用 $f(i,j)$表示“把前 i 个物品装到容量为 j 的背包中的最大总重量”，则状态转移方程为 $f(i,j)=\max\{f(i-1,j), f(i-1,j-V_i)+W_i \mid V_i \leqslant j\}$，边界为 $f(0,j)=0$。可以使用滚动数组优化空间，最终程序如下。

```
memset(f, 0, sizeof(f));
for(int i = 1; i <= n; i++) {
  scanf("%d%d", &V, &W);
  for(int j = C; j >= 0; j--) if(j >= V) f[j] = max(f[j], f[j-V]+W);
}
```

它的道理蕴含在图 1-36 中。

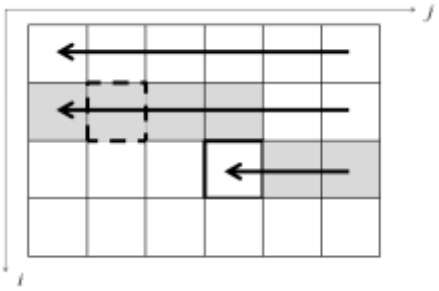


图　1-36

f 是从上到下、从右到左（而不是从左到右）计算的，所以不会覆盖到以后需要的值。

问题 5：点集配对问题。空间里有 n 个点 $P_0, P_1, \ldots, P_{n-1}$，把它们配成 $n/2$ 对（n 是偶数），使得每个点恰好在一个点对中。要求所有点对中，两点的距离之和应尽量小。其中，$n \leqslant 20$，$|x_i|,|y_i|,|z_i| \leqslant 10\,000$。

分析：设 $d(S)$为集合 S 配对后的最小距离和，则

$$d(S)=\min\{d(S-\{i\}-\{j\})+|P_iP_j| \mid j \in S, j>i, i=\min\{S\}\}$$

再次强调，由于 S 中的最小元素 i 无论如何都是要配对的，所以无须枚举（否则时间复杂度会多乘上一个 n）。为了进一步帮助读者理解，这里画出状态转移图的一部分，如图 1-37 所示。

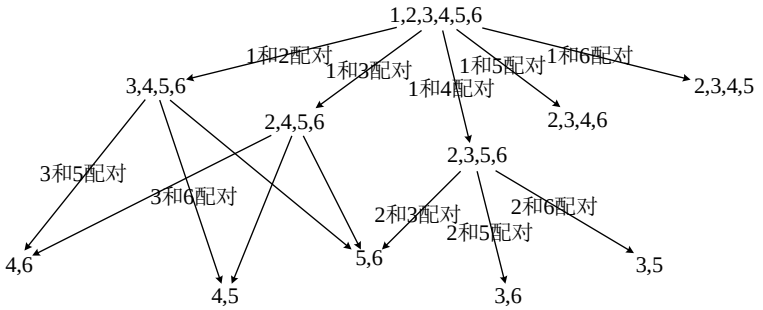


图 1-37

其中，$d(\{2,3,5,6\})=\min\{d(\{5,6\})+|P_2P_3|, d(\{3,6\})+|P_2P_5|, d(\{3,5\})+|P_2P_6|\}$，$d(\{1,2,3,4,5,6\})=\min\{d(\{3,4,5,6\})+|P_1P_2|, d(\{2,4,5,6\})+|P_1P_3|, \ldots, d(\{2,3,4,5\})|P_1P_6|\}$。因为 1 必须配对，因此{1,2,3,4,5,6}的这 5 种决策涵盖了所有情况。程序中，集合用二进制表示，在《入门经典》中已有详细描述。

问题 6：最长上升子序列问题（LIS）。给定 n 个整数 $A_1, A_2, \ldots, A_n$，按从左到右的顺序选出尽量多的整数，组成一个上升子序列（子序列可以理解为：删除 0 个或多个数，其他数的顺序不变）。比如，从序列 1, 6, 2, 3, 7, 5 中，可以选出上升子序列 1, 2, 3, 5，也可以选出 1, 6, 7，但前者更长。选出的上升子序列中相邻元素不能相等。

分析：设 $d(i)$为以 i 结尾的最长上升子序列的长度，则 $d(i)=\max\{0,d(j)|j<i, A_j<A_i\}+1$，最终答案是 $\max\{d(i)\}$。如果 LIS 中的相邻元素可以相等，把小于号改成小于等于号即可。上述算法的时间复杂度为 $O(n^2)$，下面介绍一种可把时间复杂度优化到 $O(n\log n)$的方法。

假设已经计算出的两个状态 a 和 b 满足 $A_a<A_b$ 且 $d(a)=d(b)$，则对于后续所有状态 i（即 $i>a$ 且 $i>b$）来说，a 并不会比 b 差——如果 b 满足 $A_b<A_i$ 的条件，a 也满足，且二者的 d 值相同；但反过来却不一定了，a 满足 $A_a<A_i$ 的条件时，b 却不一定满足。换句话说，如果我们只保留 a，一定不会丢失最优解。

这样，对于相同的 d 值，只需保留 A 最小的一个。我们用 $g(i)$表示 d 值为 i 的最小状态编号[①]（如果不存在，$g(i)$定义为正无穷）。根据上述推理可以证明

$$g(1) \leqslant g(2) \leqslant g(3) \leqslant \ldots \leqslant g(n)$$

需要特别注意的是，上述 g 值是动态改变的。对于一个给定的状态 i，我们只考虑在 i 之前已经计算过的状态 j（即 $j<i$），上述 g 序列也是基于这些状态的。随着 i 的不断增大，我们要考虑的状态越来越多，g 也随之发生改变。在给定状态 i 时，可以用二分查找得到满足 $g(k)\geqslant A_i$ 的第一个下标 k，则 $d(i)=k$[②]，此时 $A_i<g(k)$，而 $d(i)=k$，所以更新 $g(k)=A_i$。

```
for(int i = 1; i <= n; i++) g[i] = INF;
for(int i = 0; i < n; i++) {
  int k = lower_bound(g+1, g+n+1, A[i]) - g; // 在g[1]到g[n]中找
  d[i] = k;
  g[k] = A[i];
```

① 即满足 $d(j)=i$ 的最小 j。

② 实际上是要找满足 $g[k']<A[i]$的最后一个下标 k'，则 $d(i)=k'+1$，令 $k=k'+1$ 即可得到。

}

问题 7：最长公共子序列问题（LCS）。给出两个子序列 A 和 B，如图 1-38 所示，求长度最大的公共子序列。比如，1, 5, 2, 6, 8, 7 和 2, 3, 5, 6, 9, 8, 4 的最长公共子序列为 5, 6, 8（另一个解是 2, 6, 8）。

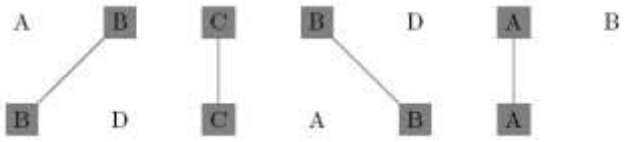


图　1-38

分析：设 $d(i,j)$为 $A_1,A_2,\ldots,A_i$ 和 $B_1,B_2,\ldots,B_j$ 的 LCS 长度，则当 $A[i]=A[j]$时，$d(i,j)=d(i-1,j-1)+1$；否则，$d(i,j)=\max\{d(i-1,j),d(i,j-1)\}$。时间复杂度为 $O(nm)$[①]，其中 n 和 m 分别是序列 A 和 B 的长度。LCS 问题也可以用滚动数组法进行优化。

问题 8：最大连续和。给出一个长度为 n 的序列 $A_1, A_2,\ldots, A_n$，求一个连续子序列 $A_i, A_{i+1},\ldots A_j$，使得元素总和 $A_i+A_{i+1}+\ldots+A_j$ 最大。

分析：本题在《算法竞赛入门经典》中已经给出了一个利用前缀和的线性时间算法。用动态规划可以得到另一个线性算法：设 $d(i)$为以 i 结尾的最大连续和，则 $d(i)=\max\{0, d(i-1)\}+A[i]$。

问题 9：货郎担问题（TSP）。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 之间的道路长度 $L_{i,j}$，求一条经过每个城市一次且仅一次，最后回到起点的路线，使得经过的道路总长度最短。其中，$n\leqslant 15$，城市编号为 0~$n-1$。

分析：TSP 是一道经典的 NPC 难题，不过因为本题规模小，可以用动态规划求解。首先注意到可以直接规定起点和终点为城市 0（想一想，为什么），然后设 $d(i,S)$表示当前在城市 i，访问集合 S 中的城市各一次后回到城市 0 的最短长度，则

$$d(i,S)=\min\{d(j,S-\{j\})+dist(i,j)\mid j\in S\}$$

边界为 $d(i,\{\})=dist(0,i)$。最终答案是 $d(0,\{1,2,3,\ldots,n-1\})$，时间复杂度为 $O(n^2 2^n)$。

问题 10：矩阵链乘（MCM）。一个 $n\times m$ 矩阵由 n 行 m 列共 $n\times m$ 个数排列而成。两个矩阵 A 和 B 可以相乘的条件为：当且仅当 A 的列数等于 B 的行数。一个 $n\times m$ 的矩阵乘以一个 $m\times p$ 的矩阵等于一个 $n\times p$ 的矩阵，运算量为 mnp。

矩阵乘法不满足分配律，但满足结合律，因此 $A\times B\times C$ 既可以按顺序 $(A\times B)\times C$ 进行，也可以按 $A\times(B\times C)$ 来进行。假设 A、B、C 分别是 2×3，3×4 和 4×5 矩阵，则 $(A\times B)\times C$ 的运算量为 $2\times 3\times 4+2\times 4\times 5=64$，$A\times(B\times C)$ 的运算量为 $3\times 4\times 5+2\times 3\times 5=90$。显然，第一种运算顺序更节省运算量。

给出 n 个矩阵组成的序列，设计一种方法把它们依次相乘，使得总运算量最小。假设第 i 个矩阵 A_i 是 $p_{i-1}\times p_i$ 的。

分析：设 $f(i,j)$表示把 $A_i, A_{i+1},\ldots, A_j$ 乘起来所需要的乘法次数，枚举“最后一次乘法”是

[①] 事实上，LCS 问题存在渐进时间复杂度比 $O(nm)$更低的算法，但超出了本书的范围。

第 k 个乘号，则 $f(i,j)=\max\{f(i,k)+f(k+1,j)+p_{i-1}p_kp_j\}$，边界是 $f(i,i)=0$，时间复杂度为 $O(n^3)$[①]。

问题 11：最优排序二叉树问题（OBST）。给 n 个符号建立一棵排序二叉树[②]。虽然平衡树的高度最小，但如果各个符号的频率相差很大，平衡反而不好。比如，若有 7 个符号 ABCDEFG，频率分别为 729, 243, 81, 27, 9, 3, 1，如图 1-39 所示，则下面的平衡树的总检索次数（即所有关键字频率和深度的乘积之和）为 27×1+(243+2)×2+(729+81+9+1)×3=2977。

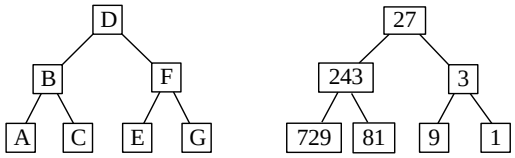


图 1-39

相比之下，图 1-40 所示的链状树反而好得多。

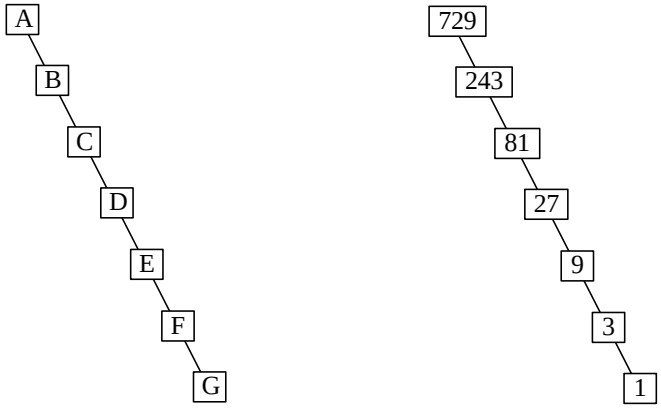


图 1-40

它的总检索次数仅为 1 636 次。给定 n 个关键字的频率 $f_1, f_2, \ldots, f_n$，要求构造一棵最优的排序二叉树，使得每个关键字的频率和深度的乘积之和最小。

分析：根据排序二叉树的递归定义，可以先选根，然后递归建立左右子树。记 $d(i,j)$为符号 $i, i+1,\ldots, j$ 所建立的排序二叉树的最小检索次数。如果选根为 k，总检索次数应该如何计算？

树根 k 只需要检索一次，累加上 f_k。左子树在单独作为一棵树时，其总检索次数为 $d(i,k-1)$；但在作为 k 的子树后，所有结点的深度都增加了 1，因此总检索次数需要加上 $f_i+f_{i+1}+\ldots+f_{k-1}$。右子树类似。这样，若记 $w(i,j)=f_i+f_{i+1}+\ldots+f_j$，状态转移方程为 $d(i,j)=\max\{d(i,k-1)+d(k+1,j)\}+w(i,j)$。状态有 $O(n^2)$个，每个状态的决策有 $O(n)$个，总时间复杂度为 $O(n^3)$。

有一个方法可以把时间复杂度降为 $O(n^2)$。记 $K(i,j)$为让 $d(i,j)$取到最小值的决策，则可以证明 $K(i,j) \leqslant K(i,j+1) \leqslant K(i+1,j+1)$（$i \leqslant j$），即 K 在同行和同列上都是递增的。证明需要

① 事实上，本问题存在 $O(n\log n)$时间的算法，但超出了本书的范围。
② 详见第 3 章。

用到四边形不等式，这里略去，有兴趣的读者可以自行查阅相关资料。

有了这个结论，我们在计算 $d(i,j)$时，只需把决策枚举从 $i\sim j$ 改成从 $K(i,j-1)\sim K(i+1,j)$即可。注意到后面两个状态都在 $d(i,j)$之前已经算过，所以 $K(i,j-1)$和 $K(i+1,j)$已经得到。

下面分析时间复杂度。当 $L=j-i$ 固定时，

$d(1,L+1)$的决策是 $K(1,L)\sim K(2,L+1)$

$d(2,L+2)$的决策是 $K(2,L+1)\sim K(3,L+2)$

$d(3,L+3)$的决策是 $K(3,L+2)\sim K(4,L+3)$

…

全部合并起来，当 L 固定时的总决策为 $K(1,L)\sim K(n-L+1,n)$，共 $O(n)$个。由于 L 有 $O(n)$个，总时间复杂度降为 $O(n^2)$。

例题 26　约瑟夫问题的变形（And Then There Was One, Japan 2007, LA 3882）

n 个数排成一个圈。第一次删除 m，以后每数 k 个数删除一次，求最后一个被删除的数。当 n=8, k=5, m=3 时，删数过程如图 1-41 所示。

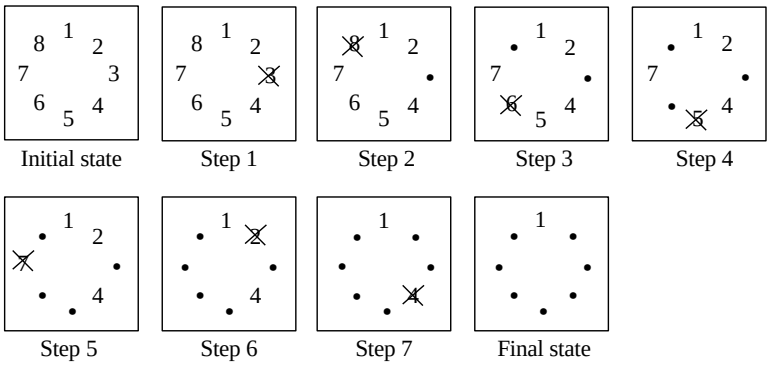


图　1-41

【输入格式】

输入包含多组数据。每组数据包含 3 个整数 n, k, m（$2\leqslant n\leqslant 10\,000$，$1\leqslant k\leqslant 10\,000$，$1\leqslant m\leqslant n$）。输入结束标志为 $n=k=m=0$。

【输出格式】

对于每组数据，输出最后一个被删除的数。

【分析】

本题是约瑟夫问题的变种，唯一的区别就是：原版问题中，从 1 开始数数，而在本题中，规定第一个删除的数是 m。约瑟夫问题作为链表的经典应用，出现在很多数据结构与程序设计语言的书籍中。可惜链表法的时间复杂度为 $O(nk)$，无法承受本题这样大的规模。

如果像本题这样只关心最后一个被删除的编号，而不需要完整的删除顺序，则可以用递推法求解。假设编号为 0~$n-1$ 的 n 个数排成一圈，从 0 开始每 k 个数删除一个，最后留下的数字编号记为 $f(n)$，则 $f(1)=0$，$f(n)=(f(n-1)+k)\ \%\ n$。为什么呢？因为删除一个元素之后，可以把所有元素重新编号，如图 1-42 所示。

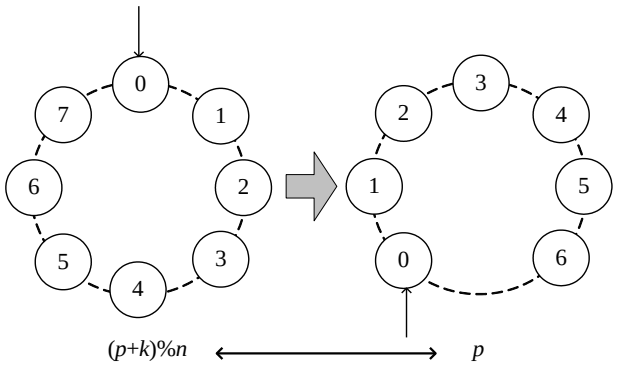


图 1-42

本题第一个删除的为 m，因此答案为$(m - k + 1 + f[n]) \% n$[①]。注意，本题虽然不是动态规划，但思路是相通的[②]。代码如下。

```
#include<cstdio>
const int maxn = 10000 + 2;
int f[maxn];

int main() {
  int n, k, m;
  while(scanf("%d%d%d", &n, &k, &m) == 3 && n) {
    f[1] = 0;
    for(int i = 2; i <= n; i++) f[i] = (f[i-1] + k) % i;
    int ans = (m - k + 1 + f[n]) % n;
    if (ans <= 0) ans += n;
    printf("%d\n", ans);
  }
  return 0;
}
```

例题 27　王子和公主（Prince and Princess, UVa 10635）

有两个长度分别为 p+1 和 q+1 的序列，每个序列中的各个元素互不相同，且都是 1~n^2 之间的整数。两个序列的第一个元素均为 1。求出 A 和 B 的最长公共子序列长度。

【输入格式】

输入的第一行为数据组数 T（T≤10）。每组数据包含 3 行，第一行为 3 个整数 n, p, q（2≤n≤250，1≤p,q≤n^2）；第二行包含序列 A，其中第一个数为 1，其元素两两不同，且都是 1~n^2 之间的整数；第三行包含序列 B，格式同序列 A。

【输出格式】

对于每组数据，输出 A 和 B 的最长公共子序列的长度。

[①] 需要把这个数改成 1~n 之间的。

[②] 事实上，很多人习惯把所有递推都叫做动态规划，不管它是否真的是在解决最优化问题。

【分析】

本题是 LCS 问题，但因为 p 和 q 可以高达 250^2=62 500，$O(pq)$的算法显然太慢。注意到 A 序列中所有元素均不相同，因此可以把 A 中元素重新编号为 1~p+1。例如，样例中 A={1,7,5,4,8,3,9}，B={1,4,3,5,6,2,8,9}，因此把 A 重新编号为{1,2,3,4,5,6,7}，则 B 就是{1,4,6,3,0,0,5,7}，其中 0 表示 A 中没有出现过（事实上，可以直接删除这些元素，因为它们肯定不在 LCS 中）。这样，新的 A 和 B 的 LCS 实际上就是新的 B 的 LIS。由于 LIS 可在 $O(n\log n)$时间内解决，因此本题也可在 $O(n\log n)$时间内得到解决。代码如下。

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

const int maxn = 250 * 250;
const int INF = 1000000000;
int S[maxn], g[maxn], d[maxn]; //LIS 所需
int num[maxn]; //num[x]为整数 x 的新编号,num[x]=0 表示 x 没有在 A 中出现过

int main() {
  int T;
  scanf("%d", &T);
  for(int kase = 1; kase <= T; kase++) {
    int N, p, q, x;
    scanf("%d%d%d", &N, &p, &q);
    memset(num, 0, sizeof(num));
    for(int i = 1; i <= p+1; i++) { scanf("%d", &x); num[x] = i; }
    int n = 0;
    for(int i = 0; i < q+1; i++) { scanf("%d", &x); if(num[x]) S[n++] = num[x]; }

    //求解 S[0]...S[n-1]的 LIS
    for(int i = 1; i <= n; i++) g[i] = INF;
    int ans = 0;
    for(int i = 0; i < n; i++) {
      int k = lower_bound(g+1, g+n+1, S[i]) - g; //在 g[1]~g[n]中查找
      d[i] = k;
      g[k] = S[i];
      ans = max(ans, d[i]);
    }
    printf("Case %d: %d\n", kase, ans);
  }
  return 0;
}
```

例题 28　Sum 游戏（Game of Sum, UVa 10891）

有一个长度为 n 的整数序列，两个游戏者 A 和 B 轮流取数，A 先取。每次玩家只能从左端或者右端取一个数，但不能两端都取。所有数都被取走后游戏结束，然后统计每个人取走的所有数之和，作为各自的得分。两个人采取的策略都是让自己的得分尽量高，并且两人都足够聪明，求 A 的得分减去 B 的得分后的结果。

【输入格式】

输入包含多组数据。每组数据的第一行为正整数 n（$1 \leqslant n \leqslant 100$），第二行为给定的整数序列。输入结束标志为 $n=0$。

【输出格式】

对于每组数据，输出 A 和 B 都采取最优策略的情况下，A 的得分减去 B 的得分后的结果。

【分析】

整数的总和是一定的，所以一个人得分越高，另一个人的得分就越低。不管怎么取，任意时刻游戏的状态都是原始序列的一段连续子序列（即被两个玩家取剩下的序列）。因此，我们想到用 $d(i,j)$表示原序列的第 i~j 个元素组成的子序列（元素编号为 1~n），在双方都采取最优策略的情况下，先手得分的最大值（只考虑 i~j 这些元素）。

状态转移时，我们需要枚举从左边取还是从右边取以及取多少个。这等价于枚举给对方剩下怎样的子序列：是(k,j)（$i<k\leqslant j$），还是(i,k)（$i\leqslant k<j$）。因此：

$$d(i,j)= \text{sum}(i,j)-\min\{d(i+1,j), d(i+2,j),\ldots, d(j,j), d(i,j-1), d(i,j-2),\ldots, d(i,i), 0\}$$

其中，sum(i,j)是元素 i 到元素 j 的数之和。注意，这里的“0”是“取完所有数”的决策，有了它，方程就不需要显式的边界条件了。

两人得分之和为 sum$(1,n)$，因此答案是 $d(1,n)-(\text{sum}(1,n)-d(1,n))=2d(1,n)-\text{sum}(1,n)$。注意，sum$(i,j)$的计算不需要循环累加，可以预处理 $S[i]$为前 i 个数之和，则 sum$(i,j)=S[j]-S[i-1]$。

下面是完整代码。它采用了记忆化搜索的方式，显得更加自然。

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

const int maxn = 100 + 10;
int S[maxn], A[maxn], d[maxn][maxn], vis[maxn][maxn], n;

int dp(int i, int j) {
  if(vis[i][j]) return d[i][j];
  vis[i][j] = 1;
  int m = 0;                          //全部取光
  for(int k = i+1; k <= j; k++) m = min(m, dp(k,j));
  for(int k = i; k < j; k++) m = min(m, dp(i,k));
  d[i][j] = S[j]-S[i-1] - m;          //如果 i 从 0 开始编号，这里得判断一下是否 i==0
```

```
  return d[i][j];
}

int main() {
  while(scanf("%d", &n) && n) {
    S[0] = 0;
    for(int i = 1; i <= n; i++) { scanf("%d", &A[i]); S[i]=S[i-1]+A[i]; }
    memset(vis, 0, sizeof(vis));    //千万不要漏掉
    printf("%d\n", 2*dp(1,n)-S[n]);
  }
  return 0;
}
```

状态有 $O(n^2)$个，每个状态有 $O(n)$个转移，所以时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$。对于本题的规模，这样的时间复杂度已经不错了，但其实还可以进一步改进。让我们回顾一下状态转移方程：

$$d(i,j)= \text{sum}(i,j)-\min\{d(i+1,j), d(i+2,j),\ldots, d(j,j), d(i,j-1), d(i,j-2),\ldots, d(i,i), 0\}$$

如果令 $f(i,j)= \min\{d(i,j),d(i+1,j),\ldots,d(j,j)\}$，$g(i,j)= \min\{d(i,j),d(i,j-1),\ldots,d(i,i)\}$，则状态转移方程可以写成：

$$d(i,j)= \text{sum}(i,j)-\min\{f(i+1,j), g(i,j-1), 0\}$$

f 和 g 也可以快速递推出来：$f(i,j) = \min\{d(i,j), f(i+1,j)\}$，$g(i,j) = \min\{d(i,j), g(i,j-1)\}$，因此每个 $f(i,j)$的计算时间都降为了 $O(1)$。下面我们用递推（而非记忆化搜索）的方法编写。代码如下。

```
for(int i = 1; i <= n; i++) f[i][i] = g[i][i] = d[i][i] = A[i]; //边界
for(int L = 1; L < n; L++)      //按照 L=j-i 递增的顺序计算
  for(int i = 1; i+L <= n; i++) {
    int j = i+L;
    int m = 0;                  // m = min{f(i+1,j), g(i,j-1), 0}
    m = min(m, f[i+1][j]);
    m = min(m, g[i][j-1]);
    d[i][j] = S[j]-S[i-1] - m;
    f[i][j] = min(d[i][j], f[i+1][j]); //递推 f 和 g
    g[i][j] = min(d[i][j], g[i][j-1]);
  }
printf("%d\n", 2*d[1][n]-S[n]);
```

新算法的时间复杂度为 $O(n^2)$。

例题 29　黑客的攻击（Hacker's Crackdown, UVa 11825）

假设你是一个黑客，侵入了一个有着 n 台计算机（编号为 0, 1,…, $n-1$）的网络。一共有 n 种服务，每台计算机都运行着所有服务。对于每台计算机，你都可以选择一项服务，终止这台计算机和所有与它相邻计算机的该项服务（如果其中一些服务已经停止，则这些

服务继续处于停止状态）。你的目标是让尽量多的服务器完全瘫痪（即：没有任何计算机运行该项服务）。

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 16$）；以下 n 行每行描述一台计算机的相邻计算机，其中第一个数 m 为相邻计算机个数，接下来的 m 个整数为这些计算机的编号。输入结束标志为 $n=0$。

【输出格式】

对于每组数据，输出完全瘫痪的服务器的最大数量。

【分析】

本题的数学模型是：把 n 个集合 $P_1, P_2, \ldots, P_n$ 分成尽量多组，使得每组中所有集合的并集等于全集。这里的集合 P_i 就是计算机 i 及其相邻计算机的集合，每组对应于题目中的一项服务。注意到 n 很小，可以用《算法竞赛入门经典》中提到的二进制法表示这些集合，即在代码中，每个集合 P_i 实际上是一个非负整数。输入部分代码如下。

```
for(int i = 0; i < n; i++) {
  int m, x;
  scanf("%d", &m);
  P[i] = 1<<i;
  while(m--) { scanf("%d", &x); P[i] |= (1<<x); }
}
```

为了方便，我们用 cover(S)表示若干 P_i 的集合 S 中所有 P_i 的并集（二进制表示），即这些 P_i 在数值上的“按位或”。

```
for(int S = 0; S < (1<<n); S++) {
  cover[S] = 0;
  for(int i = 0; i < n; i++)
    if(S & (1<<i)) cover[S] |= P[i];
}
```

不难想到这样的动态规划：用 $f(S)$表示子集 S 最多可以分成多少组，则

$$f(S)=\max\{f(S_0)|S_0 \text{ 是 } S \text{ 的子集，cover}[S_0]\text{等于全集}\}+1$$

这里有一个重要的技巧：枚举 S 的子集 S_0。详见下面的代码。

```
f[0] = 0;
int ALL = (1<<n) - 1;
for(int S = 1; S < (1<<n); S++) {
  f[S] = 0;
  for(int S0 = S; S0; S0 = (S0-1)&S)
    if(cover[S0] == ALL) f[S] = max(f[S], f[S^S0]+1);
}
printf("Case %d: %d\n", ++kase, f[ALL]);
```

如何分析上述算法的时间复杂度呢？它等于全集{1, 2,…, n}的所有子集的子集个数之和，也可以令 $c(S)$表示集 S 的子集的个数（它等于 $2^{|S|}$），则本题的时间复杂度为 sum{$c(S_0)$ | S_0 是{1,2,3,…,n}的子集}。

注意到元素个数相同的集合，其子集个数也相同，我们可以按照元素个数“合并同类项”。元素个数为 k 的集合有 $C(n,k)$个，其中每个集合有 2^k 个子集，因此本题的时间复杂度为 sum{$C(n,k)2^k$}=$(2+1)^n=3^n$，其中第一个等号用到了二项式定理（不过是反着用的）。

本题比较抽象，但对思维训练很有帮助，希望读者花点时间将它彻底搞懂。

例题 30　放置街灯（Placing Lampposts, UVa 10859）

给你一个 n 个点 m 条边的无向无环图，在尽量少的结点上放灯，使得所有边都被照亮。每盏灯将照亮以它为一个端点的所有边。在灯的总数最小的前提下，被两盏灯同时照亮的边数应尽量大。

【输入格式】

输入的第一行为测试数据组数 T（T≤30）。每组数据第一行为两个整数 n 和 m（m< n≤1 000），即点数（所有点编号为 0~n−1）和边数；以下 m 行每行为两个不同的整数 a 和 b，表示有一条边连接 a 和 b（0≤a，b≤n）。

【输出格式】

对于每组数据，输出 3 个整数，即灯的总数、被两个灯照亮的边数和只被一个灯照亮的边数。

【分析】

无向无环图的另一个说法是“森林”，它由多棵树组成。动态规划是解决树上的优化问题的常用工具，本题就是一个很好的例子。

首先，本题的优化目标有两个：放置的灯数 a 应尽量少，被两盏灯同时照亮的边数 b 应尽量大。为了统一起见，我们把后者替换为：恰好被一盏灯照亮的边数 c 应尽量小，然后改用 $x=Ma+c$ 作为最小化的目标，其中 M 是一个很大的正整数。当 x 取到最小值时，x/M 的整数部分就是放置的灯数的最小值；$x\%M$ 就是恰好被一盏灯照亮的边数的最小值。

一般来说，如果有两个需要优化的量 v_1 和 v_2，要求首先满足 v_1 最小，在 v_1 相同的情况下 v_2 最小，则可以把二者组合成一个量 Mv_1+v_2，其中 M 是一个比“v_2 的最大理论值和 v_2 的最小理论值之差”还要大的数。这样，只要两个解的 v_1 不同，则不管 v_2 相差多少，都是 v_1 起到决定性作用；只有当 v_1 相同时，才取决于 v_2。在本题中，可以取 M=2 000[①]。

每棵树的街灯互不相干，因此可以单独优化，最后再把答案加起来即可。下面我们只考虑一棵树的情况。首先对这棵树进行 DFS，把无根树转化为有根树，然后试着设状态 $d(i)$ 为以 i 为根的子树的最小 x 值，看看能不能写出状态转移方程。

决策只有两种：在 i 处放灯和不在 i 处放灯。后继状态是 i 的各个子结点。可是问题来了：i 处是否放灯将影响到其子结点的决策。因此，我们需要把“父结点处有没有放灯”加入状态表示中。新状态为：$d(i,j)$表示 i 的父结点“是否放灯”的值为 j（1 表示放灯，0 表示没放），以 i 为根的树的最小 x 值（算上 i 和其父结点这条边）。

注意到各子树可以独立决策，因此可作出如下决策。

[①] M 不要取得太大，以免算术运算溢出。

决策一：结点 i 不放灯。必须 j=1 或者 i 是根结点时才允许作这个决策。此时 $d(i,j)$等于 sum{$d(k,0)|k$ 取遍 i 的所有子结点}。如果 i 不是根，还得加上 1，因为结点 i 和其父结点这条边上只有一盏灯照亮。

决策二：结点 i 放灯。此时 $d(i,j)$等于 sum{$d(k,1)|k$ 取遍 i 的所有子结点} + M。如果 j=0 且 i 不是根，还得加上 1，因为结点 i 和其父结点这条边只有一盏灯照亮。

用数学式子很难表达上面的状态转移，但用程序表达却可以很清晰。

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

vector<int> adj[1010]; //森林是稀疏的，这样保存省空间，枚举相邻结点也更快
int vis[1010][2], d[1010][2], n, m;

int dp(int i, int j, int f) {
//在 DFS 的同时进行动态规划，f 是 i 的父结点，它不存入状态里
  if(vis[i][j]) return d[i][j];
  vis[i][j] = 1;
  int& ans = d[i][j];

  //放灯总是合法决策
  ans = 2000;                //灯的数量加 1，x 加 2000
  for(int k = 0; k < adj[i].size(); k++)
    if(adj[i][k] != f)       //这个判断非常重要！除了父结点之外的相邻结点才是子结点
      ans += dp(adj[i][k], 1, i); //注意，这些结点的父结点是 i
  if(!j && f >= 0) ans++;    //如果 i 不是根，且父结点没放灯，则 x 加 1

  if(j || f < 0) {           //i 是根或者其父结点已放灯，i 才可以不放灯
    int sum = 0;
    for(int k = 0; k < adj[i].size(); k++)
      if(adj[i][k] != f)
        sum += dp(adj[i][k], 0, i);
    if(f >= 0) sum++;        //如果 i 不是根，则 x 加 1
    ans = min(ans, sum);
  }
  return ans;
}

int main() {
  int T, a, b;
  scanf("%d", &T);
  while(T--) {
    scanf("%d%d", &n, &m);
```

```
    for(int i = 0; i < n; i++) adj[i].clear();
    //adj 里保存着上一组数据的值，必须清空
    for(int i = 0; i < m; i++) {
      scanf("%d%d", &a, &b);
      adj[a].push_back(b);
      adj[b].push_back(a);   //因为是无向图
    }
    memset(vis, 0, sizeof(vis));
    int ans = 0;
    for(int i = 0; i < n; i++)
      if(!vis[i][0])         //新的一棵树
        ans += dp(i,0,-1);   //i 是树根，因此父结点不存在（-1）
    printf("%d %d %d\n", ans/2000, m-ans%2000, ans%2000); //从 x 计算 3 个整数
  }
  return 0;
}
```

例题 31　捡垃圾的机器人（Robotruck, SWERC 2007, LA 3983）

有 n 个垃圾，第 i 个垃圾的坐标为(x_i,y_i)，重量为 w_i。有一个机器人，要按照编号从小到大的顺序捡起所有垃圾并扔进垃圾桶（垃圾桶在原点(0,0)）。机器人可以捡起几个垃圾以后一起扔掉，但任何时候其手中的垃圾总重量不能超过最大载重 C。两点间的行走距离为曼哈顿距离（即横坐标之差的绝对值加上纵坐标之差的绝对值）。求出机器人行走的最短总路程（一开始，机器人在(0,0)处）。

【输入格式】

输入的第一行为数据组数。每组数据的第一行为最大承重 C（$1 \leqslant C \leqslant 100$）；第二行为正整数 n（$1 \leqslant n \leqslant 100\,000$），即垃圾的数量；以下 n 行每行为两个非负整数 x, y 和一个正整数 w，即坐标和重量（重量保证不超过 C）。

【输出格式】

对于每组数据，输出总路径的最短长度。

【分析】

如果把“当前垃圾序号”和“当前载重量”作为状态，则状态个数就已经高达 $O(NC)$，不管怎样优化状态转移，时间也无法承受。迫不得已，我们只得设 $d(i)$为从原点出发、将前 i 个垃圾清理完并放进垃圾筒的最小距离，则

$$d[i] = \min\{d[j] + \text{dist2origin}(j+1) + \text{dist}(j+1,i) + \text{dist2origin}(i) \mid j \leqslant i,\ w(j+1,i) \leqslant C\}$$

其中，dist(j+1,i)表示从第 j+1 个垃圾出发，依次经过垃圾 j+2、垃圾 j+3、…、最终到达垃圾 i 的总距离，dist2origin(i)表示垃圾 i 到原点的距离（即$|x_i|+|y_i|$），w(i,j)表示第 i~j 个垃圾的总重量。

设 total_dist(i)为从第 1 个垃圾开始，依次经过垃圾 2, 3,…，最终到达垃圾 i 的总距离，则 dist(j+1,i) = total_dist(i) – total_dist(j+1)。这样，上式可以改写为：

$$d[i] = \min\{d[j] - \text{total\_dist}(j+1) + \text{dist2origin}(j+1) \mid w(j+1,i) \leqslant C\} + \text{total\_dist}(i) + \text{dist2origin}(i)$$

如果令 func(j)=d[j]−total_dist(j+1)+dist2origin(j+1)，上式还可以进一步简化为

$$d[i] = \min\{\text{func}(j) \mid w(j+1,i) \leqslant C\} + \text{total\_dist}(i) + \text{dist2origin}(i)$$

其中，阴影部分是问题的关键。注意到满足 $w(j+1,i) \leqslant C$ 的所有 j 形成一个区间，而且随着 i 的增大，这个区间会往右移动（因为所有 w_i 均为正数），我们常常把这个区间称为滑动窗口，则问题就转化为：维护一个滑动窗口中的最小值。

当滑动窗口的右边界增大时，相当于往滑动窗口里添加新元素；当滑动窗口的左边界增大时，相当于往滑动窗口里删除元素。这样，我们可以用一个数据结构维护滑动窗口，要求支持插入、删除、取最小值。在学习完本书的数据结构部分后，相信读者能够找到一个合适的数据结构，在 $O(\log n)$时间内完成上述 3 种操作。但其实这并不是最高效的方法。

假设滑动窗口中有两个元素 1 和 2，且 1 在 2 的右边，会怎样？这意味着 2 在离开窗口之前一直会被 1 给“压迫着”，永远不可能成为最小值。换句话说，这个 2 是无用的，应当及时删除。当删除掉无用元素之后，滑动窗口中剩下的东西（有用元素）从左到右是递增的。我们把这些元素看成一个队列[①]，每次有新元素进来时，需要删除所有比新元素大的元素，如图 1-43 所示。

2 4 6 ~~8 10~~ 7

图 1-43

还需要及时把不在滑动窗口范围之内的元素移出队列。读者可能会问，如果老是要删除很多元素怎么办，时间复杂度会不会很差？不会的，因为每个元素最多被删除一次，所以总时间复杂度仍是 $O(n)$。

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

const int maxn = 100000 + 10;

int x[maxn], y[maxn];
int total_dist[maxn], total_weight[maxn], dist2origin[maxn];
int q[maxn], d[maxn];

int func(int i) {
  return d[i] - total_dist[i+1] + dist2origin[i+1];
}

main() {
  int T, c, n, w, front, rear;
  scanf("%d", &T);
  while(T--) {
```

[①] 队列中的元素递增，因此也称为单调队列。

```
        scanf("%d%d", &c, &n);
        total_dist[0] = total_weight[0] = x[0] = y[0] = 0;
        for(int i = 1; i <= n; i++) {
          scanf("%d%d%d", &x[i], &y[i], &w);
          dist2origin[i] = abs(x[i]) + abs(y[i]);
          total_dist[i] = total_dist[i-1] + abs(x[i]-x[i-1]) + abs(y[i]-y[i-1]);
          total_weight[i] = total_weight[i-1] + w;
        }
        front = rear = 1;
        for (int i = 1; i <= n; i++) {
          while (front <= rear && total_weight[i] - total_weight[q[front]] > c)
front++;
          d[i] = func(q[front]) + total_dist[i] + dist2origin[i];
          while (front <= rear && func(i) <= func(q[rear])) rear--;
          q[++rear] = i;
        }
        printf("%d\n", d[n]);
        if(T > 0) printf("\n");
      }
      return 0;
    }
```

例题 32　分享巧克力（Sharing Chocolate, World Finals 2010, LA 4794）

给出一块长为 x，宽为 y 的矩形巧克力，每次操作可以沿一条直线把一块巧克力切割成两块长宽均为整数的巧克力（一次不能同时切割多块巧克力）。

问：是否可以经过若干次操作得到 n 块面积分别为 $a_1,a_2,\ldots,a_n$ 的巧克力。如图 1-44 所示，可以把 3×4 的巧克力切成面积分别为 6, 3, 2, 1 的 4 块。

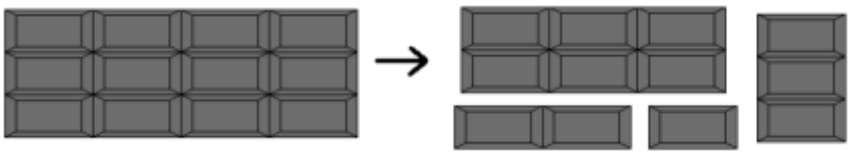


图　1-44

【输入格式】

输入包含若干组数据。每组数据的第一行为一个整数 n（$1\leqslant n\leqslant 15$）；第二行为两个整数 x 和 y($1\leqslant x, y\leqslant 100$)；第三行为 n 个整数 $a_1,a_2,\ldots,a_n$。输入结束标志为 n=0。

【输出格式】

对于每组数据，如果可以切割成功，输出“Yes”，否则输出“No”。

【分析】

注意到 n 的规模很小，可以把与 n 有关的子集作为动态规划状态的一部分。设 $f(r,c,S)$ 表示 r 行 c 列的巧克力是否可以切割成面积集合 S。样例 1 的答案为 Yes，即 $f(3,4,\{6,3,2,1\})=1$。

第一刀把巧克力切成了 3×3 和 3×1 两块，即 $f(3,3,\{6,2,1\})=f(3,1,\{3\})=1$。

不难得到下面的状态转移规则：$f(r,c,S)=1$ 当且仅当

- 存在 $1\leqslant r_0<r$ 和 S 的子集 S_0，使得 $f(r_0,c,S_0)=f(r-r_0,c,S-S_0)=1$，或者
- 存在 $1\leqslant c_0<c$ 和 S 的子集 S_0，使得 $f(r,c_0,S_0)=f(r,c-c_0,S-S_0)=1$。

前者对应横着切，后者对应竖着切。状态有 $O(xy2^n)$个，每个状态转移到 $O(x+y)$个状态，总时间复杂度为 $O((x+y)xy2^n)$，有些偏大。

其实，上述状态有些浪费。如果 $r\times c$ 不等于 S 中所有元素之和（记为 sum(S)），显然 $f(r,c,S)=0$。换句话说，我们可以只计算 $r*c$=sum(S)的状态 $f(r,c,S)$。另外，$f(r,c,S)=f(c,r,S)$，所以不妨设 $r\leqslant c$，然后用 $g(r,S)$代替 $f(r,c,S)$。这样，状态降为了 $O(x2^n)$个。在枚举决策时，一旦确定了 S_0，实际上可以计算出 r_0 或者 c_0（或者发现不存在这样的 r_0 或者 c_0），因此总决策数为 $O(x3^n)$，这也是本算法的时间复杂度。由于很多状态达不到，推荐用记忆化搜索实现，实际运算量往往远小于 $O(x3^n)$。

最后有一点需要注意，输入之后需要比较所有 a_i 之和是否为 $x\times y$（想一想，为什么）。代码如下。

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

const int maxn = 16;
const int maxw = 100 + 10;
int n, A[maxn], sum[1<<maxn], f[1<<maxn][maxw], vis[1<<maxn][maxw];

int bitcount(int x) { return x == 0 ? 0 : bitcount(x/2) + (x&1); }

int dp(int S, int x) {
  if(vis[S][x]) return f[S][x];
  vis[S][x] = 1;
  int& ans = f[S][x];
  if(bitcount(S) == 1) return ans = 1;
  int y = sum[S] / x;
  for(int S0 = (S-1)&S; S0; S0 = (S0-1)&S) {
    int S1 = S-S0;
    if(sum[S0]%x==0&&dp(S0,min(x,sum[S0]/x))&&dp(S1,min(x,sum[S1]/x)))
return ans = 1;
    if(sum[S0]%y==0&&dp(S0,min(y,sum[S0]/y))&&dp(S1,min(y,sum[S1]/y)))
return ans = 1;
  }
  return ans = 0;
}
```

```
int main() {
  int kase = 0, n, x, y;
  while(scanf("%d", &n) == 1 && n) {
    scanf("%d%d", &x, &y);
    for(int i = 0; i < n; i++) scanf("%d", &A[i]);

    //每个子集中的元素之和
    memset(sum, 0, sizeof(sum));
    for(int S = 0; S < (1<<n); S++)
      for(int i = 0; i < n; i++) if(S & (1<<i)) sum[S] += A[i];

    memset(vis, 0, sizeof(vis));
    int ALL = (1<<n) - 1;
    int ans;
    if(sum[ALL] != x*y || sum[ALL] % x != 0) ans = 0;
    else ans = dp(ALL, min(x,y));
    printf("Case %d: %s\n", ++kase, ans ? "Yes" : "No");
  }
  return 0;
}
```

1.5　小结与习题

本章介绍了不少问题求解和算法设计的方法和技巧。这些内容有难有易，不必强求第一次就全部看懂，需要反复阅读、细心体会。

1．问题求解策略

本章介绍了贪心法、暴力法、二分法等常用算法，以及各种思维方式。表 1-1 中列出了本章中的例题。

表　1-1

类　别	题　号	统　计	题目名称（英文）	备　注
例题 1	UVa11292	810/91.60%	The Dragon of Loowater	排序后用贪心法
例题 2	UVa11729	274/86.86%	Commando War	用贪心法求最优排列；用相邻交换法证明正确性
例题 3	UVa11300	109/58.72%	Spreading the Wealth	用代数法进行数学推导；中位数
例题 4	LA3708	177/85.88%	Graveyard	推理；参考系
例题 5	UVa10881	177/85.88%	Piotr’s Ants	等效变换；排序
例题 6	LA2995	87/78.16%	Image is Everything	三维坐标系；迭代更新

例题 7	UVa11464	87/78.16%	Even Party	部分枚举；递推
例题 8	LA3401	84/89.29%	Colored Cubes	部分枚举；贪心
例题 9	UVa11210	111/85.59%	Chinese Mahjong	回溯法；以中国麻将为背景
例题 10	UVa11384	333/96.10%	Help is needed for Dexter	问题转化；递归
例题 11	UVa10795	113/84.96%	A Different Task	汉诺塔问题；递归
例题 12	LA3971	277/89.17%	Assemble	二分法；贪心
例题 13	LA3635	329/87.54%	Pie	二分法
例题 14	UVa11520	650/96.92%	Fill the Square	求字典序最小的解；贪心
例题 15	LA3902	127/68.50%	Network	树上的最优化问题；贪心
例题 16	LA3177	45/77.78%	Beijing Guards	二分法；贪心

仅完成书中的例题还远远不够，下面将给出一定数量的习题，以方便读者练习和提高。限于篇幅，有些输入输出格式较为简单的题目省略了“输入格式”和“输出格式”两个部分，读者可自行参考英文原题。

首先是一些入门习题，如表 1-2 所示。

表 1-2

题　号	统　计	题目名称（英文）	备　注
UVa11636	2334/95.76%	Hello World!	（请读者独立思考）
UVa11039	747/91.16%	Building Designing	（请读者独立思考）
LA3213	103/86.41%	Ancient Cipher	（请读者独立思考）
LA3602	9/55.56%	DNA Consensus String	（请读者独立思考）
UVa10970	4760/97.44%	Big Chocolate	不需要动态规划
UVa10340	7935/87.54%	All in All	（请读者独立思考）

接下来是需要多一些思考的题目，如表 1-3 所示。

表 1-3

题　号	统　计	题目名称（英文）	备　注
UVa10382	816/73.04%	Watering Grass	经典模型；贪心
UVa10905	2445/66.58%	Children’s Game	贪心。容易想错！建议编程并提交
LA4254	6/83.33%	Processor	二分法
UVa11627	81/76.54%	Slalom	二分法
UVa11134	194/81.44%	Fabled Rooks	经典问题的变形
UVa11100	533/77.49%	The Trip, 2007	（请读者独立思考）
LA4725	44/47.73%	Airport	（请读者独立思考）
LA4850	32/90.62%	Installations	（请读者独立思考）
LA3266	24/83.33%	Tian Ji - The Horse Racing	有多种方法
UVa11389	758/87.73%	The Bus Driver Problem	有多种方法
LA4094	120/72.50%	Wonder Team	（请读者独立思考）
LA3303	110/88.18%	Songs	相邻交换法
LA2757	150/74.67%	Supermarket	经典问题
LA3507	112/75.00%	Keep the Customer Satisfied	经典问题的变形

LA4234	231/96.54%	Binary Clock	注意细节
LA4238	132/95.45%	Area of Polycubes	认真分析题目
LA4636	127/94.49%	Cubist Artwork	有趣的题目；逻辑推理

你好　世界！（Hello World!, UVa 11636）

你刚刚学会用“printf("Hello World!\n")”向世界问好了，因此非常兴奋，希望输出 n 条“Hello World”信息，但你还没有学习循环语句，因此只能通过复制/粘贴的方式用 n 条 printf 语句来解决。比如，经过一次复制/粘贴后，一条语句会变成两条语句，再经过一次复制/粘贴后，两条语句会变成 4 条语句……至少需要复制/粘贴几次，才能使语句的条数恰好为 n？输入 n（$0<n<10\,001$），输出最小复制/粘贴的次数。

提示：每次可以只复制/粘贴一部分语句。

设计建筑物（Building Designing, UVa 11039）

有 n 个绝对值各不相同的非 0 整数，选出尽量多的数，排成一个序列，使得正负号交替，且绝对值递增。输入整数 n（$1\leqslant n\leqslant 500\,000$）和 n 个整数，输出最长序列长度。

古老的密码（Ancient Cipher, NEERC 2004, LA 3213）

给定两个长度均为 n 的字符串，判断它们之间的 26 个字母能否一一对应，即做一个一一映射后使得两个字符串相同（比如 ABB 和 CDD 可以一一对应，方法是 A→C，B→D，但 ABC 和 DED 不能一一对应）。输入两个字符串，输出 YES 或者 NO。

DNA 序列（DNA Consensus String, Seoul 2006, LA 3602）

给定 m 个长度均为 n 的 DNA 序列，求一个 DNA 序列，使其到所有序列的总 Hamming 距离尽量小。两个等长字符串的 Hamming 距离等于字符不同的位置个数。如有多解，求字典序最小的解。输入整数 m 和 n（$4\leqslant m\leqslant 50$，$4\leqslant n\leqslant 1\,000$），以及 m 个长度为 n 的 DNA 序列（只包含字母 A、C、G、T），输出让 Hamming 距离最小的 DNA 序列和其对应的距离。

大块巧克力（Big Chocolate, UVa 10970）

把一个 m 行 n 列的矩形巧克力切成 mn 个 1×1 的方块，需要切几刀？每刀只能沿着直线把一块巧克力切成两部分（不能用一刀同时去切两块巧克力）。输入 m 和 n，输出最少需要的刀数。

子序列（All in All, UVa 10340）

输入两个字符串 s 和 t，判断是否可以从 t 中删除 0 个或多个字符（其他字符顺序不变），得到字符串 s。比如，abcde 可以得到 bce，但无法得到 dc。

喷水装置（Watering Grass, UVa 10382）

有一块草坪，长为 l，宽为 w。在其中心线的不同位置处装有 n 个点状的喷水装置。每个喷水装置 i 可将以它为中心，半径为 r_i 的圆形区域润湿（见图 1-45）。请选择尽量少的喷水装置，把整个草坪全部润湿。

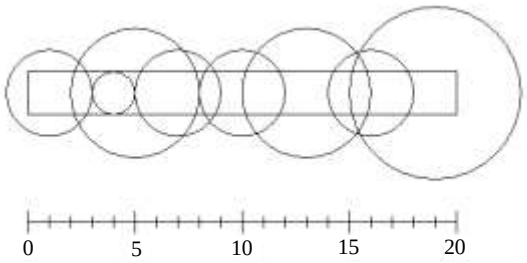


图 1-45

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n、l、w（$1 \leqslant n \leqslant 10\,000$）；以下 n 行每行包含两个整数 p_i 和 r_i，即每个喷水装置的位置和半径。

【输出格式】

对于每组数据，输出需要打开的喷水装置数目的最小值。如果无解，输出-1。

孩子们的游戏（Children's Game, UVa 10905）

给定 n 个正整数，你的任务是把它们连接成一个最大的整数。比如，123、124、56、90 有 24 种连接方法，最大的结果是 9 056 124 123。

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 50$）；第二行为 n 个正整数。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出可以得到的最大整数。

处理器（Processor, Seoul 2008, LA 4254）

有 n 个任务，每个任务有 3 个参数 r_i、d_i 和 w_i，表示必须在时刻$[r_i,d_i]$之内执行，工作量为 w_i。处理器执行的速度可以变化，当执行速度为 s 时，一个工作量为 w_i 的任务需要执行 w_i/s 个单位时间。另外，任务不一定要连续执行，可以分成若干块。你的任务是求出处理器在执行过程中最大速度的最小值。处理器速度可以是任意整数值。

假设有 5 个任务，r_i 和 d_i 分别是[1,4], [3,6], [4,5], [4,7], [5,8]，工作量分别为 2, 3, 2, 2, 1，则图 1-46 是一个最优执行方案，最大速度为 2。

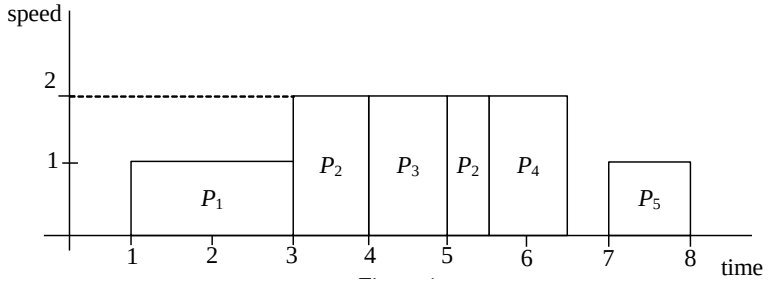


图 1-46

【输入格式】

输入的第一行为数据组数 T（$T \leqslant 20$）。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 10\,000$）；以下 n 行每行包含 3 个整数 r_i, d_i, w_i（$1 \leqslant r_i < d_i \leqslant 20\,000$，$1 \leqslant w_i \leqslant 1\,000$）。

【输出格式】

对于每组数据，输出在执行过程中处理器最大速度的最小值。

障碍滑雪比赛（Slalom, UVa 11627）

在一场滑雪比赛中，你需要通过 n 个旗门（均可看成水平线段）。第 i 个旗门左端的坐标为(x_i,y_i)，所有旗门的宽度均为 W。旗门海拔高度严格递减，即对所有 $1 \leqslant i < n$ 满足 $y_i < y_{i+1}$。你有 S 双滑雪板，第 j 双的速度为 s_j（即向下滑行速度为 s_j 米/秒）。你的水平速度在任何时刻都不能超过 v_h 米/秒，但可以任意变速。如果起点和终点的水平坐标可以任意选择，用哪些滑雪板可以顺利通过所有旗门？

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为 3 个整数 W、v_h 和 N（$1 \leqslant W \leqslant 10^8$，$1 \leqslant v_h \leqslant 10^6$，$1 \leqslant N \leqslant 10^5$）；以下 N 行每行为两个整数 x_i 和 y_i，即每个旗门左端的坐标（$1 \leqslant x_i,y_i \leqslant 10^8$）。下一行包含一个整数 S，即滑雪板的数量（$1 \leqslant S \leqslant 10^6$）；以下 S 行每行一个整数 s_j，即每双滑雪板的速度（$1 \leqslant s_j \leqslant 10^6$）。

【输出格式】

对于每组数据，输出可以通过所有旗门的滑雪板数量。

传说中的车（Fabled Rooks, UVa 11134）

你的任务是在 $n \times n$ 棋盘上放 n 辆车，使得任意两辆车不相互攻击，且第 i 辆车在一个给定的矩形 R_i 之内。

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 5\,000$）；以下 n 行每行用 4 个整数 xl_i, yl_i, xr_i, yr_i（$1 \leqslant xl_i \leqslant xr_i \leqslant n$，$1 \leqslant yl_i \leqslant yr_i \leqslant n$）描述一个矩形，其中$(xl_i,yl_i)$是左上角坐标，$(xr_i,yr_i)$是右下角坐标。换句话说，第 i 个车的位置(x,y)必须满足 $xl_i \leqslant x \leqslant xr_i$ 和 $yl_i \leqslant y \leqslant yr_i$。输入结束标志为 $n=0$。

【输出格式】

对于每组数据，如果无解，输出 IMPOSSIBLE；否则，输出 n 行，依次为第 1，2，…，n 个车的坐标。

旅行 2007（The Trip, 2007, UVa 11100）

给定 n 个正整数，把它们划分成尽量少的严格递增序列（前一个数必须小于后一个数）。比如，6 个正整数 1, 1, 2, 2, 2, 3 至少要分成 3 个序列：{1,2}, {1,2}和{2, 3}。

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 10\,000$），第二行是 n 个不超过 1 000 000 的正整数。输入结束标志为 $n=0$。

【输出格式】

对于每组数据，输出序列个数的最小值 k 和这 k 个序列。如果有多种划分方法，任何一组解均可。

机场（Airport, Seoul 2009, LA 4725）

有一个客流量巨大的机场，却只有一条起飞跑道，如图 1-47 所示。

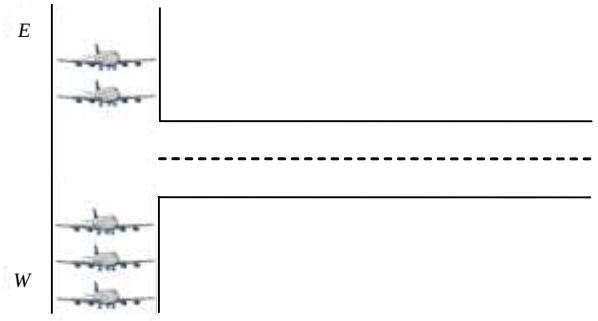


图 1-47

换句话说，每个时刻只能有一架飞机起飞（从 E 或者 W 通道进入起飞跑道），每个时刻也都有一些飞机到达 E 或者 W 通道中。在任意时刻，E 通道和 W 通道里的飞机分别从 0 开始编号（图 1-47 中，E 通道里的飞机编号为 0 和 1，W 通道里的飞机编号为 0,1,2）。你的任务是在每个时刻选择一架飞机起飞，使得任意时刻飞机的最大编号最小。

例如，若飞机到达方式如表 1-4 所示。

表 1-4

时　　刻	W 通道新到达的飞机	E 通道新到达的飞机
1	A_1 A_2 A_3	B_1 B_2
2		B_3 B_4 B_5
3	A_4 A_5	

最优策略是这样的：时刻 1，飞机 A1, A2, A3 编号为 0, 1, 2，飞机 B1, B2 编号为 0, 1，然后让 B1 起飞；时刻 2，飞机 B3, B4, B5 编号为 1, 2, 3，然后让 A1 起飞；时刻 3，A4 和 A5 编号为 2, 3。这样，飞机的最大编号为 3，是所有可能的方案中最小的。

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为时刻总数 n（$1 \leqslant n \leqslant 5\,000$）；以下 n 行每行两个整数 a_i 和 b_i（$0 \leqslant a_i, b_i \leqslant 20$），分别是该时刻到达 W 通道和 E 通道的飞机数量。

【输出格式】

对于每组数据，输出飞机最大编号的最小值。

安装服务（Installations, Daejon 2010, LA 4850）

工程师要安装 n 个服务，其中第 i 个服务 J_i 需要 s_i 单位的安装时间，截止时间为 d_i。如果在截止时间之前完成任务，不会有任何惩罚；否则惩罚值为任务完成时间与截止时间之差。换句话说，如果实际完成时间为 C_i，则惩罚值为 $\max\{0, C_i - d_i\}$。从 $t=0$ 时刻开始执行任务，但同一时刻只能执行一个任务。你的任务是让惩罚值最大的两个任务的惩罚值之和最小。

假定有两个任务，安装时间 s_i 和截止时间 d_i 所组成的二元组 (s_i, d_i) 分别为 (1,7), (4,7), (2,4), (2,15), (3,5), (6,8)。如图 1-48 所示描述了一个最优解，其中惩罚值最小的两个任务分别为 J_2

和 J_6，二者的惩罚值之和为 6+1=7。

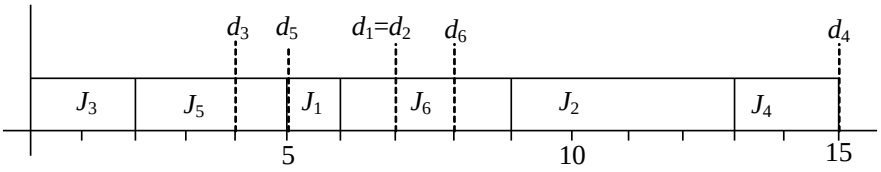


图　1-48

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 500$）；以下 n 行每行两个整数 s_i 和 d_i，即任务 J_i 的安装时间和截止时间（$1 \leqslant s_i \leqslant d_i \leqslant 10\,000$）。

【输出格式】

对于每组数据，输出两个最大惩罚值之和的最小值。

田忌赛马（Tian Ji - The Horse Racing, Shanghai 2004, LA 3266）

田忌与齐王赛马，两人各出 N 匹马。赢一场比赛得 200 两银子，输了赔 200 银子，平局不赔不赚。已知两人每匹马的速度，问田忌至多能赢多少两银子。

【输入格式】

输入包含最多 50 组数据。每组数据的第一行为一个整数 n（$n \leqslant 1\,000$）；第二行包含 n 个正整数，即田忌每匹马的速度；第三行包含 n 个正整数，即齐王每匹马的速度。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出田忌最多能赢多少两银子。

巴士司机问题（The Bus Driver Problem, UVa 11389）

有 n 个司机、n 个下午路线和 n 个夜间路线。给每个司机安排一个下午路线和夜间路线，使得每条路线恰好被分配到一个司机，且需要支付给司机的总加班费尽量少。如果一个司机的行驶总时间（下午路线的时间与夜间路线的时间之和）不超过 d，则没有加班费；超出的部分每小时需要支付 r 元的加班费。

【输入格式】

输入包含多组数据。每组数据的第一行包含 3 个整数 n, d, r（$1 \leqslant n \leqslant 100$，$1 \leqslant d \leqslant 10\,000$，$1 \leqslant r \leqslant 5$）；第二行包含 n 个整数，即各条下午路线的行驶时间；第三行包含 n 个整数，即各夜间路线的行驶时间。所有行驶时间均为不超过 10 000 的正整数。输入结束标志为 n=d=r=0。

【输出格式】

对于每组数据，输出最小总加班费。

梦之队（Wonder Team, Tehran 2007, LA 4094）

有 n（$1 \leqslant n \leqslant 50$）支队伍比赛，每两支队伍打两场（主客场各一次），胜得 3 分，平得 1 分，输不得分。比赛结束之后会评选出一个梦之队（也可能空缺），它满足如下条件：进球总数最多（不能并列），胜利场数最多（不能并列），丢球总数最少（不能并列）。

求梦之队的最低可能排名。一支得分为 p 的球队的排名等于得分严格大于 p 的球队个数加 1。

积木艺术（Cubist Artwork, Tokyo 2009, LA 4636）

用一些等大的立方体搭积木，每个立方体或者直接放在地面的网格上，或者放在另一个立方体的上面，给出正视图和侧视图，如图 1-49 所示。你的任务是判断最少要用多少个立方体。

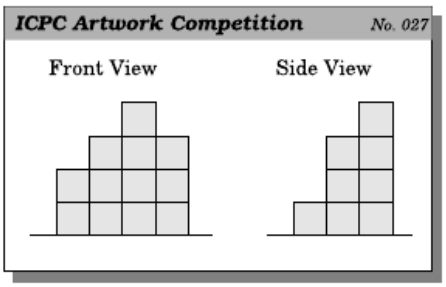


图　1-49

图 1-50 是两种可能的方案，其中图 1-50（b）所示的是最优方案（立方体数目最少）。

（a）　　（b）

图　1-50

最后是一些需要暴力求解的题目，见表 1-5。其中有些题目难度较大，请读者根据实际情况选择适合自己的题目完成，不必勉强。

表　1-5

题　号	统　计	题目名称（英文）	备　注
LA4253	11/0.00%	Archery	枚举。注意特殊情况和精度
LA3667	92/68.48%	Ruler	搜索
LA5693	1/100.00%	Compress the String	搜索。需要优化
LA5704	7/85.71%	Yummy Triangular Pizza	回溯法。可参考 https://oeis.org/A006534
UVa10825	176/80.68%	Anagram and Multiplication	枚举（需要猜想）
UVa10639	32/46.88%	Square Puzzle	回溯法。注意细节
LA3403	63/79.37%	Mobile Computing	枚举二叉树
LA3406	41/53.66%	Bingo	注意枚举方式
LA3621	92/82.61%	Power Calculus	经典的搜索题目；注意优化
LA2108	2/0.00%	Houses Divided	搜索；需要优化
LA5842	13/76.92%	Equipment	需认真分析题目

LA5844	16/75.00%	Leet	以“火星文”为背景的题目
LA4644	4/75.00%	Hobby on Rails	比较繁琐的搜索题目

箭术（Archery, Seoul 2008, LA 4253）

有 n 个平行于 x 轴的线段，每条线段代表一个靶子。你的任务是判断是否可以站在 x 轴上[0,W]区间内的某个位置射箭，使得箭能穿过所有靶子。假设箭沿直线飞行，直到无穷远处。不同靶子的高度（y 坐标）不同。

如图 1-51 所示，站在 B 点可以射穿所有靶子，但站在 A 点不行。

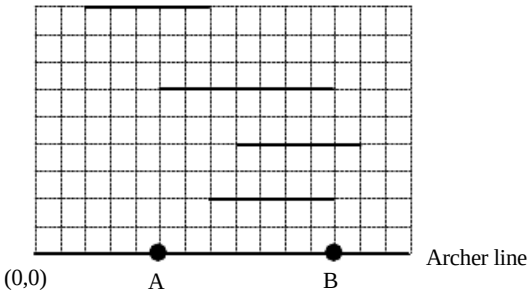


图　1-51

【输入格式】

输入的第一行为数据组数 T（T≤30）。每组数据的第一行为整数 W（2≤W≤10 000 000）；第二行为整数 n（2≤n≤5 000），即靶子的个数，以下 n 行每行 3 个整数 D_i, L_i, R_i（1≤D_i≤W，0≤L_i<R_i≤W），表示有一个高度为 D_i，左右端点的 x 坐标分别为 L_i 和 R_i 的靶子。不同靶子的高度 D_i 保证不同。

【输出格式】

对于每组数据，如果可以射穿所有靶子，输出 YES，否则输出 NO。

刻度尺（Ruler, Beijing 2006, LA 3667）

给出 n 种距离 d_i，设计一个有 m 个刻度的尺子，使得每个 d_i 都可以直接量出来（即存在某两个刻度之间的距离恰好为 d_i）。要求在 m 尽量小的前提下保证尺子的总长度尽量短。

【输入格式】

输入包含多组数据。每组数据包含两行，第一行为一个整数 n（1≤n≤50），第二行包含 n 个整数 d_i（1≤d_i≤10^6）。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出 m 和这 m 个刻度（从小到大排列，第一个数必须为 0）。输入保证 m≤7。

美味的三角匹萨（Yummy Triangular Pizza, Shanghai 2011, LA5704）

用 n（1≤n≤16）个等大的等边三角形，可以组成多少个形状不同的匹萨？匹萨必须是连通的，中间可以有洞。平移或旋转（不能翻转）之后能重合的只算一种。例如，n=4 时有 4 种组合方案，如图 1-52 所示。n=10 时有 866 种组合方案。

图　1-52

神奇的乘法（Anagram and Multiplication, UVa 10825）

有些 m 位 n 进制整数非常神奇：乘以 2, 3,…, m 之后，所得到的数恰好是原数各数字的一个排列。比如，142857 就是这样一个十进制 6 位整数。

2×142 857 = 285 714

3×142 857 = 428 571

4×142 857 = 571 428

5×142 857 = 714 285

6×142 857 = 857 142

输入 m、n，你的任务是找到这样一个整数。输入保证这样的整数最多只有一个。

【输入格式】

输入包含若干组数据。每组数据包含两个整数 m 和 n（$3 \leqslant m \leqslant 6$，$4 \leqslant n \leqslant 400$）。输入结束标志为 $m=n=0$。

【输出格式】

对于每组数据，输出这个整数。每位数字是 0~n−1 之间的整数，相邻两位数字之间用一个空格隔开。如果无解，输出“Not found.”。

正方形拼图（Square Puzzle, UVa 10639）

给出一些拼块，如图 1-53 所示。你的任务是判断它们是否可以拼成一个给定大小的正方形。拼块可以旋转，但不能翻转，所有拼块都必须使用，且拼块不能重叠。

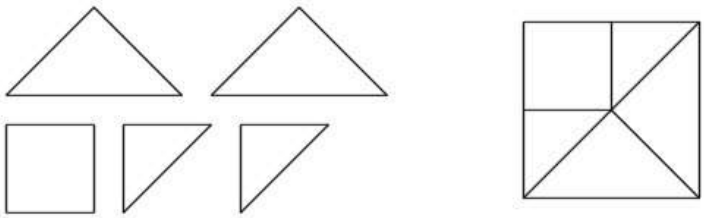


图　1-53

每个拼块用多边形表示。所有顶点坐标都是整数，所有边都是横、竖或者斜向 45°。

【输入格式】

输入的第一行为数据组数 T（$T \leqslant 20$）。每组数据的第一行为两个整数 n 和 m（$1 \leqslant n$，$m \leqslant 6$），其中 n 为拼块的个数，m 为正方形边长。以下 n 行每行描述一个拼块，其中第一行为整数 k，即多边形顶点数；接下来为 k 对整数，按照逆时针顺序给出各顶点的坐标。所有顶点坐标均为 0~m 的整数。

【输出格式】

对于每组数据，根据情况输出 yes 或者 no。

天平难题（Mobile Computing, Tokyo 2005, LA 3403）

给出房间的宽度 r 和 s 个挂坠的重量 w_i。设计一个尽量宽，但宽度不能超过房间宽度 r 的天平，挂着所有挂坠。

每个天平的每一端要么挂一个挂坠，要么挂另外一个天平。如图 1-54 所示，设 n 和 m 分别是两端挂的总重量，要让天平平衡，必须满足 $n \times a = m \times b$。

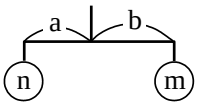


图　1-54

例如，如果有 3 个重量分别为 1, 1, 2 的挂坠，有如图 1-55 所示 3 种平衡方案。

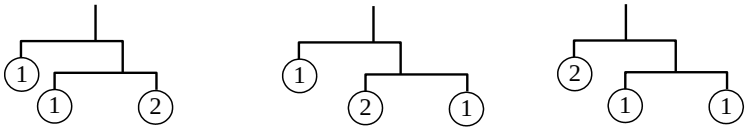


图　1-55

挂坠的宽度忽略不计，且不同的子天平可以相互重叠。如图 1-56 所示，宽度为(1/3)+1+(1/4)。

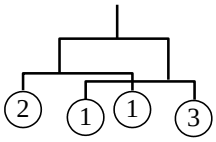


图　1-56

房间宽度 r 和挂坠数目 s 满足 $0 < r < 10$ 和 $1 \leqslant s \leqslant 6$，且每个挂坠的重量 w_i 满足 $1 \leqslant w_i \leqslant 1\,000$。输入保证不存在天平的宽度恰好在 $r-10^{-5}$ 和 $r+10^{-5}$ 之间。

Bingo 游戏（Bingo, Tokyo 2005, LA 3406）

有 P 个 $M \times M$ 的数字矩阵（$2 \leqslant P \leqslant 4$，$3 \leqslant M \leqslant 4$）。你可以依次选择一些整数，每选一个整数后就把所有矩阵中的同一数字全部圈起来。如果某个矩阵中某一行、某一列或者某条主对角线（一共有两条）上的 M 个数字都被圈起来了，我们就说这个矩阵 Bingo 了。你的任务是找到一个最短的数字序列，使得最后所有的矩阵按照从左到右的顺序依次 Bingo（相邻矩阵可以同时 Bingo）。

图 1-57 是一个成功的序列。注意，5 不能在 16 之前选择，否则第 4 列会在第 2、3 列之前 Bingo，违反了题目规定。

图 1-57

快速幂计算（Power Calculation, Yokohama 2006, LA 3621）

给出 x 和正整数 n（$1 \leqslant n \leqslant 1000$），问最少需要几次乘除法可以得到 x^n。比如，x^{31} 需要 6 次乘除计算：$x^2=x\times x$，$x^4=x^2\times x^2$，$x^8=x^4\times x^4$，$x^{16}=x^8\times x^8$，$x^{32}=x^{16}\times x^{16}$，$x^{31}=x^{32}/x$。计算过程中 x 的指数应当总是正整数（比如 $x^{-3}=x/x^4$ 是不允许的）。

2．高效算法设计

还有一些题目并不需要巧妙的思路和缜密的推理就能找到一个解决方案，只是时间效率难以令人满意。降低时间复杂度的方法有很多，本章的例题就覆盖了其中最常见的一些，如表 1-6 所示。

表 1-6

类　别	题　号	统　计	题目名称（英文）	备　注
例题 17	UVa11462	3193/87.97%	Age Sort	排序后用贪心法
例题 18	UVa11078	673/89.15%	Open Credit System	扫描、维护最大值
例题 19	UVa11549	344/84.59%	Calculator Conundrum	Floyd 判圈算法
例题 20	LA3905	106/71.70%	Meteor	线性扫描；事件点处理
例题 21	LA2678	351/83.19%	Subsequence	线性扫描；前缀和；单调性
例题 22	LA3029	197/81.22%	City Game	递推；扫描法
例题 23	LA3695	48/85.42%	Distant Galaxy	枚举；线性扫描
例题 24	UVa10755	218/77.52%	Garbage Heap	前缀和、降维、递推
例题 25	LA2965	18/83.33%	Jurassic Remains	中途相遇法

接下来仍然列举一些习题，以供读者练习和提高，如表 1-7 所示。

表　1-7

题　　号	统　　计	题目名称（英文）	备　　注
LA2963	16/81.25%	Hypertransmission	扫描；维护信息
UVa10827	799/78.97%	Maximum sum on a torus	前缀和；降维
UVa10125	2151/71.97%	Sumsets	中途相遇法
UVa10763	1494/85.54%	Foreign Exchange	快速检索
UVa10391	1061/79.55%	Compound Words	字符串检索；哈希表
UVa11054	865/94.57%	Wine trading in Gergovia	扫描法
LA4726	36/41.67%	Average	树形结合或者单调队列
LA4851	36/91.67%	Restaurant	（请读者独立思考）
LA4950	107/80.37%	Selling Land	（请读者独立思考）
LA4356	13/61.54%	Fire-Control System	扫描法
LA2689	69/73.91%	Cricket Field	在 $W\times H$ 网格里找一个最大空正方形
LA5052	85/84.71%	Genome Evolution	（请读者独立思考）
LA3716	20/85.00%	DNA Regions	利用数学变形或者数形结合
LA4629	24/54.17%	Knowledge for the masses	（请读者独立思考）
LA4621	117/79.49%	Cav	（请读者独立思考）
LA3693	47/97.87%	Balancing the Scale	（请读者独立思考）
LA4294	158/86.08%	Shuffle	有多种方法
LA5848	14/85.71%	Soju	（请读者独立思考）
LA4062	65/76.92%	You are around me ...	（请读者独立思考）

超级传输（Hypertransmission, NEERC 2003, LA 2963）

需要在 n 个星球上各装一个广播装置，作用范围均为 R（即和它距离不超过 R 的星球能收听到它的广播）。每个星球广播 A 类节目或者 B 类节目。令 $N^+(i)$表示星球 i 收听到的和自己广播相同节目的星球数（包括星球 i 自己），$N^-(i)$表示星球 i 收听到的广播另一种节目的星球数。如果 $N^+(i)<N^-(i)$，我们说星球 i 是不稳定的。你是暗黑世界的间谍，因此希望选择 R，使得不稳定的星球尽量多些。在此前提下，R 应尽量小。

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n（$1\leqslant n\leqslant 1\,000$），即星球个数；以下 n 行每行包含 4 个整数 x_i, y_i, z_i 和 p_i，其中，(x_i,y_i,z_i)是星球 i 的空间位置，$p_i=0$ 表示星球 i 广播 A 类节目，$p_i=1$ 表示星球 i 广播 B 类节目。所有坐标的绝对值不超过 10 000。不同星球的位置保证不同。输入结束标志为文件结束符（EOF）。

【输出格式】

对于每组数据，输出两行。第一行为不稳定的星球个数。第二行为让不稳定星球数最大化的最小的 R 值，精确到 10^{-4}。

环面上的最大和（Maximum sum on a torus, UVa 10827）

把一个网格的第一行和最后一行粘起来，第一列和最后一列粘起来，可以得到一个环面。给定一个整数网格，求出所对应环面上的最大子矩形（该子矩形的所有元素之和最大）。

如图 1-58 所示就是一个最大的子矩形。

1	−1	0	0	-4
2	3	−2	−3	2
4	1	−1	5	0
3	−2	1	−3	2
−3	2	4	1	-4

图 1-58

【输入格式】

输入的第一行为数据组数 T（$T \leqslant 18$）。每组数据的第一行为网格的行数和列数（它总是一个正方形）n（$1 \leqslant n \leqslant 75$）；以下 n 行每行 n 个−100~100 之间的整数。

【输出格式】

对于每组数据，输出一行，即最大子矩形内元素的和。

和集（Sumsets, UVa 10125）

给定一个整数集合 S，找出一个最大的 d，使得 $a+b+c=d$，其中 a, b, c, d 是 S 中的不同元素。

【输入格式】

输入包含多组数据。每组数据的第一行为集合内的元素个数 n（$1 \leqslant n \leqslant 1\,000$），以下 n 行每行一个−53 6870 912~+536 870 911 的整数。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出最大的 d。如果无解，输出“no solution”。

交换学生（Foreign Exchange, UVa 10763）

有 n 个学生想交换到其他学校学习。为了简单起见，规定每个想从 A 学校换到 B 学校的学生必须找一个想从 B 学校换到 A 学校的“搭档”。如果每个人都能找到搭档（一个人不能当多个人的搭档），学校就会同意他们交换。你的任务是判断交换是否可以进行。

【输入格式】

输入包含多组数据。每组数据的第一行为学生个数 n（$1 \leqslant n \leqslant 500\,000$）；以下每行包含两个不同的非负整数 A 和 B，表示该学生想从 A 学校换到 B 学校。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出 YES 或者 NO。

复合词（Compound Words, UVa 10391）

给定一个词典，要求找出其中所有的复合词，即恰好由两个单词连接而成的单词。

【输入格式】

输入只有一组数据，其中每行都是一个由小写字母组成的单词。输入已按照字典序排序，且不超过 120 000 个单词。

【输出格式】

输出所有复合词，按照字典序排列。

Gergovia 的酒交易（Wine trading in Gergovia, UVa 11054）

直线上有 n 个等距的村庄，每个村庄要么买酒，要么卖酒。把 k 个单位的酒从一个村庄运到相邻村庄需要 k 个单位的劳动力。问最少需要多少劳动力才能满足所有村庄的需求。

【输入格式】

输入包含多组数据。每组数据的第一行为村庄个数 n（$2 \leqslant n \leqslant 100\,000$）；第二行从左到右给出各个村庄对酒的需求 a_i（$-1\,000 \leqslant a_i \leqslant 1\,000$），其中 $a_i>0$ 表示买酒，$a_i<0$ 表示卖酒。输入保证所有 a_i 之和等于 0。输入结束标志为 $n=0$。

【输出格式】

输出劳动力总和的最小值。输出保证在 64 位带符号整数的范围内。

平均值（Average, Seoul 2009, LA 4726）

给定一个长度为 n 的 01 序列，选一个长度至少为 L 的连续子序列，使得子序列中数字的平均值最大。如果有多解，子序列长度应尽量小；如果仍有多解，起点编号应尽量小。序列中的字符编号为 1~n，因此[1,n]就是指完整的序列。

例如，对于长度为 17 的序列 00101011011011010，如果 $L=7$，子序列[7,14]的平均值最大，为 6/8（它的长度为 8）；如果 $L=5$，子序列[7,11]的平均值最大，为 4/5。

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为两个整数 n 和 L（$1 \leqslant n \leqslant 100\,000$，$1 \leqslant L \leqslant 1\,000$），第二行为一个长度为 n 的 01 序列。

【输出格式】

对于每组数据，输出满足条件的最优子序列的起点和终点编号。

餐厅（Restaurant, Daejon 2010, LA 4851）

有一个 $M \times M$ 的网格，左下角坐标为(0,0)，右上角坐标为($M-1$,$M-1$)。网格里有两个 y 坐标相同的宾馆 A 和 B，以及 n 个餐厅。宾馆 A 和宾馆 B 里各有一个餐厅，编号为 1 和 2，其他地方的餐厅编号为 3~n。现在你打算开一家新餐厅，需要考查一下可能的位置。

一个位置 p 是“好位置”的条件为：当且仅当对于已有的每个餐厅 q，要么 p 比 q 离 A 近，要么 p 比 q 离 B 近，即 dist(p,A)<dist(q,A)或者 dist(p,B)<dist(q,B)。如图 1-59 所示，A 和 B 的坐标分别为(0,5)和(10,5)，(7,4)是个好位置，但(4,6)不是好位置，因为位于(3,5)处的餐厅不管是到宾馆 A 还是到宾馆 B，都比(4,6)要近。

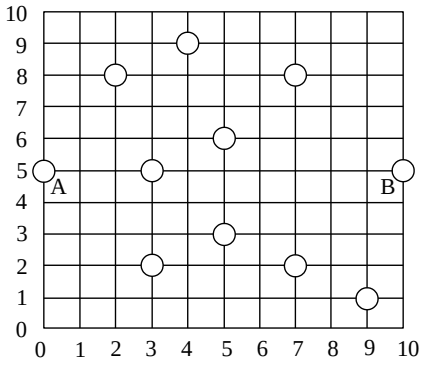


图　1-59

你的任务是统计网格中好位置的个数。

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为两个整数 M 和 n（$2 \leqslant M \leqslant 60\,000$，$2 \leqslant n \leqslant 50\,000$）；以下 n 行每行包含两个数 x 和 y（$0 \leqslant x,y < M$），即各个餐厅的坐标。不同餐厅的坐标保证不同，宾馆 A 里的餐厅编号为 1，宾馆 B 里的餐厅编号为 2。餐厅 1 和餐厅 2 的 y 坐标保证相同。

【输出格式】

对于每组数据，输出好位置的个数。

火势控制系统（Fire-Control System, Hangzhou 2008, LA 4356）

在平面上有 n 个目标点，你的任务是找出一个圆心在(0,0)点处的扇形，至少覆盖其中的 k 个点，使得该扇形的面积最小。

【输入格式】

输入包含多组数据。每组数据的第一行为两个整数 n 和 k（$1 \leqslant n \leqslant 5\,000$，$k \leqslant n$）；以下 n 行每行包含两个整数 x 和 y，即每个目标点的坐标。坐标均为绝对值不超过 1 000 的整数，且没有目标点在(0,0)。输入结束标志为 $n=k=0$。

【输出格式】

对于每组数据，输出覆盖至少 k 个点的最小扇形的面积，保留两位小数。

基因组进化（Genome Evolution, Tehran 2010, LA 5052）

给出 1~n 的两个排列 A 和 B，统计有多少个二元组(A', B')满足以下条件：A'是 A 的连续子序列，B'是 B 的连续子序列，且 A'和 B'包含的整数集完全相同。A'和 B'均应至少包含两个元素。

例如，A={3,2,1,4}，B={1,2,4,3}时，有 3 组解：{2,1}，{1,2}；{2,1,4},{1,2,4}；{3,2,1,4},{1,2,4,3}。

【输入格式】

输入包含多组数据。每组数据的第一行为 n（$2 \leqslant n \leqslant 3\,000$）；第二行和第三行各包含一个 1~$n$ 的排列，分别为 A 和 B。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出满足条件的二元组的个数。

DNA 突变区域（DNA Regions, CERC 2006, LA 3716）

给出两条长度均为 n 的 DNA 链（字符串）A 和 B，你的任务是找出一段最长的区域，使得该区域内的突变位置不超过 p% 。换句话说，你需要找出一个尽量长的闭区间$[L, R]$，使得对于区间内的所有位置 x（$L \leqslant x \leqslant R$），有不超过 p%的 x 满足 $A_x \neq B_x$。

【输入格式】

输入包含多组数据。每组数据的第一行为两个整数 n 和 p（$1 \leqslant n \leqslant 150\,000$，$1 \leqslant p \leqslant 99$）；以下两行各为一条长度为 n 的 DNA 链，只由大写字母 A、C、G、T 组成。输入结束标志为 n=0。

【输出格式】

输出满足条件的区域长度的最大值。如果不存在，输出“No solution.”（不含引号）。

3．动态规划

动态规划是几乎所有算法竞赛的宠儿。理由很简单：动态规划对思维的要求比较高，常用来解决那些其他算法都不奏效的题目。本章的动态规划例题并不多，但却包含了不少重要的思想和方法，如表 1-8 所示。

表　1-8

类　别	题　号	统　计	题目名称（英文）	备　注
例题 26	LA3882	241/85.48%	And Then There Was One	递归、问题转化
例题 27	UVa10635	767/76.79%	Prince and Princess	LCS；可转化为 LIS
例题 28	UVa10891	936/88.78%	Game of Sum	避免重复计算
例题 29	UVa11825	122/50.00%	Hacker’s Crackdown	集合动态规划；子集枚举
例题 30	UVa10859	147/62.59%	Placing Lampposts	树上的动态规划
例题 31	LA3983	205/81.46%	Robotruck	动态规划；滑动窗口优化；单调队列
例题 32	LA4794	299/74.25%	Sharing Chocolate	集合动态规划、状态精简

下面仍然列举一些习题。数量虽不少，但却并不是前面例题的简单重复和改头换面。其中有些题目可以直接转化为经典题目，或者顺着经典问题的思路即可解决，但也有一些题目需要认真分析才能解决。动态规划题目对思维训练非常有帮助，请读者予以重视。

首先是相对简单的题目，如表 1-9 所示。

表　1-9

题　号	统　计	题目名称（英文）	备　注
UVa11584	266/89.10%	Partitioning by Palindromes	入门题目
LA4256	16/87.50%	Salesman	入门题目
UVa10534	1404/79.63%	Wavio Sequence	可以转化为经典问题，时间 O(nlogn)
UVa11552	161/83.85%	Fewest Flops	序列划分模型；状态设计
UVa11404	298/59.73%	Palindromic Subsequence	可以转化为 LCS
LA4731	17/88.24%	Cellular Network	需要一点概率知识和推理
UVa11795	130/87.69%	Mega Man’s Mission	基础的集合动态规划
LA4727	40/57.50%	Jump	Joseph 问题的变形
LA3530	104/92.31%	Martian Mining	模型简单，但需要减少重复计算
UVa10564	521/79.08%	Paths through the Hourglass	类似 01 背包问题
UVa10817	334/81.14%	Headmaster’s Headache	集合动态规划
LA2038	150/78.00%	Strategic game	树上动态规划（基础题）
LA3363	3/100.00%	String Compression	字符串动态规划
LA2031	35/65.71%	Dance Dance Revolution	以跳舞机为背景的题目
LA4643	108/67.59%	Twenty Questions	有趣的问题；比较基础的动态规划

续表

划分成回文串（Partitioning by Palindromes, UVa 11584）

输入一个由小写字母组成的字符串，你的任务是把它划分成尽量少的回文串。比如，

racecar 本身就是回文串；fastcar 只能分成 7 个单字母的回文串；aaadbccb 最少可分成 3 个回文串：aaa、d、bccb。字符串长度不超过 1 000。

商人（Salesman, Seoul 2008, LA 4256）

给定一个包含 n 个点（$n \leqslant 100$）的无向连通图和一个长度为 L 的序列 A（$L \leqslant 200$），你的任务是修改尽量少的数，使得序列中的任意两个相邻数或者相同，或者对应图中两个相邻结点。

波浪子序列（Wavio Sequence, UVa 10534）

给定一个长度为 n 的整数序列，求一个最长子序列（不一定连续），使得该序列的长度为奇数 $2k+1$，前 $k+1$ 个数严格递增，后 $k+1$ 个数严格递减。注意，严格递增/递减意味着该序列中的两个相邻数不能相同。$n \leqslant 10\,000$。

最小的块数（Fewest Flops, UVa 11552）

输入一个正整数 k 和字符串 S，字符串的长度保证为 k 的倍数。把 S 的字符按照从左到右的顺序每 k 个分成一组，每组之间可以任意重排，但组与组之间的先后顺序应保持不变。你的任务是让重排后的字符串包含尽量少的“块”，其中每个块为连续的相同字母。比如，uuvuwwuv 可分成两组：uuvu 和 wwuv，第一组可重排为 uuuv，第二组可重排为 vuww，连起来是 uuuvvuww，包含 4 个“块”。

【输入格式】

输入的第一行包含一个整数 t（$t \leqslant 100$），即测试数据的数量；以下 t 行每行包含整数 k 和字符串 S。串长保证是 k 的整数倍。S 由不超过 1 000 个小写字母组成。

【输出格式】

对于每组数据，输出重排后的 S 所包含的最小“块”数。

回文子序列（Palindromic Subsequence, UVa 11404）

给定一个由小写字母组成的字符串，删除其中的 0 个或多个字符，使得剩下的字母（顺序不变）组成一个尽量长的回文串。如果有多解，输出字典序最小的解。

【输入格式】

输入包含多组测试数据。每组数据仅一行，为一个长度不超过 1 000 的非空字符串。

【输出格式】

对于每组数据，输出所求的最长回文串。

蜂窝网络（Cellular Network, Seoul 2009, LA 4731）

手机在蜂窝网络中的定位是一个基本问题。假设蜂窝网络已经得知手机处于 $c_1, c_2, \ldots, c_n$ 这些区域中的一个，最简单的方法是同时在这些区域中寻找手机。但这样做很浪费带宽。由于蜂窝网络中可以得知手机在这不同区域中的概率，因此一个折中的方法就是把这些区域分成 w 组，然后依次访问。比如，已知手机可能位于 5 个区域中，概率分别为 0.3、0.05、0.1、0.3 和 0.25，w=2，则一种方法是先同时访问$\{c_1,c_2,c_3\}$，再同时访问$\{c_4,c_5\}$，访问区域数的数学期望为 3*(0.3+0.05+0.1)+(3+2)*(0.3+0.25)=4.1。另一种方法是先同时访问$\{c_1,c_4\}$，再访问$\{c_2,c_3,c_5\}$，访问区域数的数学期望为 2×(0.3+0.3)+(3+2)×(0.05+0.1+0.25)=3.2。

【输入格式】

输入的第一行为数据组数 T。每组数据的第一行为两个整数 n 和 w（$1 \leqslant w \leqslant n \leqslant 100$）；

第二行为包含 n 个不超过 10 000 的正整数 $u_1, u_2, \ldots, u_n$。手机在 c_i 的概率为 $p_i=u_i/(u_1+u_2+\ldots+u_n)$。

【输出格式】

对于每组数据，输出访问区域数的数学期望的最小值。

洛克人的难题（Mega Man's Missions, UVa 11795）

洛克人最初只有一个武器“Mega Buster”。你需要按照一定的顺序消灭 n 个其他机器人。每消灭一个机器人将会得到他的武器，而某些机器人只能用特定的武器才能消灭。你的任务是计算出可以消灭所有机器人的顺序总数。

【输入格式】

输入的第一行为数据组数 T（T≤50）。每组数据的第一行为机器人个数 n（1≤n≤16）；以下 n+1 行描述各种武器，其中第 1 行描述 Mega Buster，第 k+1 行描述消灭 k 号机器人后获得的武器。其中每行包含 n 个整数，分别表示该武器是否能消灭第 1、2、3、…、n 号机器人（1 表示可以，0 表示不可以）。注意，一个机器人的武器可能可以消灭自己，但不会影响到本题的答案，因为必须先消灭这个机器人，才能拿到武器。

【输出格式】

对于每组数据，输出可以消灭所有机器人的顺序总数。

跳跃(Jump, Seoul 2009, LA 4727)

把 1~n 按逆时针顺序排成一个圆圈，从 1 开始每 k 个数字删掉一个，直到所有数字都被删除。这些数的删除顺序记为 Jump(n,k)（n,k≥1）。

例如，Jump(10, 2) = [2,4,6,8,10,3,7,1,9,5]，Jump(13, 3) = [3,6,9,12,2,7,11,4,10,5,1,8,13]，Jump(13, 10) = [10,7,5,4,6,9,13,8,3,12,1,11,2]，Jump(10, 19) = [9,10,3,8,1,6,4,5,7,2]。

你的任务是求出 Jump(n,k)的最后 3 个数。

【输入格式】

输入的第一行为数据组数 T。每组数据仅一行，为两个整数 n 和 k (5≤n≤500 000，2≤k≤500 000)

【输出格式】

对于每组数据，按顺序输出 Jump(n,k)的最后 3 个元素。

火星采矿（Martian Mining, LA 3530）

给出 n×m 网格中每个格子的 A 矿和 B 矿的数量，A 矿必须由右向左运输，B 矿必须由下向上运输，如图 1-60 所示。管子不能拐弯或者间断。要求收集到的 A、B 矿总量尽量大。

【输入格式】

输入包含多组数据。每组数据的第一行为两个整数 n 和 m（1≤n,m≤500），即行数和列数；以下 n 行每行 m 个整数，即每个格子中的 A 矿数量；再以下 n 行每行 m 个整数，即每个格子中的 B 矿数量。输入结束标志为 n=m=0。

【输出格式】

对于每组数据，输出收集到的矿总量的最大值。

沙漏里的路径（Paths through the Hourglass, UVa 10564）

有一个沙漏，第一行有 n 个格子，第二行 n–1 个格子……最中间的行只有 1 个格子，然后它下面一行 2 个格子，再下面一行 3 个格子……最后一行 n 个格子，如图 1-61 所示。

你可以从第一行开始往下走，每次往下走一行，往左或往右走一列，但不能走出沙漏。你的目标是让沿途经过的所有整数之和恰好为一个给定整数 S。求出符合上述条件的路径条数和一条路径。

如果有多条路径，起点编号应尽量小（第一行格子从左到右编号为 0~n−1）。如果仍有多解，移动序列（L 代表左，R 代表右）的字典序应最小。

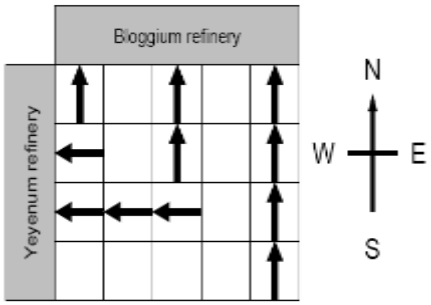


图 1-60

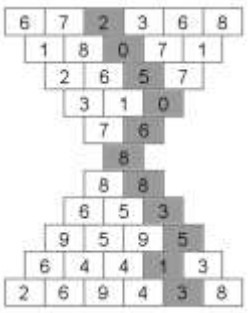


图 1-61

【输入格式】

输入包含多组数据。每组数据的第一行为两个整数 n 和 S（2≤n≤20，0≤S<500）。以下 2n−1 行为各个格子里的整数（均为 0~9 的整数）。

【输出格式】

对于每组数据，输出路径条数和路径（如果有解的话）。

校长的烦恼（Headmaster's Headache, UVa 10817）

某校有 n 个教师和 m 个求职者。已知每人的工资和能教的课程集合，要求支付最少的工资使得每门课都至少有两名教师教学。在职教师必须招聘。

【输入格式】

输入包含多组数据。每组数据的第一行为 3 个整数 s、m 和 n（1≤s≤8，1≤m≤20，1≤n≤100），即科目的个数、在职教师个数和申请者个数；以下 m 行每行用一些整数描述一位在职教师，其中第一个整数 c（10 000≤c≤50 000）是工资，接下来的若干整数是他能教的科目列表（课程编号为 1~s 之间的整数）；接下来的 n 行描述申请者，格式同上。输入结束标志为 s=0。

【输出格式】

对于每组数据，输出工资总额的最小值。

战略游戏（Strategic Game, SEERC 2000, LA 2038）

给定一棵树，选择尽量少的结点，使得每个没有选中的结点至少和一个已选结点相邻。

【输入格式】

输入包含多组数据。每组数据的第一行为结点数 n（n≤1 500）；以下 n 行每行描述一个结点的相邻点列表。结点编号为 0~n−1，每条边恰好在输入中出现一次。输入结束标志为 n=0。

【输出格式】

对于每组数据，输出最少需要选的结点数。

接下来的题目有一定难度，如表 1-10 所示。

表　1-10

题　　号	统　　计	题目名称（英文）	备　　注
LA4394	12/83.33%	String Painter	序列的动态规划，有一定难度
LA4327	97/80.41%	Parade	模型不难想，但需要优化
LA4945	127/79.53%	Free Goodies	也可以贪心，时间效率更高
LA4015	2/50.00%	Caves	树的动态规划
LA4490	92/85.87%	Help Bubu	（请读者独立思考）
UVa11600	78/89.74%	Masud Rana	注意状态表示
LA4987	43/0.00%	Evacuation Plan	（请读者独立思考）
LA4593	2/50.00%	Exclusive Access 2	（请读者独立思考）
LA4048	23/21.74%	Fund Management	注意状态表示
LA4625	57/54.39%	Garlands	（请读者独立思考）
LA4613	104/78.85%	Mountain Road	（请读者独立思考）
LA4614	165/79.39%	Moving to Nuremberg	（请读者独立思考）
LA4050	88/82.95%	Hanoi Towers	（请读者独立思考）
LA3305	118/79.66%	Tour	经典问题
LA3683	30/50.00%	A Scheduling Problem	树的动态规划
LA3637	84/73.81%	The Bookcase	不太容易想到，且需要优化
LA3412	72/62.50%	Pesky Heroes	树的动态规划（题目不太好理解）
LA5717	28/60.71%	Peach Blossom Spring	一类经典题目（最早出现在 NWERC 2006，但本题数据更强）
LA3679	45/64.44%	Pitcher Rotation	需要一点优化（精简状态）

续表

题　　号	统　　计	题目名称（英文）	备　　注
LA3605	4/0.00%	Roommate	（请读者独立思考）
LA3608	1/0.00%	Period	（请读者独立思考）
LA3610	1/0.00%	Log Jumping	可以转化为经典问题
LA3623	40/50.00%	The Best Name for Your Baby	有难度的动态规划；注意计算顺序
LA4002	47/36.17%	The Ultimate Password	有难度的动态规划；注意计算顺序
LA2178	5/40.00%	The Minimum Number of Rooks	有难度的动态规划
LA2221	31/61.29%	Frontier	涉及几何（见第 4 章）的动态规划
LA2923	7/42.86%	Bundling	（请读者独立思考）
LA2930	64/70.31%	Minimizing Maximizer	01 原则；数据结构优化动态规划
LA3132	84/73.81%	Minimax Triangulation	（请读者独立思考）
LA3181	41/51.22%	Fixing the Great Wall	（请读者独立思考）
LA3710	84/80.95%	Interconnect	注意状态表示
LA4290	85/45.88%	Easy Climb	需要优化
LA5088	33/93.94%	Alice and Bob’s Trip	树上的动态规划
UVa10559	481/58.00%	Blocks	重点是设计状态及其转移

LA3782	178/65.73%	Bigger is Better	有多种方法。可以不用高精度
LA4031	97/94.85%	Integer Transmission	需要认真思考。可以做到 $O(n^2)$时间
UVa11521	33/45.45%	Compressor	需要认真思考。很容易写错

字符串“刷子”（String Painter, Chengdu 2008, LA 4394）

给定两个长度相等，只有小写字母组成的字符串 *s* 和 *t*，每步可以把 *s* 的一个连续子串“刷”成同一个字母，问至少需要多少步才能把 *s* 变成 *t*。比如，*s*=bbbbbbb，*t*=aaabccb，最少需要两步可实现将 *s* 变成 *t*：bbbbbbb→aaabbbb→aaabccb。

【输入格式】

输入包含多组数据。每组数据包含两行，第一行为 *s*，第二行为 *t*。两个字符串的长度保证相等，且不超过 100。

【输出格式】

对于每组数据，输出一个整数，即最少步数。

游行（Parade, Beijing 2008, LA 4327）

F 城由 *n*+1 个横向路和 *m*+1 个竖向路组成。你的任务是从最南边的路走到最北边的路，使得走过的路上的高兴值和最大（注意，一段路上的高兴值可以是负数）。同一段路不能经过两次，且不能从北往南走，如图 1-62 所示。另外，在每条横向路上所花的时间不能超过 *k*。

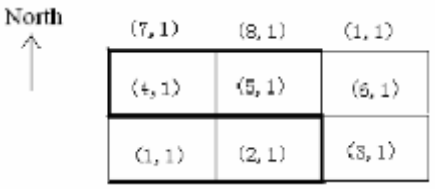


图 1-62

【输入格式】

输入包含多组数据。每组数据的第一行为 3 个整数 *n*, *m*, *k*（1≤*n*≤100，1≤*m*≤10 000，0≤*k*≤3 000 000）；以下 *n*+1 行每行 *m* 个整数，按照从北到南、由西向东的顺序给出各段横向道路上的高兴值；再以下 *n*+1 行每行 *m* 个整数，按照从北到南、由西向东的顺序给出各段横向道路的行走时间。

【输出格式】

对于每组数据，输出最大高兴值。

洞穴（Cave, Chengdu 2007, LA 4015）

一棵 *n* 个结点的有根树，树的边有正整数权，表示两个结点之间的距离。你的任务是回答这样的询问：从根结点出发，走不超过 *x* 单位距离，最多能经过多少个结点？同一个结点经过多次只算一个。

【输入格式】

输入包含多组数据。每组数据的第一行为结点数 *n*（0≤*n*≤500）；以下 *n*−1 行每行 3 个整数 *i*, *j*, *d*（1≤*d*≤10 000），表示结点 *i* 的父亲为结点 *j*，二者的距离为 *d*；洞穴编号为 0~*n*−1；下一行包含整数 *Q*（1≤*Q*≤1 000），即查询的个数；以下 *Q* 行每行包含一个整数 *x*（0≤*x*≤5 000 000），即机器人可以行走的最大距离。输入结束标志为 *n*=0。

【输出格式】

对于每组数据，输出 Q 行，每个查询占一行，即可以经过的结点数的最大值。

帮助布布（Help Bubu, Wuhan 2009, LA 4490）

书架上有 n 本书。如果从左到右写下书架上每本书的高度，我们能够得到一个序列，比如 30,30,31,31,32。我们把相邻的高度相同的书看成一个片段，并且定义该书架的混乱程度为片段的个数。比如，30,30,31,31,32 的混乱程度为 3。同理，30,32,32,31 的混乱程度也是 3，但 31,32,31,32,31 的混乱程度高达 5（请想象一下这个书架，确实够乱的吧。）。

为了整理书架，你最多可以拿出 k 本书，然后再把它们插回书架（其他书的相对顺序保持不变），使书架的混乱程度降至最低。

【输入格式】

输入包含不超过 20 组数据。每组数据的第一行为两个正整数 n 和 k（$1 \leqslant k \leqslant n \leqslant 100$）；第二行包含 n 个整数 h_i（$25 \leqslant h_i \leqslant 32$），即初始时从左到右每本书的高度。输入结束标志为 $n=k=0$。

【输出格式】

对于每组数据，输出在整理结束后书架混乱程度的最小值。

消灭妖怪（Masud Rana, UVa 11600）

某国有 n 个城市，编号为 1~n。这些城市两两之间都有一条双向道路（一共有 $n(n-1)/2$ 条），其中一些路上有妖怪，其他路是安全的。为了保证城市间两两可达，你第一天晚上住在城市 1，然后每天白天随机选择一个新的城市，然后顺着它与当前所在城市之间的道路走过去，途中消灭这条道路上所有的妖怪，晚上住在这座城市。在平均情况下，需要多少个白天才能让任意两个城市之间均可以不经过有妖怪的道路而相互可达？

【输入格式】

输入的第一行为数据组数 T（$T \leqslant 100$）。每组数据的第一行为两个整数 n 和 m（$1 \leqslant n \leqslant 30$，$0 \leqslant m \leqslant n(n-1)/2$），即城市数目和安全的道路数目；以下 m 行每行两个整数 a 和 b（$1 \leqslant a,b \leqslant n$），表示连接城市 a 和城市 b 的道路是安全的（即没有妖怪）。

【输出格式】

对于每组数据，输出平均情况下需要的白天数目。

疏散计划（Evacuation Plan, NEERC 2010, LA 4987）

战争时期，有 n 支施工队在修一条笔直的高速公路，其中第 i 支队伍离高速公路起点的距离为 a_i。另外还有 m 个避难所，其中第 i 个避难所离高速公路起点的距离为 b_i。给每只施工队分配一个避难所，以方便其在敌人轰炸时能够迅速逃往避难。假定施工队 i 分配到避难所 j，施工队的移动距离为 $|a_i-b_j|$。由于避难所的门只能从里面反锁，要求每个避难所至少应分配一支施工队。你的任务是确定分配方案，使得所有施工队移动的总距离最小。

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 4\,000$）；第二行包含 n 个整数，即 $a_1, a_2, \ldots, a_n$。第三行为整数 m（$1 \leqslant m \leqslant n$）。第四行包含 m 个整数，即 $b_1, b_2, \ldots, b_m$。所有坐标均为不超过 10^9 的正整数，且不同的施工队位置不同，不同的避难所位置也不同。输入结束标志为文件结束符（EOF）。

【输出格式】

对于每组数据，第一行输出最小总距离，第二行输出每个施工队分配的避难所编号。避难所按照输入顺序编号为 1~m。

独占访问 2（Exclusive Access 2, NEERC 2009, LA 4593）

在一个庞大的系统里运行着 n 个守护进程。每个进程恰好用到两个资源。这些资源不支持并发访问，所以这些进程通过锁来保证互斥访问。每个进程的主循环如下。

```
loop forever
DoSomeNonCriticalWork()
  P.lock()
  Q.lock()
  WorkWithResourcesPandQ()
  Q.unlock()
  P.unlock()
end loop
```

注意，P 和 Q 的顺序是至关重要的。如果某进程用到了消息队列和数据库，“先获取数据库的锁”与“先获取消息队列的锁”可能会产生截然不同的效果。给定每个进程所需要的两种资源，你的任务是确定每个进程获取锁的顺序，使得进程永远不会死锁，且最坏情况下，等待链的最大长度最短。

在本题中，一个长度为 n 的等待链是一个不同资源和不同进程的交替序列：$R_0\ c_0\ R_1\ c_1 \ldots R_n\ c_n\ R_{n+1}$，其中进程 c_i 已经获取 R_i 的锁，正在等待 R_{i+1} 的锁。当 $R_0=R_{n+1}$ 时死锁，否则说明已获取 R_{n+1} 的锁的进程正在执行操作（而非等待中）。

【输入格式】

输入包含多组数据。每组数据的第一行为整数 n（$1 \leqslant n \leqslant 100$）；以下 n 行每行两个 L~Z 之间的大写字符（即一共有 15 种资源），即该进程需要的两个资源。输入结束标志为文件结束符（EOF）。

【输出格式】

对于每组数据，第一行输出最坏情况下等待链的最大长度。以下 n 行每行输出两个字符，表示该进程获取锁的顺序（先获取第一个字符对应资源的锁）。

基金管理（Fund Management, NEERC 2007, LA 4048）

你有 c 美元，但没有股票。给你 m 天时间和 n 支股票供你买卖（$c \leqslant 10^8$，$m \leqslant 100$，$n \leqslant 8$），问最后最多能剩下多少钱（最后一天结束时不能持有任何股票）。。

每天只能买卖一支股票，并且只能买一手或卖一手，对于第 i 支股票来说，已知第 j 天的价格为 $p_{i,j}$，一手为 s_i 股，且每天最多能持有 k_i 手这支股票。所持股票的总“手”数每天都不能超过 k。$k \leqslant 8$，$s_i \leqslant 10^6$，$k_i \leqslant k$，$0 < p_{i,j} \leqslant 1\,000$。

一个调度问题（A Scheduling Problem, Kaoshiung 2006, LA 3683）

有 n（$n \leqslant 200$）个恰好需要一天完成的任务，要求用最少的时间完成所有任务。任务可以并行完成，但必须满足一些约束。约束分有向约束和无向约束两种，其中，A→B 表示 A 必须在 B 之前完成，A-B 表示 A 和 B 不能在同一天完成。输入保证约束图是将一棵树的某

些边定向后得到的。

（提示：有这样一个定理。设由有向边组成的最长路长度为 k，则答案为 k 或者 k+1）。

书架（Bookcase, NWERC 2006, LA 3637）

有 n（3≤n≤70）本书，每本书有一个高度 H_i 和宽度 W_i（150≤H_i≤300，5≤W_i≤30）。现在要构建一个 3 层的书架，你可以选择将 n 本书放在书架的哪一层。设 3 层高度（该层书的最大高度）之和为 h，书架总宽度（即每层总宽度的最大值）为 w，要求 h×w 尽量小。

修缮长城（Fixing the Great Wall, CERC 2004, LA 3181）

长城被看做一条直线段，有 n 个损坏点需要用机器人 GWARR 修缮。可以用三元组(x_i,c_i,d_i)描述第 i 个损坏点的参数，其中，x_i 是位置，c_i 是立刻修缮（即时刻=0 时开始修缮）的费用，d_i 是单位时间增加的修缮。换句话说，如果在时刻 t_i 开始修缮第 i 个损坏点，费用为 c_i+$t_i$$d_i$。修缮的时间忽略不计，GWARR 的速度恒定为 v，因此从修缮点 i 走到修缮点 j 需要|x_i−x_j|/v 单位的时间。

【输入格式】

输入包含多组数据。每组数据和第一行包含 3 个整数：n, v, x（1≤n≤1 000，1≤v≤100，1≤x≤500 000），即损坏点的个数、GWARR 的速度和初始坐标；以下 n 行每行包含 3 个整数 x_i, c_i, d_i（1≤x_i≤500 000，0≤c_i≤50 000，1≤d_i≤50 000），含义如题。输入保证损坏点的位置各不相同，且 GWARR 的初始位置不与任何一个损坏点重合。输入结束标志为 n=v=x=0。

【输出格式】

对于每组数据，输出一行，即最小费用（用截尾法保留整数部分）。输入保证最小费用不超过 10^9。

给孩子起名（The Best Name for Your Baby, Japan 2006, LA 3623）

给定一个包含 n 条规则的上下文无关文法和长度 l，求出满足该文法的字符串中，长度恰好为 l 的字典序最小串。

“满足文法”是指可以不断使用规则，把单个大写字母 S 变成这个字符串。每条规则形如 A→a，其中 A 是一个大写字母（表示非终结符），a 是一个由大小写字母组成的字符串。该规则的含义是可以用字符串 a 来替换当前字符串中的大写字母 A（如果有多个 A，每次只替换一个）。

比如，有 4 条规则：S→aAB，A→，A→Aa，B→AbbA，那么 aabb 满足该文法，因为 S→aAB（规则 1）→aB（规则 2）→aAbbA（规则 4）→aAabbA（规则 2）→（规则 3）→aAabb（规则 2）→aabb（规则 2）。

【输入格式】

输入包含多组数据。每组数据的第一行为两个整数 n 和 l（1≤n≤50，0≤l≤20），即规则的个数和字符串的长度。接下来的 n 行每行为一条规则，其中每条规则的第一个字母为大写字母，然后是等号，然后是由大小写字母组成的字符串。等号右端的字符串长度不超过 10，且可以为空串。规则内部无空白字符。输入结束标志为 n=l=0。

【输出格式】

对于每组数据，输出满足条件的字典序最小串。如果不存在，输出一个字符'-'。

方块消除（Blocks, UVa 10559）

n（n≤200）个带颜色的方块排成一列。每次可以选择一段颜色相同的连续方块消除，得分为 x^2，其中 x 是这一段连续方块的个数。图 1-63 是两种可能的操作序列，其中第一个是最优的。

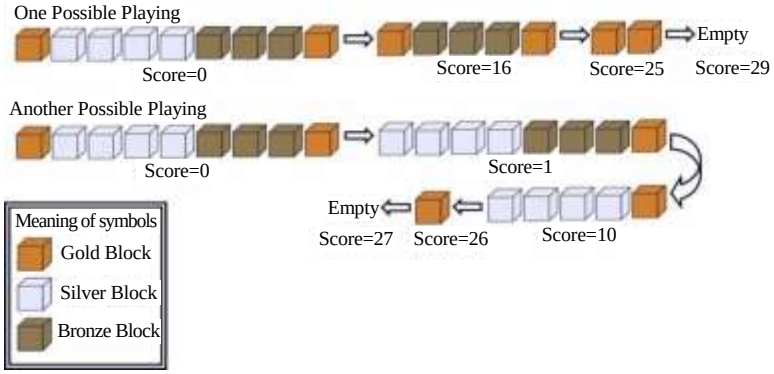


图 1-63

你的任务是消除所有的方块，以得到尽量多的分数。

越大越好（Bigger is Better, Xi'an 2006, LA 3782）

你的任务是用不超过 n（n≤100）根火柴摆一个尽量大的，能被 m（m≤3 000）整除的正整数，如图 1-64 所示。例如 n=6 和 m=3，解为 666。无解时输出-1。

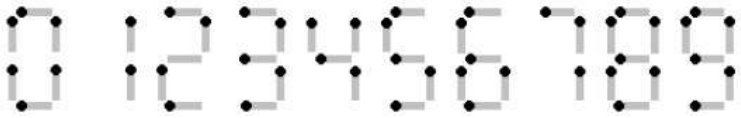


图 1-64

整数传输（Integer Transmission, Beijing 2007, LA 4031）

你要在一个仿真网络中传输一个 n 比特的非负整数 k。各比特从左到右传输，第 i 个比特的发送时间为 i。每个比特的网络延迟总是为 1~d 的实数（因此从左到右第 i 个比特的到达时间为 i~i+d 之间）。若同时有多个比特到达，实际收到的顺序任意。求实际收到的整数有多少种，以及它们的最小值和最大值。

【输入格式】

输入包含多组数据。每组数据仅一行，为 3 个整数 n, d, k（1≤n≤64，0≤d≤n，0≤k<2^n）。输入结束标志为 n=d=k=0。

【输出格式】

对于每组数据，输出可能收到的整数个数以及最小值和最大值。

压缩（Compressor, UVa 11521）

压缩一个字符串，在压缩串中，[S]k 表示 S 重复 k 次，[S]k{S_1}t_1{S_2}t_2…{S_r}t_r 表示 S 重复 k 次，然后在其中第 t_i 个 S 后面插入 S_i。压缩是递归进行的，因此上面的 S, S_1, S_2,…也可以是压缩串。你的任务是使压缩串的长度最小。比如，I_am_WhatWhat_is_WhatWhat 的最

优压缩结果是 I_am_[What]4{_is_}2。注意，上述 k, t_1, t_2,…的长度均算作 1，哪怕它们的十进制表示中包含超过 1 个数字。

【输入格式】

输入包含不超过 20 组数据。每组数据包含不超过 200 个可打印字符，但不含空白字符、括号（小括号()、方括号[]或者花括号{}都算括号）或者数字。字母是大小写敏感的。

【输出格式】

对于每组数据，输出长度和压缩串。如果有多个解，任意输出一个压缩串即可。