HDU1043

题目描述（HDU1043）：十五数码问题是由 15 块滑动的方块构成的，在每一块上有一个 1～15 的数字，所有方块都是一个 4×4 的排列，其中一块方块丢失，称之为“x”。拼图的目的是排列方块，使其按以下顺序排列：

```
1   2   3   4
5   6   7   8
9   10  11  12
13  14  15  x
```

其中唯一合法的操作是将“x”与相邻的方块之一交换。下面的移动序列解决了一个稍微混乱的拼图：

```
1   2   3   4     1   2   3   4     1   2   3   4     1   2   3   4
5   6   7   8     5   6   7   8     5   6   7   8     5   6   7   8
9   x   10  12    9   10  x   12    9   10  11  12    9   10  11  12
13  14  11  15    13  14  11  15    13  14  x   15    13  14  15  x
                r->               d->               r->
```

上一行中的字母表示在每个步骤中“x”方块的哪个邻居与“x”交换；合法值分别为“r”“l”“u”和“d”，表示右、左、上和下。

在这个问题中，编写一个程序来解决八数码问题，它由 3×3 的排列组成。

输入：输入包含多个测试用例，描述是初始位置的方块列表，从上到下列出行，在一行中从左到右列出方块，其中的方块由数字 1～8 加上“x”表示。例如以下拼图:

```
1 2 3
x 4 6
7 5 8
```

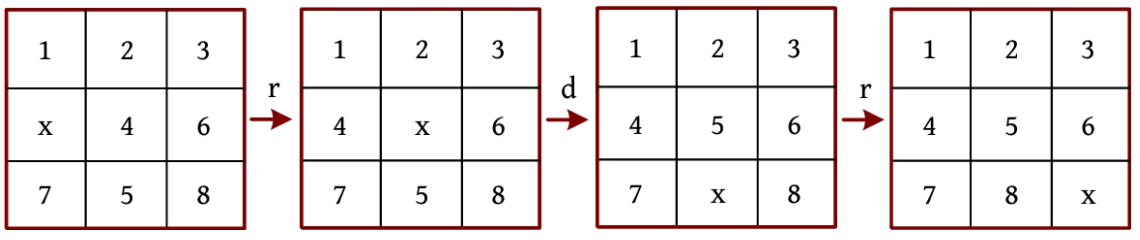
由以下列表描述：

```
1 2 3 x 4 6 7 5 8
```

输出：如果没有答案，则输出“unsolvable”，否则输出由字母“r”“l”“u”和“d”组成的字符串，描述产生答案的一系列移动。字符串不应包含空格，并从行首开始。

输入样例	输出样例
2 3 4 1 5 x 7 6 8	ulddrurdllurdruld

题解：本题为**八数码问题**，包含多个测试用例，同一题目的 POJ1077 数据较弱，只有 1 个测试用例。要求通过 x 方块上下左右四个方向移动，经过最少的步数达到目标状态。例如，初始状态 1 2 3 x 4 6 7 5 8，经过 r、d、r 等 3 步后达到目标状态。



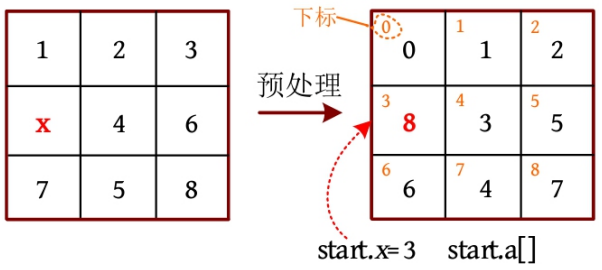
注意：答案不唯一，任一正确答案均可。**可以采用 A*算法、IDA*算法或打表解决。**

1. A*算法

本题采用**康托展开判断重复状态，以当前状态和目标状态的曼哈顿距离作为启发函数，评估函数为已走过的步数+启发函数，评估函数值越小越优先。**从初始状态开始，根据优先队列广度优先搜索目标状态。

1）预处理

首先将字符串读入，例如，1 2 3 x 4 6 7 5 8，将 x 转换为数字 8，其他字符 1～8 转换成数字 0～7。转换之后的棋盘如下图所示。用 start.x 记录 x 所在位置的下标，方便以后移动。



2）可解性判断

把除 x 外的所有数字排成一个序列，求序列的逆序对数。逆序对数指对于第 i 个数，后面有多少个数比它小。例如，对于 1 2 3 x 4 6 7 5 8，6 后面有一个数 5 比它小，6 和 5 是一个逆序对，7 后面有一个数 5 比它小，7 和 5 是一个逆序对，该序列共两个逆序对。数码问题可以被看作 N×N 的棋盘，八数码问题 N=3，十五数码问题 N=4。对于每一次交换操作，左右交换都不改变逆序对数，上下交换时逆序对数增加(N-1)、减少(N-1)或不变。

- **N 为奇数时**：上下交换时每次增加或减少的逆序对数都为偶数，因此每次移动逆序对数，奇偶性不变。若初态的逆序对数与目标状态的逆序对数的奇偶性相同，则有解。
- **N 为偶数时**：上下交换时每次增加或减少的逆序对数都为奇数，上下交换一次，奇偶性改变一次。因此需要计算初态和目标状态 x 相差的行数 k，若初态的逆序对数加上 k 与目标状态逆序对数奇偶性相同，则有解。

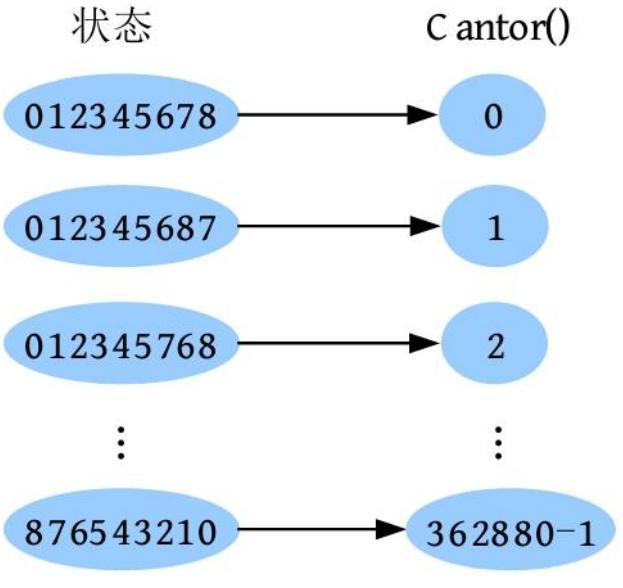
八数码问题 N=3，若初态的逆序对数与目标状态逆序对数奇偶性相同，则有解。本题目标状态的逆序对数为 0，因此初态的逆序对数必须为偶数才有解。注意：统计逆序对数时 x 除外。

算法代码：

```
bool check(node s){//判断是否有解（初态的逆序对数为偶数）
   int cnt=0;
   for(int i=0;i<9;i++){
     if(s.a[i]==8) continue;
     for(int j=i+1;j<9;j++)
             if(s.a[j]<s.a[i]) cnt++;
     }
   if(cnt%2) return 0;
   return 1;
}
```

3）康托展开判重

在 A*算法中，每种状态只需在第一次取出时扩展一次。如何判断这种状态已经扩展过了呢？可以设置哈希函数或使用 STL 中的 map、set 等方法。有一个很好的**哈希方法**是“**康托（Cantor）展开**”，它可以将每种状态都与 0～(9!-1)的整数建立一一映射，快速判断一种状态是否已扩展。状态是数字 0～8 的全排列，共 362 880 个。将所有排列都按照从小到大的顺序映射到一个整数（位序），将排列最小的数 012345678 映射到 0，将排列最大的数 876543210 映射到 362880-1，如下图所示。



如果采用排序算法，则最快 O(n!log(n!))，其中 n=9。而**康托展开可以在 $O(n^2)$时间内将一种状态映射到这个整数。**康托展开是怎么计算的呢？例如，2031，求其在{0,1,2,3}全排列中的位序，其实就是计算排在 2031 前面的排列有多少个，可以按位统计，如下所述。

- 第 0 位的数字 2：在 2031 中，2 后面比 2 小的有两个数字{0,1}。以 0 开头，其他 3 个数字全排列有 3!个，即(0123,0132,0213,0231,0312,0321)；以 1 开头，其他 3 个数字全排列有 3!个(1023,1032,1203,1230,1302,1320)。因此排在以 2 开头的数字之前共 2×3!个数字。
- 第 1 位的数字 0：在 2031 中，0 后面没有比 0 小的数字。
- 第 2 位的数字 3：在 2031 中，3 后面比 3 小的有 1 个数字{1}，前两位 20 已确定，以 1 开头，剩余 1 个数字的全排列有 1!个数字，即 2013。排在 3 之前的共 1×1!个数字。
- 第 3 位的数字 1：在 2031 中，1 后面没有比它小的数字。

因此 2031 的位序为 2×3!+1=13。

位序计算公式：

$$\text{code}=\sum_{i=0}^{n-1} \text{cnt}[i]\times(n-i-1)!$$

其中，cnt[i]为 a[i]后面比 a[i]小的数字个数，n 为数字个数。

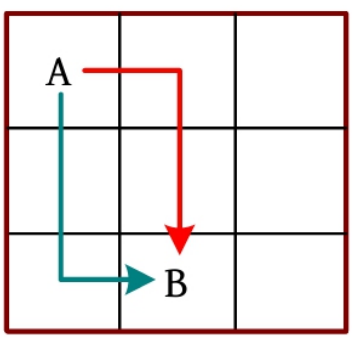
8 数码问题包含 0～8 共 9 个数字，首先求出 0～8 的阶乘并将其保存到数组中。然后统计在每一个数字后面有多少个数字比它小，累加 cnt*fac[8-i]即可得到该状态的位序。**状态与位序之间是一一映射的，无须处理哈希冲突问题。**

算法代码：

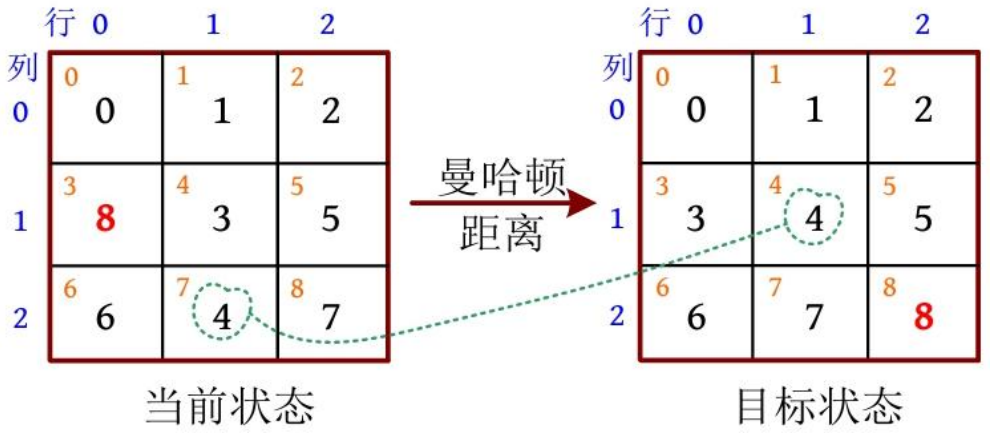
```
fac[0]=1;
for(int i=1;i<9;i++) fac[i]=fac[i-1]*i;
int cantor(node s){//康托判重
   int code=0;
   for(int i=0;i<9;i++){
      int cnt=0;
      for(int j=i+1;j<9;j++)
         if(s.a[j]<s.a[i]) cnt++;
      code+=cnt*fac[8-i];
   }
   return code;
}
```

4）曼哈顿距离

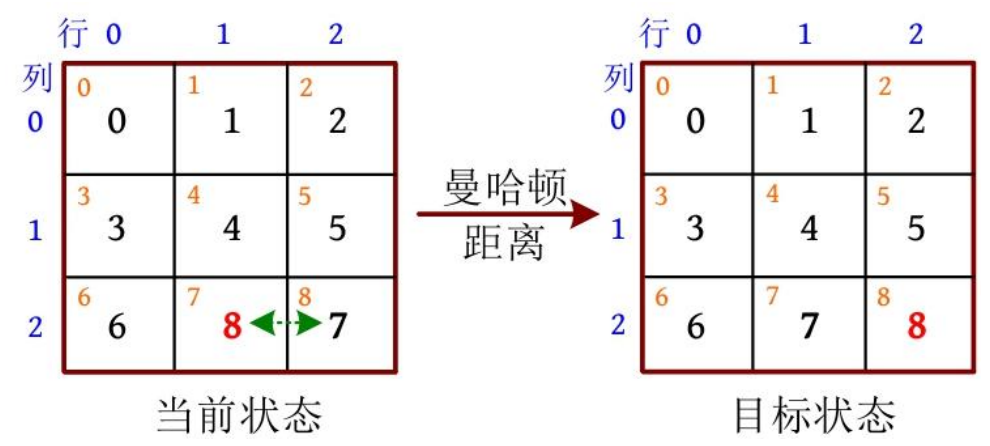
A*算法的启发函数有多种设计方法，可以选择当前状态与目标状态位置不同的数字个数，也可以选择当前状态的逆序对数（目标状态逆序对数为 0），还可以选择当前状态与目标状态的曼哈顿距离。本题选择当前状态和目标状态的曼哈顿距离作为启发函数。曼哈顿距离又被称为“出租车距离”，指行列差的绝对值之和，即从一个位置到另一个位置的最短距离。例如，从 A 点到 B 点，无论是先走行后走列，还是先走列后走行，走的距离都为行列差的绝对值之和。如下图所示，A 和 B 的曼哈顿距离为 2+1=3。



求当前状态与目标状态的曼哈顿距离，需要将两种状态上的数字位置转换为行、列，然后求行、列差的绝对值之和。例如，当前状态和目标状态如下图所示，将位置下标 i 转换为行(i/3)，转换为列(i%3)。当前状态的数字 4 的位置下标为 7，转换为 7/3 行、7%3 列，即 2 行、1 列。目标状态的数字 4 的位置下标为 4，转换为 4/3 行、4%3 列，即 1 行、1 列。两个位置的曼哈顿距离为|2-1|+|1-1|=1。



除了 8（x 滑块），计算当前状态和目标状态中每个位置的曼哈顿距离之和。曼哈顿距离为什么不需要计算 8（x 滑块）？因为其他数字都是通过和滑块交换达到目标状态的。例如下图中，当前状态只有数字 7，与目标状态的数字 7 位置不同，差一个曼哈顿距离，与滑块交换一次，7 即可归位。当所有数字都与目标状态的位置相同时，滑块自然跑到了它应该在的位置。如果计算 8（x 滑块）的曼哈顿距离，那么当前状态和目标状态的曼哈顿距离为 2，很明显是错误的，进行一步交换就可以达到目标状态。



算法代码：

```
int h(node s){//启发函数，曼哈顿距离（行列差的绝对值之和）
     int cost=0;
     for(int i=0;i<9;i++){
          if(s.a[i]!=8)
               cost+=abs(i/3-s.a[i]/3)+abs(i%3-s.a[i]%3);
     }
     return cost;
}
```

5）A*算法算法步骤：

（1）创建一个优先队列，将评估函数 f(t)=g(t)+h(t)作为优先队列的优先级，g(t)为已走过的步数，h(t)为当前状态与目标状态的曼哈顿距离，f(t)越小越优先。计算初始状态的启发函数 h(start)，计算初始状态的康托展开值 cantor(start)并标记已访问，初始状态入队。

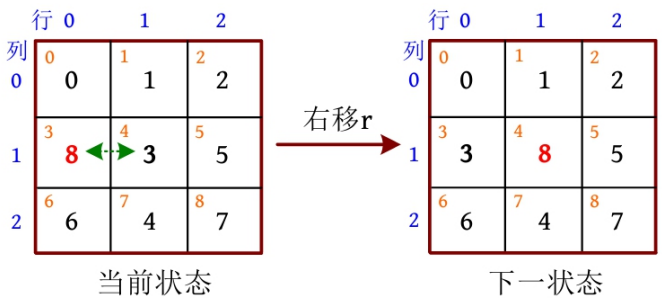
（2）如果队列不空，则队头 t 出队，否则算法结束。

（3）计算康托展开值 k_s=cantor(t)，从 t 出发向 4 个方向扩展。

计算 x 新位置的行列值。

```
int row=t.x/3+dir[i][0];//行
int col=t.x%3+dir[i][1];//列
int newx=row*3+col;//转换为下标
```

例如，如下图所示，当前状态 x（数字 8）的位置 t.x=3，将其转换为 3/3=1 行、3%3=0 列，向右移动一格后，x 的新位置为 1 行、1 列，转换为下标为 4。



当前状态　　下一状态

如果新位置超出边界，则继续下一循环，否则令新旧位置上的数字交换，记录新状态 x 的位置。计算新状态的评估函数，nex.g++; nex.h=h(nex); ex.f=nex.g+nex.h;

计算新状态的康托展开值 k_n=cantor(nex)，如果该状态已被访问，则继续下一循环；否则标记已访问，并将新状态入队。

```
pre[k_n]=k_s; //记录新状态的前驱，康托展开值唯一标识该状态
ans[k_n]=to_c[i]; //记录移动方向字符
```

如果 k_n=0，则说明已找到目标（目标状态康托展开值为 0），返回。

算法代码：

```
void Astar(){
   int k_s,k_n;
   priority_queue<node>q;
   while(!q.empty()) q.pop();
   memset(vis,0,sizeof(vis));
   start.g=0;start.f=start.h=h(start);
   vis[cantor(start)]=1;
   q.push(start);
   while(!q.empty()){
      node t=q.top();
      q.pop();
      k_s=cantor(t);
      for(int i=0;i<4;i++){
         nex=t;
         int row=t.x/3+dir[i][0];
         int col=t.x%3+dir[i][1];
         int newx=row*3+col;//转换为下标
         if(row<0||row>2||col<0||col>2) continue;
         swap(nex.a[t.x],nex.a[newx]);
         nex.x=newx;
         nex.g++;
         nex.h=h(nex);
         nex.f=nex.g+nex.h;
         k_n=cantor(nex);
         if(vis[k_n]) continue;
         vis[k_n]=1;
         q.push(nex);
         pre[k_n]=k_s;
         ans[k_n]=to_c[i];
         if(k_n==0) return;
      }
   }
   return;
}
```

2. IDA*算法

IDA*算法是带有评估函数的迭代加深 DFS 算法，本题设计**评估函数 f(t)=g(t)+h(t)，g(t)为已走过的步数，h(t)为当前状态与目标状态的曼哈顿距离。**

算法步骤：

（1）从 depth=1 开始进行深度优先搜索。

（2）计算当前状态与目标状态的曼哈顿距离 t=h()，如果 t=0，则说明已找到目标，ans[d]='\0'，返回 1。如果 d+t > depth，则返回 0。

（3）从当前状态出发，沿 4 个方向扩展。

（4）如果没有找到目标，则增加深度，++depth，继续搜索。

算法代码：

```
bool dfs(int x,int d,int pre){
    int t=h();
    if(!t){
        ans[d]='\0';
        return 1;
    }
    if(d+t>depth) return 0;
    for(int i=0;i<4;i++){
        int row=x/3+dir[i][0];
        int col=x%3+dir[i][1];
        int newx=row*3+col;//转换为数字
        if(row<0||row>2||col<0||col>2||newx==pre) continue;
        swap(a[newx],a[x]);
        ans[d]=str[i];
        if(dfs(newx,d+1,x)) return 1;
        swap(a[newx],a[x]);
    }
    return 0;
}

void IDAstar(int x){
    depth=0;
    while(++depth){
        if(dfs(x,0,-1))
            break;
    }
}
```

IDA*算法优化算法：上面的 IDA*算法深度从 1 开始，每次都增加 1，这样搜索的速度不快。其实可以从初始状态到目标状态的曼哈顿距离开始，**每次都增加上一次搜索失败的最小深度，从而提高搜索效率。**

算法步骤：

（1）从 depth=h()开始进行深度优先搜索。

（2）计算当前状态与目标状态的曼哈顿距离 t=h()，如果 t=0，则说明已找到目标，ans[d]='\0'，返回 1。如果 d+t > depth，则更新 mindep=min(mindep,d+t)，返回 0。

（3）从当前状态出发，沿着 4 个方向扩展。

（4）如果没有找到目标，则增加深度，depth=mindep，继续搜索。

算法代码：

```
bool dfs(int x,int d,int pre){
     int t=h();
     if(!t){
          ans[d]='\0';
          return 1;
     }
     if(d+t>depth){
          mindep=min(mindep,d+t);
          return 0;
     }
     for(int i=0;i<4;i++){
          int row=x/3+dir[i][0];
          int col=x%3+dir[i][1];
          int newx=row*3+col;//转换为数字
          if(row<0||row>2||col<0||col>2||newx==pre) continue;
          swap(a[newx],a[x]);
          ans[d]=to_c[i];
          if(dfs(newx,d+1,x)) return 1;
          swap(a[newx],a[x]);
     }
     return 0;
}

void IDAstar(int x){
     depth=h();
     while(1){
          mindep=inf;
          if(dfs(x,0,-1))
               break;
          depth=mindep;
     }
}
```

3. 打表

打表是一种典型的**用空间换时间**的技巧，一般指将所有可能需要用到的结果都事先计算出来，在后面需要用到时可以直接查表。当所有的可能状态都不多时，用打表的办法速度更快。从目标状态开始进行广度优先搜索，反向搜索所有状态，记录该状态的前驱及方向字符。记录方向字符时，因为是倒推的，所以左移相当于前一状态到目标状态的右移，因此方向字符为 r，如下图所示。

对每一个状态都用康托展开值作为唯一标识，如果求解从某一个状态到目标状态，则可以直接根据该状态的前驱数组找到目标状态。如果初始状态没被标记过，则说明从该状态无法到达目标状态。

算法代码：

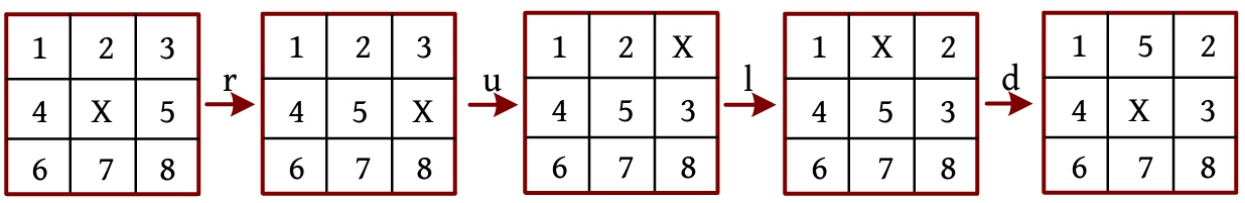
```
void get_all_result(){//打表求解所有答案
   int k_s,k_n;
   memset(vis,0,sizeof(vis));
   for(int i=0;i<9;i++)
     st.a[i]=i;
   st.x=8;
   vis[cantor(st)]=1;
   q.push(st);
   while(!q.empty()){
      node t=q.front();
      q.pop();
      k_s=cantor(t);
      for(int i=0;i<4;i++){
         node nex=t;
         int row=t.x/3+dir[i][0];
         int col=t.x%3+dir[i][1];
         int newx=row*3+col;//转换为下标
         if(row<0||row>2||col<0||col>2) continue;
         nex.a[t.x]=t.a[newx];
         nex.a[newx]=8;
         nex.x=newx;
         k_n=cantor(nex);
         if(vis[k_n]) continue;
         vis[k_n]=1;
         q.push(nex);
         pre[k_n]=k_s;
         ans[k_n]=to_c[i];
      }
   }
}
```

4 种算法的运行时间及空间比较如下表

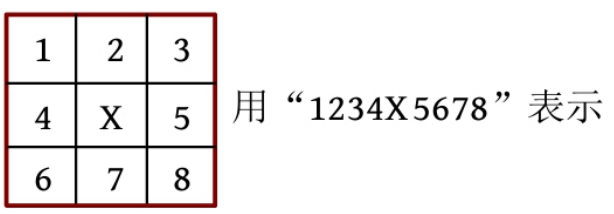
算　法	搜索策略	题　目	运行时间	占用空间
A*算法	cantor+曼哈顿距离+优先队列广度优先搜索	HDU1043	733ms	6.9MB
IDA*算法	曼哈顿距离+迭代加深深度优先搜索	HDU1043	202ms	1.2MB
IDA*优化算法	曼哈顿距离+迭代加深深度优先搜索	HDU1043	124ms	1.2MB
打表	cantor+广度优先搜索	HDU1043	93ms	5.6 MB

HDU3567

描述（HDU3567）：八数码，也叫作“九宫格”，来自一个古老的游戏。在这个游戏中，你将得到一个 3×3 的棋盘和 8 个方块。方块的编号为 1～8，其中一块方块丢失，称之为“X”。“X”可与相邻的方块交换位置。用符号“r”表示将“X”与其右侧的方块进行交换，用“l”表示左侧的方块，用“u”表示其上方的方块，用“d”表示其下方的方块。



棋盘的状态可以用字符串 S 表示，使用下面显示的规则。



问题是使用“r”“u”“l”“d”操作列表可以将棋盘的状态从状态 A 转到状态 B，需要找到满足以下约束的结果：

（1）在所有可能的解决方案中，它的长度最小；

（2）它是所有最小长度解中词典序最小的一个。

输入：第 1 行是 T（T≤200），表示测试用例数。每个测试用例的输入都由两行组成，状态 A 位于第 1 行，状态 B 位于第 2 行。保证从状态 A 到状态 B 都有有效的解决方案。

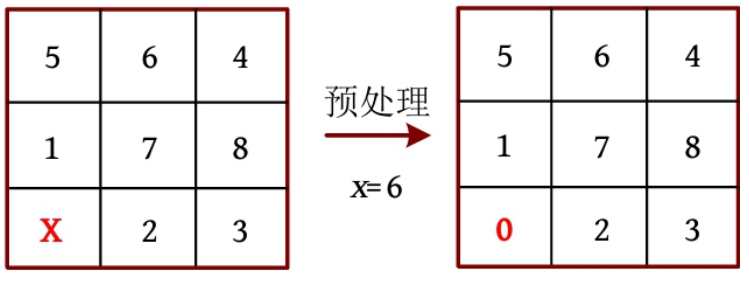
输出：对于每个测试用例，都输出两行。第 1 行是“Case x:d”格式，其中 x 是从 1 开始计算的案例号，d 是将 A 转换到 B 的操作列表的最小长度。第 2 行是满足约束条件的操作列表。

输入样例	输出样例
2 12X453786 12345678X 564178X23 7568X4123	Case 1: 2 dd Case 2: 8 urrulldr

题解：本题为八数码问题，与八数码问题（HDU1043）不同的是，**本题的终态（目标状态）不是固定不变的，而是由输入确定的。**要求从初态 A 到终态 B，输出最少的步数和操作序列，而且如果最小步数相同，则输出字典序最小的一个。本题保证有解，无须可解性判断，可以采用 A*、IDA*算法解决，在此采用 IDA*算法。

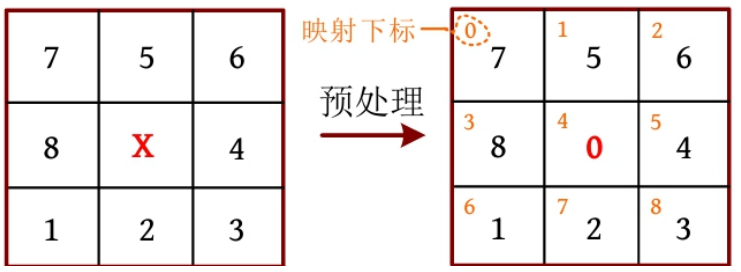
1. 算法设计

（1）读入初态，用变量 x 记录“X”出现的位置 i，令 a[i]=0，将其他位置减去“0”转换成数字。例如，初态为 564178X23，用变量 x 记录“X”出现的位置 6，转换之后的棋盘如下图所示。

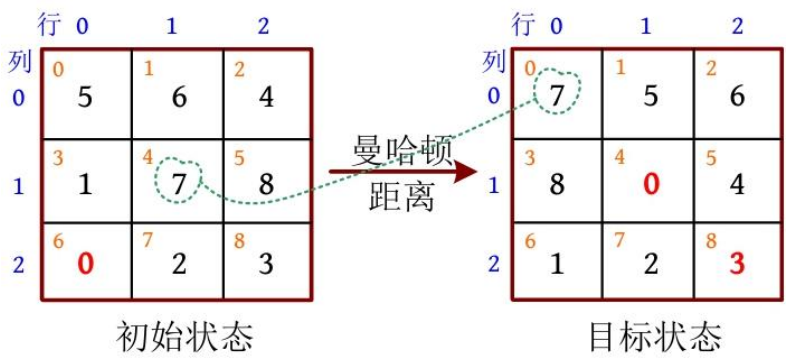


（2）读入终态，“X”出现的位置为 i，令 goal[i]=0，其他位置减去“0”转换成数字。上题（HDU1043）中目标状态数字正好等于位置下标，本题中的目标状态是根据输入数据确定的，为了方便计算启发函数，对目标状态建立一个从数字到位置下标的映射。将 goal[i]映射到位置下标 i，m[goal[i]]=i。

例如，终态为 7568X4123，转换之后的棋盘如下图所示，m[7]=0，m[6]=2。



（3）计算初态启发函数并初始化深度 depth=h()。如下图所示，初始状态中数字 7 的位置下标为 4，转换为 4/3 行、4%3 列，即 1 行、1 列。目标状态中数字 7 的映射位置下标为 0，转换为 0/3 行、0%3 列，即 0 行、0 列。两个位置的曼哈顿距离为|1-0|+|1-0|=2。除了 0（X 滑块），计算当前状态和目标状态中每个位置的曼哈顿距离之和。



（4）深度优先搜索，计算当前状态的启发函数 h()，如果正好为 0，则找到目标输入答案，返回 1。如果 d+t>depth，则更新 mindep=min(mindep, d+t)，返回 0。

（5）沿着 4 个方向搜索，如果 x 的新位置未出边界、不是前一个位置，则交换原位置和新位置，记录操作序列，从新位置开始深度加 1，进行深度优先搜索，如果找到答案，则返回 1，否则交换原位置和新位置，还原现场并回溯。

（6）如果未找到答案，则深度为 depth=mindep，继续进行**迭代加深搜索。**

2. 算法实现

定义方向数组及操作序列，操作序列字母按字典序排序。

```
const int dir[4][2]={{1,0},{0,-1},{0,1},{-1,0}};//方向数组
const char str[]={'d','l','r','u'};//保证字母按字典序排序
int h(){//启发函数，欧氏距离（行列差绝对值之和）
    int cost=0;
    for(int i=0;i<9;i++){
        if(a[i])
            cost+=abs(i/3-m[a[i]]/3)+abs(i%3-m[a[i]]%3);
    }
    return cost;
}

bool dfs(int x,int d,int pre){
    int t=h();
    if(!t){
        printf("%d\n",d);
        ans[d]='\0';
        printf("%s\n",ans);
        return 1;
    }
    if(d+t>depth){
        mindep=min(mindep,d+t);
        return 0;
    }
```

```
    for(int i=0;i<4;i++){
        int row=x/3+dir[i][0];
        int col=x%3+dir[i][1];
        int newx=row*3+col;//转换为数字
        if(row<0||row>2||col<0||col>2||newx==pre) continue;
        swap(a[newx],a[x]);
        ans[d]=str[i];
        if(dfs(newx,d+1,x)) return 1;
        swap(a[newx],a[x]);
    }
    return 0;
}

void IDAstar(int x){
    depth=h();
    while(1){
        mindep=inf;
        if(dfs(x,0,-1))
            break;
        depth=mindep;
    }
}
```

POJ2449

题目描述（POJ2449）：给定一个有向图，N 个节点，M 条边。求从源点 S 到终点 T 的第 K 短路。路径可能包含两次或两次以上的同一节点，甚至是 S 或 T。具有相同长度的不同路径将被视为不同。

输入：第 1 行包含两个整数 N 和 M（1≤N≤1000，0≤M≤100 000）。节点编号为 1～N。以下 M 行中的每一行都包含 3 个整数 A、B 和 T（1≤A, B≤N，1≤T≤100），表示从 A 到 B 有一条直达的路径，需要时间 T。最后一行包含 3 个整数 S、T 和 K（1≤S, T≤N，1≤K≤1000）。

输出：单行输出第 K 短路径的长度（所需时间）。如果不存在第 K 短路，则输出-1。

输入样例	输出样例
2 2 1 2 5 2 1 4 1 2 2	14

题解：本题求第 K 短路。如果采用优先队列式广度优先搜索算法，则记录当前节点 v 和源点 s 到 v 的最短路径长度(v, dist)。首先将(s, 0)入队，然后每次都从优先队列中取出 dist 最小的二元组(x, dist)，对 x 的每一个邻接点 y 都进行扩展，将新的二元组(y, dist+w(x, y))入队。第 1 次从优先队列中取出(x, dist)时，就得到从源点 s 到 x 的最短路径长度 dist。那么在第 i 次从优先队列中取出(x, dist)时，就得到从源点 s 到 x 的第 i 短路径长度 dist。

实际上，从源点 s 到当前节点 x 的最短路径长度最小，并不代表经过 x 就能够得到从源点 s 到终点 t 的最短路径长度。因为余下的路有可能很长，并不知道从 x 到终点 t 的最短路径长度是多少。因此可以考虑采用 A*算法，设置评估函数 f(x)=g(x)+h(x)，其中 g(x)表示从源点 s 到节点 x 的最短路径长度，h(x)表示从节点 x 到终点 t 的最短路径长度。将 f(x)作为优先队列的优先级，f(x)越小，得到从起点到终点最短路径长度的可能性越大。

1. 算法设计

（1）在原图的反向图中，采用 Dijkstra 算法求出从终点 t 到所有节点 x 的最短路径长度 dist[x]。实际上，dist[x]就是原图中从节点 x 到终点 t 的最短路径长度。

（2）如果 dist[s]=inf，则说明从源点无法到达终点，返回-1，算法结束。

（3）在原图中，采用 A*算法求解。用三元组(v, g, h)记录状态，第 1 个参数为当前节点编号，后两个参数分别代表从源点到当前节点的最短路径长度和从当前节点到终点的最短路径长度。优先级为 g+h，其值越小，优先级越高。初始时，将(s, 0, 0)加入优先队列中。

（4）如果队列不空，则队头 p 出队，u=p.v，节点 u 的访问次数加 1，即 times[u]++。如果 u 正好是终点且访问次数为 k，则返回最短路径长度 p.g+p.h，算法结束。

（5）如果 times[u]>k，则不再扩展，否则扩展 u 的所有邻接点 E[i].v，将(E[i].v, p.g+E[i].w, dist[E[i].v])入队。

2. 算法实现

```
int Astar(int s,int t){
   if(dist[s]==inf) return -1;
   memset(times,0,sizeof(times));
   priority_queue<point> Q;
   Q.push(point(s,0,0));
   while(!Q.empty()){
      point p=Q.top();
      Q.pop();
      int u=p.v;
      times[u]++;
      if(times[u]==k&&u==t)
         return p.g+p.h;
      if(times[u]>k)
         continue;
      for(int i=head[u];~i;i=E[i].nxt)
         Q.push(point(E[i].v,p.g+E[i].w,dist[E[i].v]));
   }
   return -1;
}
```

POJ3134

题目描述（POJ3134）：从 x 开始，反复乘以 x，可以用 30 次乘法计算 x^{31}：$x^2=x\times x$，$x^3=x^2\times x$，$x^4=x^3\times x$，…，$x^{31}=x^{30}\times x$。

平方运算可以明显地缩短乘法序列，以下是用 8 次乘法计算 x^{31} 的方法：$x^2=x\times x$，$x^3=x^2\times x$，$x^6=x^3\times x^3$，$x^7=x^6\times x$，$x^{14}=x^7\times x^7$，$x^{15}=x^{14}\times x$，$x^{30}=x^{15}\times x^{15}$，$x^{31}=x^{30}\times x$。

这不是计算 x^{31} 的最短乘法序列。有很多方法只有 7 次乘法，以下是其中之一：$x^2=x\times x$，$x^4=x^2\times x^2$，$x^8=x^4\times x^4$，$x^{10}=x^8\times x^2$，$x^{20}=x^{10}\times x^{10}$，$x^{30}=x^{20}\times x^{10}$，$x^{31}=x^{30}\times x$。

如果除法也可用，则可以找到一个更短的操作序列。可以用 6 个运算（5 乘 1 除）计算 x^{31}：$x^2= x\times x$，$x^4=x^2\times x^2$，$x^8=x^4\times x^4$，$x^{16}=x^8\times x^8$，$x^{32}=x^{16}\times x^{16}$，$x^{31}=x^{32}\div x$。

如果除法和乘法一样快，则这是计算 x^{31} 最有效的方法之一。

编写一个程序，通过从 x 开始的乘法和除法，为给定的正整数 n 找到计算 x^n 的最少运算次数。在序列中出现的乘积和商应该是 x 的正整数幂。

输入：输入是由一行或多行组成的序列，每行都包含一个整数 n（$0<n\le1000$）。以输入 0 结束。

输出：单行输出从 x 开始计算 x^n 所需的最小乘法和除法总数。

输入样例	输出样例
1	0
31	6
70	8
91	9
473	11
512	9
811	13
953	12
0	

题解：本题从 x 开始计算 x^n 所需的最小乘法和除法总数，可以**采用 IDA*算法解决。**

1. 算法设计

（1）初始化，指数 ex[0]=1，深度 depth=0。

（2）进行深度优先搜索，如果 ex[d]=n，则返回 1。如果 d≥depth，则返回 0。如果当前指数在倍增 depth-d 之后还小于 n，则返回 0。

（3）从 0 到 d 执行乘法，ex[d+1]=ex[d]+ex[i]，深度优先搜索 dfs(d+1)，如果成功，则返回 1；执行除法，ex[d+1]=abs(ex[d]-ex[i])，进行深度优先搜索 dfs(d+1)，如果成功，则返回 1。

（4）如果搜索失败，则深度 depth++，重新开始搜索。

2. 算法实现

```
bool dfs(int d){
   if(ex[d]==n) return 1;
   if(d>=depth) return 0;
   if(ex[d]<<(depth-d)<n) return 0;
   for(int i=0;i<=d;i++){
      ex[d+1]=ex[d]+ex[i];//乘法
      if(dfs(d+1)) return 1;
      ex[d+1]=abs(ex[d]-ex[i]);//除法
      if(dfs(d+1)) return 1;
   }
   return 0;
}
void IDAstar(){
   ex[0]=1;
   for(depth=0;;depth++){
      if(dfs(0)){
         printf("%d\n",depth);
         break;
      }
   }
}
```