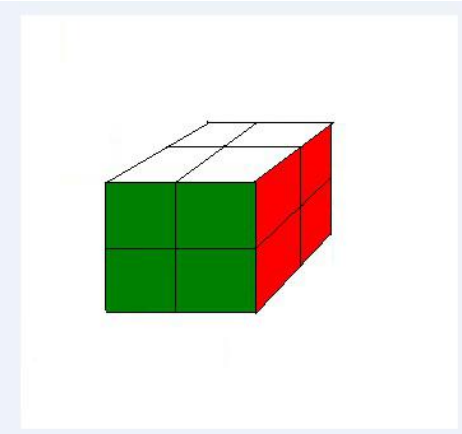
二阶魔方（Pocket Cube）算法

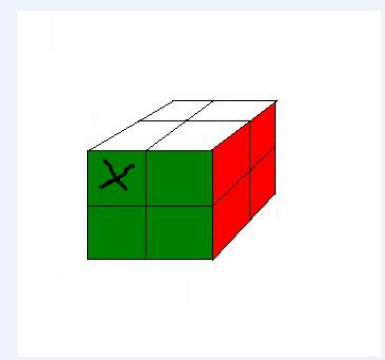
1 代码

1.1 数据结构

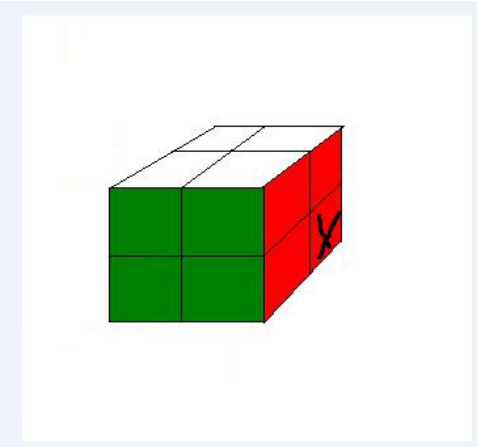


我们先定义颜色，0-5 分别表示橙蓝红绿黄白六种颜色。二维数组 cube[6][4] 用来存储 8 个方块共 24 面的颜色。其中第一个维度用来表示 6 个面，0-5 分别为前、右、后、左、上、下，第二个维度表示该面四个小块，按照从左到右从上到下依次为 0、1、2、3。

例如，下图中打叉的面可表示为 cube[0][0]=3。



下图中打叉的面可表示为 cube[1][3]=2。

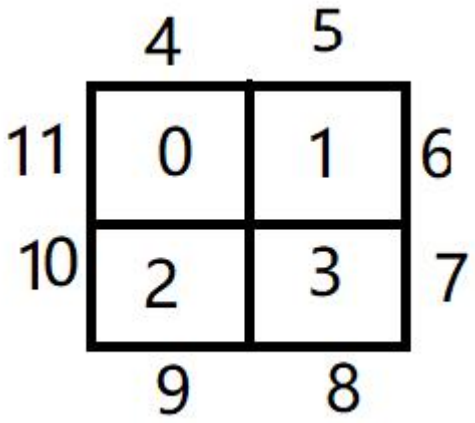


1.2 函数

int main()//主函数
int InitCube()//初始化函数，新建四维数组并从键盘输入魔方的初始状态。
int FirstStep()//还原魔方的第一步
int SecondStep()//还原魔方的第二步
int ThirdStep()//还原魔方的第三步
int Rotate(int direction)//对魔方进行旋转操作，direction 为 0、1、2、3 分别表示左、上、右、下
int PrintCube()//将魔方当前的状态打印出来
int JudgeSolve()//检验魔方是否已经复原

1.3 具体方法

FirstStep()//还原魔方的第一步
第一步的函数有两种实现方法。第一种方法是在找到白色块之后先将整体结构都统一转换为白色处于 cube[5][0]的位置。这样我们能统一使用较为简单的旋转函数。第二个思路是找到之后 cube[][]不用转换处理，而是在旋转函数下功夫，使其能够适应较为复杂的旋转情况。如果使用第一种方法，事实上本身在转换为固定位置时仍是要使用较为复杂的旋转函数。因此我们选择使用第一种方法，主动增加旋转函数的扩展性而不仅仅是只有“左上右下”功能。
int Rotate(int direction)//旋转函数
二阶魔方的旋转总共有几种呢？这值得我们深思。既要全面，还有不能有重复的。经过考虑，总共有 12 种，上层做顺时针旋转 1-3 步、下层做顺时针旋转 1-3 步（注意不要认为这两种情况有重复的可能，因为我们还要认为第一步采用转换方法。）、左层做顺时针旋转 1-3 步、右层做顺时针旋转 1-3 步。总共有 12 种情况。那么我们具体在写函数的时候有没有什么好的办法减少重复度呢？
我们先用图片演示旋转 1-3 步的情况。



若顺时针旋转 1、2、3 步的情况如下表所示。

初始色块	旋转 1 次	旋转 2 次	旋转 3 次
0	2	3	1
1	0	2	3
2	3	1	0
3	1	0	2
4	10	8	6
5	11	9	7
6	4	10	8
7	5	11	9
8	6	4	10
9	7	5	11
10	8	6	4
11	9	7	5

（注：上表指的是旋转之后该位置的色块变化情况，如旋转一次则之前位置 2 的色块跑到了 0，而不是位置 0 的色块跑到了 2）

这样我们可以将旋转函数的结构复杂度降低了。

具体方法：def Rotate(direction,step)。其中 direction 可以取 0、1、2、3 分别代表上层顺时针旋转、下层顺时针旋转、左层顺时针旋转、右层顺时针旋转，然后 step 取 1、2、3。

若想顺时针旋转上层，则 direction 取 0，上图中的初始色块 0-11 分别为 cube[4][0-3]、cube[2][1]、cube[2][0]、cube[1][1]、cube[1][0]、cube[0][1]、cube[0][0]、cube[3][1]、cube[3][0]。

若想顺时针旋转下层，则 direction 取 1，上图中的初始色块 0-11 分别为 cube[5][0-3]、cube[0][2]、cube[0][3]、cube[1][2]、cube[1][3]、cube[2][2]、cube[2][3]、cube[3][3]、cube[3][2]。

若想顺时针旋转左层，则 direction 取 2，上图中的初始色块 0-11 分别为 cube[3][0-3]、cube[4][0]、cube[4][2]、cube[0][0]、cube[0][2]、cube[5][0]、cube[5][2]、cube[2][3]、cube[2][1]。

若想顺时针旋转右层，则 direction 取 3，上图中的初始色块 0-11 分别为 cube[1][0-3]、cube[4][3]、cube[4][1]、cube[2][0]、cube[2][2]、cube[5][3]、cube[5][1]、cube[0][3]、cube[0][1]。

随后我们再写一个子函数 subRotate(cube[11],step)用来把这些初始块和 direction 作为参数传过去，从而直接利用上表进行转换，大大减少代码重复度。

上段想法是好的，但是考虑到 python 没有地址指针或引用的语法，因此我们还是直接使用最原始的本方法，对这 12 种旋转情况都一个一个写出来。这样就完成了我们的 Rotate 函数。

写完 Rotate 函数之后，就来到了 FirstStep 函数。思路如下。先随便找到一个白色的块并将其当作固定块，实际方法是要将其放在 cube[5][0]的位置，然后执行后续操作，那么在代码实现中应该怎么做呢，就回到了我们前面提到过的两种思路，要么就是直接把该块放到 cube[5][0]在执行后续操作要么就直接使用复杂的旋转操作(见 1.3 第一段)。经过仔细考量前者更为简单而且更为尊重解魔方算法的流程，因此我们选择这样写 FirstStep():先找到一个白色块后将其移动到 cube[5][0]位置，具体怎么做？白色块位置有 24 个种情况，能否直接将这 24 种情况都写出来用 if 语句判断呢？事实上 24 个情况虽然多但是每个情况只需要旋转一两步不算麻烦，因此列举如下。(左层顺时针旋转一步简写为左顺 1，其他略)

cube[0][0]	左顺 1	Rotate(2,1)
cube[0][1]	右顺 3，下顺 3	Rotate(3,3)Rotate(1,3)
cube[0][2]	左顺 1，下顺 1	Rotate(2,1)Rotate(1,1)
cube[0][3]	右顺 3，下顺 2	Rotate(3,3)Rotate(1,2)
cube[1][0]	上顺 1，左顺 1	Rotate(0,1)Rotate(2,1)
cube[1][1]	上顺 1，右顺 3，下顺 3	Rotate(0,1)Rotate(3,3)Rotate(1,3)
cube[1][2]	下顺 1，左顺 1，下顺 1	Rotate(1,1)Rotate(2,1)Rotate(1,1)
cube[1][3]	下顺 1，右顺 3，下顺 2	Rotate(1,1)Rotate(3,3)Rotate(1,2)
cube[2][0]	上顺 2，左顺 1	Rotate(0,2)Rotate(2,1)
cube[2][1]	上顺 2，右顺 3，下顺 3	Rotate(0,2)Rotate(3,3)Rotate(1,3)
cube[2][2]	下顺 2，左顺 1，下顺 1	Rotate(1,2)Rotate(2,1)Rotate(1,1)
cube[2][2]	下顺 2，右顺 3，下顺 2	Rotate(1,2)Rotate(3,3)Rotate(1,2)
cube[3][0]	上顺 3，左顺 1	Rotate(0,3)Rotate(2,1)
cube[3][1]	上顺 3，右顺 3，下顺 3	Rotate(0,3)Rotate(3,3)Rotate(1,3)
cube[3][2]	下顺 3，左顺 1，下顺 1	Rotate(1,3)Rotate(2,1)Rotate(1,1)
cube[3][3]	下顺 3，右顺 3，下顺 2	Rotate(1,3)Rotate(3,3)Rotate(1,2)
cube[4][0]	左顺 2	Rotate(2,2)
cube[4][1]	右顺 2，下顺 3	Rotate(3,2)Rotate(1,3)
cube[4][2]	左顺 2，下顺 1	Rotate(2,2)Rotate(1,1)
cube[4][3]	右顺 2，下顺 2	Rotate()
cube[5][0]		
cube[5][1]		
cube[5][2]		

cube[5][3]		