MATH 495R, HOMEWORK 14
SOLVING THE RUBIK'S CUBE: PART I

The set of all possible "moves" of the Rubik's cube is a good example of a group. Each possible move can be constructed as a product of moves consisting of rotating a single face of the Rubik's cube through a 90° rotation, either clockwise or counterclockwise. We will use the following notation for these moves:

One face of the Rubik's cube will be specified as the "Up" (top) face, and another as the "Front" face. These specifications will fix a "Down" (bottom) face (the face opposite the top face), a "Back" face (opposite the front face), and a "Right" and "Left" face (on the right or left as you face the front face, with the top face on the top). We will use the following letters to denote the different faces:

U	Top
F	Front
R	Right
L	Left
B	Back
D	Down

Representing a turn of a face will be indicated by giving the letter of the face that we wish to turn, in upper case for a clockwise turn (as we look directly at the face), and in lower case for a counterclockwise turn. The turns will be listed in order from left to right in the order in which we perform them. For instance, the sequence `RFrf` would correspond to the following:

(1) Turn the right face 90 degrees clockwise.
(2) Turn the front face 90 degrees clockwise.
(3) Turn the right face 90 degrees counterclockwise.
(4) Turn the front face 90 degrees counterclockwise.

To keep the programming easier, we will not allow ourselves to turn the entire cube (in particular the squares in the center of each face will never change position). All move that we allow will be moves of individual faces.

We will number the smaller faces of the Rubik's cube as indicated on the handout at

`https://math.byu.edu/~doud/Math495R/Rubik-handout.pdf`.

With this numbering, you should check that a turning the right fact 90 degrees clockwise (the move denoted by `R`) would move the small face in spot 2 to spot 18, the face in spot 18 to spot 23, the face in 23 to spot 7, and the face in spot 7 to spot 2. The move `R` will actually involve five such 4-cycles: we have that as a permutation of the faces,

$$\texttt{R} = (2, 18, 23, 7)(3, 10, 22, 15)(6, 14, 19, 11)(26, 37, 46, 35)(30, 34, 42, 38)$$

We will define a variable `cube` that keeps track of the position of each sub-face as we manipulate the cube. When the cube is solved, `cube` will be a vector of length 49, with the ith entry equal to i (note that there are only 48 subfaces that can move, but in order to keep the numbering simple, we will leave the zero entry always equal to 0).

1. Write a function `apply(move,cube)` that takes a vector representing a movement of the cube (written as a product of cycles as above) and a vector representing the cube (as described above), and performs the indicated move. It should actually change the values of the vector `cube`, so that it does not actually need to return a value.

   In addition, find the cycle structures for the moves `R`, `L`, `U`, `D`, `F`, and `B`. As an example, the cycle representing `R` would be

   `[[2,18,23,7],[3,10,22,15],[6,14,19,11],[26,37,46,35],[30,34,42,38]]`.

   Store these cycle structures in variables named `Right`, `Left`, `Up`, `Down`, `Front`, and `Back`.
2. Write a function `execute(moves,cube)` that takes a string containing moves of the cube, and a position of the cube, and executes those moves on the cube.

   The function should apply all of the indicated moves to the cube, actually changing the values of the vector. For instance, for a capital letter `R`, it should apply the permutation `Right` that you found above. For a lowercase `r`, it should apply the permutation `Right` three times (since a 90° counterclockwise move is the same as three 90° clockwise moves). After applying all moves specified in the string, the function should then return the sequence of moves that it made. If `moves` contains characters other than "`FBRLUDfbrlud`", these characters should be ignored, and not included in the returned variable.

   After this function is written and debugged, you should not change `cube` other than through this function (except to debug your code).

   In order to test the `execute` function, use it to find the orders of the following sequences of moves: `'RF'` should have order 105 (i.e., applying the move `RF` 105 times should return the cube to its initial position, but fewer than 105 times should not), `Ur` should have order 63, `rdL` should have order 180, and `RDFLBUrdflbu` should have order 360. Note that this last uses all twelve possible moves, so it should be a good test for your function.
3. Write functions: `align25(cube)`, `align26(cube)`, `align27(cube)`, `align28(cube)`, that put the subfaces numbered 25, 26, 27, and 28 in their correct positions. Each one should perform its task without changing the positions of the subfaces dealt with in earlier functions. Each function should return the sequence of moves that it makes.

   Here is my suggestion of how to write these functions.

   For `align25`, create a vector of 24 sequences of moves. The zero entry of this vector would be the sequence of moves necessary to move subface 25 from position 25 to position 25; the one entry of this vector would be the sequence of moves necessary to move subface 25 from position 26 to position 25, and so on. Find the current position of subface 25, lookup the correct sequence of moves in the vector, and execute the moves.

   For `align25`, here is what my function looks like:

```
def align25(cube):
    i=cube.index(25)
    move=['25','u26','uu27','U28','BLU29','RB30','uRB31','lb32',...][i-25]
    return execute(move,cube)
```

   In order to help me keep track of which entry of the vector is which, I have included the starting position of subface 25 (so, for instance, "`Bru41`" would indicate the sequence of moves needed to move the subface in position 41 to position 25). If the `execute` function is correctly written, these extra digits will be ignored when the move is executed.

   The functions for `align26`, `align27`, `align28` can be constructed similarly.

4. Create four functions, `align1(cube)`, `align2(cube)`, `align3(cube)`, and `align4(cube)`, that return sequences of moves that put subfaces 1 through 4 in their correct positions without disturbing subfaces 25 to 28. These functions should work similarly to the ones written in part 4.
5. Create a function `solvetop(cube)` that has a scrambled cube as input, and returns a sequence of moves that solves the top level of the cube. The easiest way to do this would be to run each of the functions `align25(cube)`, `align26(cube)`, `align27(cube)`, `align28(cube)`, `align1(cube)`, `align2(cube)`, `align3(cube)`, and `align4(cube)` in the indicated order; put together their output into a single string, and return that string. Done efficiently, this can be a one or two line function.

You should test your `solvetop` program using some of the 100 scrambled cubes that I have placed online at

`https://math.byu.edu/~doud/Math495R/Cubes.txt`

If your routines work properly, subfaces 1-12 and 24-32 should all be put into the correct positions by `solvetop(cube)`.

If you need to see a graphical simulation of a cube, I have placed one online at

`https://math.byu.edu/~doud/RubiksCube/`

Enter a sequence of moves into the textbox at the top of the page, click the button labeled "Turn", and the program will execute the specified moves.