**Sicily 1151 魔板 解题报告**

*13331231*

*计应2班*

*孙圣*

1. **题目大意**

有一个由8个数码组成的魔板。给定初始状态，将终止状态和最大步数作为输入，输入-1代表结束。有三种对魔板的操作：A，交换上下行；B，将每一位数右移，最右的数移到最左边；C，中间的四个数顺时针旋转一格。如果能在最大步数内将魔板移到终止状态，则输出步数和操作符序列；否则，输出-1。

1. **算法思想及主要的数据结构**

这题给定初始态要寻找目标态，所以是一道搜索的问题。但由于数据量巨大，用深度优先搜索很有可能浪费大量的时间和空间，因此并不合理。所以主要利用广度优先搜索来求解问题，主要使用的数据结构为队列。在每一个节点处，将三个操作后形成的不同状态压入栈中，因此这是一棵三叉树。如果不进行判断是否重复，很有可能进入死循环浪费时间，所以要判断重复从而减少时间的消耗。对于重复的判断可以使用set，也可以利用数组。

1. **解题思路**

主要利用康托展开来实现对存储空间的压缩。

由于8个数码并没有重复，可以将它们视为全排列。8个数的全排列一共有8! = 40320种情况。所以我们可以使用一个布尔类型的大小为40320的数组来帮助我们判断某一状态是否已经出现过。康托展开的公式：X = a[n] \* (n-1)! + a[n-1] \* (n-2)! +...+ a[i] \* (i-1)! +...+ a[1] \* 0!，其中a[i]是指比处在i下标处的数小的且之前未出现的数的个数。因此，对于每个排列，我们都可以根据康托展开得到一个唯一的数来作为整个排列的标识。

利用队列来实现广搜，对于每一个节点，如果其为目标状态则直接跳出循环输出结果，否则分别调用三个操作得到新的状态节点。如果新的状态节点在之前已经出现过，则不将其放入队列；如果之前并未出现，则压入队列。不断的循环，直到步数超过规定步数或者找到目标状态。

1. **算法描述**

定义结构体state：

struct state {

short num[8];

string op;

};

保存8个数码的状态和操作符

进行A操作的函数，返回新的state结构体：

struct state opa(struct state& input)；

进行B操作的函数，返回新的state结构体：

struct state opb(struct state& input)；

进行C操作的函数，返回新的state结构体：

struct state opc(struct state& input)；

进行康托展开的函数，输入为储存8个数码的数组：

int cantor(short a[])；

int factorial[8]；储存7!到0!的值

bool b[8]；储存第i位数是否被访问过

count 储存比a[i]小且尚未出现的数的个数

返回康托展开的值

主函数：

main()

int n；最多允许的步数

进行循环直到n为-1：

struct state final；读入目标状态

struct state init；初始化初始状态

queue<struct state> q; 队列储存将要访问的state

bool visit[40320]; 用于判断节点是否被访问过

int flag = 0; 标识是否成功找到操作序列

进行循环直到队列为空：

struct state front；获得队列头节点

int equal；判断头节点是否与目标节点相同

如果相同，将flag标记并跳出循环

如果超出规定步数，直接跳出循环

struct state temp; 保存新的状态节点

分别进行A, B, C操作

opa();

opb();

opc();

进行康托展开并判断重复

如果节点之前未出现过，则压入队列中

根据flag的值给出对应的输出

1. **程序注释清单**

// Sicily 1151 魔板

#include<iostream>

#include<queue>

#include<string>

#include<cstdio>

using namespace std;

// 用结构体保存8个数码的状态和已经完成的操作符

struct state {

short num[8]; // 用short节省内存使用

string op;

};

// A操作：将上下两行调换

struct state opa(struct state& input) {

struct state temp;

for ( int i = 0; i < 4; ++i ) {

temp.num[i] = input.num[i + 4];

}

for ( int i = 4; i < 8; ++i ) {

temp.num[i] = input.num[i - 4];

}

temp.op = input.op + 'A';

return temp;

}

// B操作：将最右边的数移到最左边，其它的数右移

struct state opb(struct state& input) {

struct state temp;

temp.num[0] = input.num[3];

temp.num[4] = input.num[7];

for ( int i = 0; i < 3; ++i ) {

temp.num[i + 1] = input.num[i];

temp.num[i + 5] = input.num[i + 4];

}

temp.op = input.op + 'B';

return temp;

}

// C操作：将中间的四个数顺时针旋转一格

struct state opc(struct state& input) {

struct state temp;

temp.num[0] = input.num[0];

temp.num[3] = input.num[3];

temp.num[4] = input.num[4];

temp.num[7] = input.num[7];

temp.num[1] = input.num[5];

temp.num[2] = input.num[1];

temp.num[6] = input.num[2];

temp.num[5] = input.num[6];

temp.op = input.op + 'C';

return temp;

}

// 康托展开

int cantor(short a[]) {

// 分别对应7!到0!

int factorial[8] = {5040, 720, 120, 24, 6, 2, 1, 1};

bool b[8];

for ( int i = 0; i < 8; ++i ) {

b[i] = 0;

}

int can = 0;

/\*

\* 从第一个数开始遍历，计算比该数小且尚未出现的数的个数，

\* 并与权值(i!)相乘，即可得到康托展开的值

\*/

for ( int i = 0; i < 8; ++i ) {

int count = 0; // 比a[i]小且尚未出现的数的个数

for ( int j = 0; j < a[i] - 1; ++j ) {

if ( b[j] == 0 ) {

++count;

}

}

can += count \* factorial[i];

b[a[i] - 1] = 1;

}

return can;

}

int main() {

// freopen("a.txt", "r", stdin);

int n;

while ( scanf("%d", &n) ) {

if ( n == - 1 ) {

break;

}

struct state final;

struct state init;

scanf("%hd%hd%hd%hd%hd%hd%hd%hd", &final.num[0], &final.num[1], &final.num[2], &final.num[3],

&final.num[4], &final.num[5], &final.num[6], &final.num[7]);

// 设定初始的结构体

for ( int i = 0; i < 4; ++i ) {

init.num[i] = i + 1;

init.num[i + 4] = 8 - i;

}

init.op = "";

queue<struct state> q; // 队列储存state信息

bool visit[40320]; // 用于排除重复，由于只有8个数，一共只有8!种排列

int flag = 0; // 储存是否能在规定步数(n)内得到结果

q.push(init);

for ( int i = 0; i < 40320; ++i ) {

visit[i] = 0;

}

visit[cantor(init.num)] = 1;

// BFS

while ( !q.empty() ) {

struct state front = q.front();

q.pop();

int equal = 1;

// 判断该state是否与所期望的相同

for ( int i = 0; i < 8; ++i ) {

if ( front.num[i] != final.num[i] ) {

equal = 0;

break;

}

}

// 如果相同，则跳出循环

if ( equal == 1 ) {

final = front;

flag = 1;

break;

}

// 判断是否超出规定步数(n)

if ( front.op.length() > n ) {

break;

}

struct state temp;

int can;

// A操作，并判断得到的结果是否出现过，如未则标记则压入队列中

temp = opa(front);

can = cantor(temp.num);

if ( visit[can] == 0 ) {

visit[can] = 1;

q.push(temp);

}

// B操作

temp = opb(front);

can = cantor(temp.num);

if ( visit[can] == 0 ) {

visit[can] = 1;

q.push(temp);

}

// C操作

temp = opc(front);

can = cantor(temp.num);

if ( visit[can] == 0 ) {

visit[can] = 1;

q.push(temp);

}

}

// 如果成功找到移动方案，则输出步数和操作序列，否则输出-1

if ( flag ) {

cout << final.op.length() << " " << final.op << endl;

} else {

cout << "-1" << endl;

}

}

return 0;

}

1. **测试数据**

**a.**

1

1 2 3 4

8 7 6 5

-1

这组数据测试一个边界情况，即目标状态与终止状态一致时程序的运行情况。正确的输出应该为0。

**b.**

4

5 8 3 2

4 1 6 7

-1

这组数据测试所用步数恰好等于所规定的步数的情况，能够得到最终正确的解，输出为4 ACCB。

**c.**

5

5 8 3 2

4 1 6 7

-1

这组数据测试所用步数小于所规定的步数的情况，能够得到最终正确的解，输出为4 ACCB。

**d.**

3

5 8 3 2

4 1 6 7

-1

这组数据测试在给定的最大步数的条件下不能得到目标状态的情况，因此输出为-1。

**e.**

6

3 4 7 8

6 5 2 1

-1

这组数据测试程序的寻找最优解功能。可由序列ABCCBA得到，但程序输出了更为优化的结果：4 BCCB。

1. **分析与优化**

一开始做这题的时候用队列遇到了时间超出的问题，后来改用大小为40320的数组出现Runtime Error，又改用Vector内存超出的问题。后来才发现是由于在判断重复时出现了问题。当时在实现的时候是在访问某一个节点时才将其标记，而正确的做法是应该在压入队列中的时候就讲其标记，否则依然会存在重复，造成各种问题。

对于程序的优化：第一个版本的程序是按照老师上课讲的方法，运用了两个int(up, down)来存储8个数码，之后用int(level)存储处在第几层，方便判断是否超出步数。用一个char(op)来表示所用的操作。用一个int(num)来表示节点的编号，一个int(pre)来记录父节点的编号。一共用到了21个Bytes。但这样实现较为复杂，涉及的变量较多，程序可读性较差及debug较难。而且在输出结果的时候要一直向上寻找答案，因此不能用队列来实现，只能用数组，所浪费的内存较多。

第一个改进为将结构体优化为两个int(up, down)和一个string(op)。一共占用8 + n Bytes。这个结构体在输出结果时较为方便，直接将op输出即可，因此可以使用队列来减少内存开销。但这个方法速度依然较慢，因为在进行A，B，C操作时需要获得各个数位上的值，因此要进行一定量的算数运算，有时间损失。进行康托展开时也是如此。

第二个改进将结构体变为一个short[8]数组和一个string(op)，共占用16 + n Bytes。虽然占用的空间有所加大，但进行各项操作和康托展开时所用的时间也大幅减少，因为能很快的获得各个位上的数值。其实，还可以进行进一步的优化，将short改用为char，这时只需要在康托展开时将char转为对应的数值即可，共占用8 + n Bytes。

还可以在循环中进行优化，可以在得出新的节点后便判断是否为目标节点，可以减少1次循环。

总的时间复杂度为O(2^(n + 1))，空间复杂度为O(2^(n + 1) \* n)。