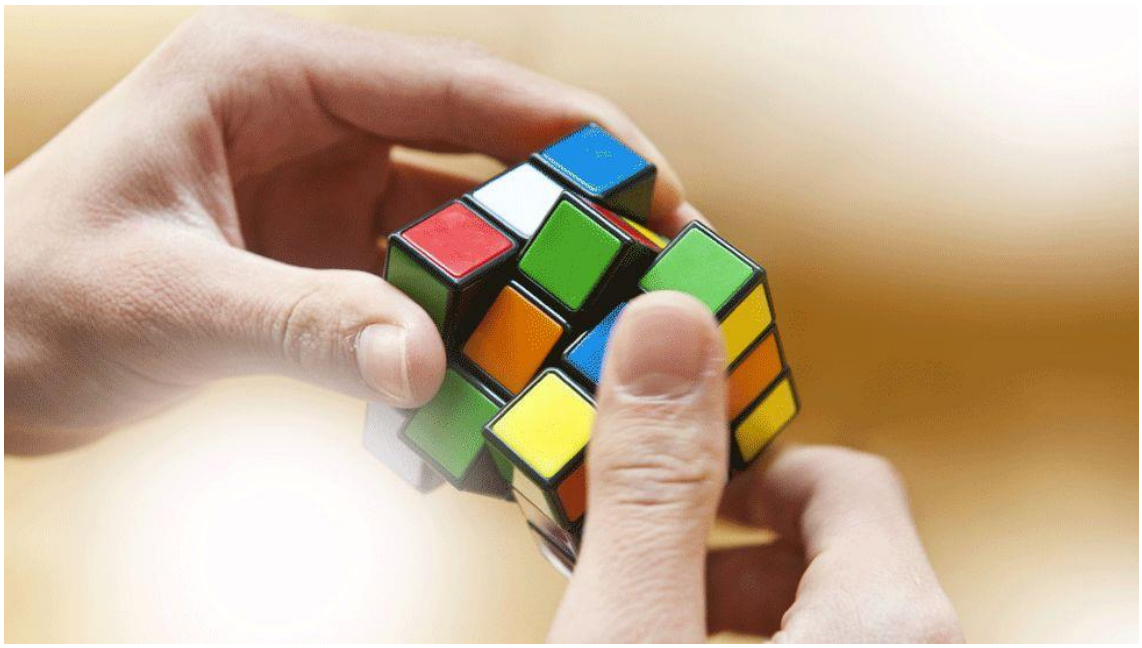
Artificial Intelligence

Rubik's Cube (10pts)

Part I: State Representation and Move Generation

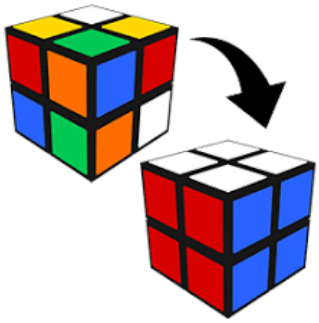
Rubik's Cube is a classic 3-Dimensional combination puzzle created in 1974 by inventor Erno Rubik. The standard version consists of a $3 \times 3 \times 3$ cube. (https://ruwix.com/online-puzzle-simulators/?d=3)

The cube looks like this:



There are six faces that can be rotated 90 degrees in either direction. The goal is to rearrange the colors to make each face monochromatic by rotating slices. Here is a funny video that justifies our intention of solving the Rubik's Cube in this assignment: https://youtu.be/z-SoPwIwgkY. ;)

One important fact about Rubik's Cube is that it has a large **state space**, with approximately 4.3×10^{19} different possible configurations for a $3 \times 3 \times 3$ cube. For comparison, an 8-puzzle state space has 362,880 different configurations of the eight tiles and blank space, which is a much smaller number. For this reason and for the sake of practicality, we tackle a $2 \times 2 \times 2$ Rubik's Cube in this assignment which has around 3,674,160 possible states. (which is still a large number)



In this part, we will write the code needed to represent a single $2 \times 2 \times 2$ Rubik's Cube state and to compute the possible following cube states after moving a slice. In the future part, we will extend this code to search through a space of cubes to find solutions to a given problem. The code for this assignment should be written in Python 3 to run on tux.cs.drexel.edu.

Implementation Setup

The various parts of this assignment will require a shell script that can pass an argument to your code—thus allowing you to use **Python3** while allowing us to be able to run your code with a consistent interface. However, **you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT use any external libraries or packages like Numpy** (if you have any questions at all about what is allowable, please email the instructor).
To this end, please create a shell script **run.sh** that calls your code and includes 2 command-line arguments that are passed to your code. For example, a shell script for Python3 might look as follows:

```
#!/bin/sh
if [ "$#" -eq 4 ]; then
    python3 RubiksCube.py "$1" "$2" "$3" "$4"
elif [ "$#" -eq 3 ]; then
    python3 RubiksCube.py "$1" "$2" "$3"
elif [ "$#" -eq 2 ]; then
    python3 RubiksCube.py "$1" "$2"
else
    python3 RubiksCube.py "$1"
fi
```

Your code (in our example above, the Python code in **RubiksCube.py**) will need to accept these two arguments and use them properly for each particular command. As you'll see in the sections below, running the code will have the general format:

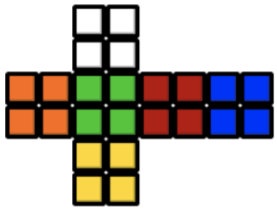
```
sh run.sh <command> [<optional-argument>]
```

Again, this scheme will allow you to test your code thoroughly and also allow us to test your code using a variety of arguments.

State Representation (1pts)

For this assignment, you first need to create a representation of the state of the cube. Let's assume that we can represent the cube as a two-dimensional representation, which we can print like this:

```
   WW
   WW
OO GG RR BB
OO GG RR BB
   YY
   YY
```



Each letter represents a color, and each 2×2 square represents a face in the cube. The goal is to have the same colors on all faces.

Write a class **Cube** that takes a string representation for a cube with the **proper size**. We will assume that the input string will have the following format: (make sure to export errors in case the input string is wrong)

```
"WWWW RRRR GGGG YYYY OOOO BBBB"
```

The indexing order that maps this string to the 2-dimensional representation of the cube is as follows:

```
         0  1
         2  3
16 17    8  9    4 5   20 21
18 19   10 11    6 7   22 23
        12 13
        14 15
```

You should also implement a **print()** function that can print the state in ASCII characters. It should be runnable from the command line with the "**print**" command as the first argument and the cube as the second argument. If the state representation argument is not provided here, please use the state above as the **default**. Here are two examples of running from the command line:

```
> sh run.sh print


   WW
   WW
OO GG RR BB
OO GG RR BB
   YY
   YY

> sh run.sh print "RWOR GOYB BOGW BRBW YWYO RGYG"

   RW
   OR
YW BO GO RG
YO GW YB YG
   BR
   BW
```

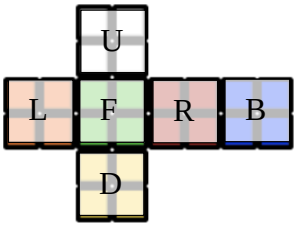
Identifying Solutions (1pts)

Write a method that determines whether the given cube is at a goal state. This should be very easy: if all the faces have the same color, then you have reached the solution state. Then, augment the possible commands to accept a "**goal**" command that prints "**True**" or "**False**" depending on whether the given state is at the solution state or not. For example:

```
> sh run.sh goal "RWOR GOYB BOGW BRBW YWYO RGYG"
False
> sh run.sh goal "WWWW RRRR GGGG YYYY OOOO BBBB"
True
```

Move generation (2pts)

The notation used for Rubik's Cube moves is **F**, **R**, **U**, **B**, **L**, and **D** capital letters. Each of these letters marks one 90-degree clockwise rotation of one face of the cube – in the order **F**ront, **R**ight, **U**p, **B**ack, **L**eft, and **D**own (below figure). And if the letter is followed by an apostrophe, that means a counterclockwise rotation. (for seeing a visual simulator of the moves, visit https://ruwix.com/online-puzzle-simulators/2x2x2-pocket-cube-simulator.php)



For rule (move) generation, implement a function **'applyMove'** that, given a state and a move, performs the move in the state. After that, implement a function **'applyMovesStr'** that, given a state and a string sequence of moves, returns a new state, resulting from cloning the state and then applying the sequence of moves to the cloned state.

Augment the possible commands to accept an "**applyMovesStr**" command that prints the resulting state after applying the sequence of moves. Again, if the state representation argument is not provided, please use the **default state** as the default **("WWWW RRRR GGGG YYYY OOOO BBBB")**. For example:

```
> sh run.sh applyMovesStr "R U' R'" "WWWW RRRR GGGG YYYY OOOO BBBB"

   GW
   WR
WB OG YR BR
OO GW GR BB
   YO
   YY
```

Important note: each move described above is actually a permutation of the indices of the colors. For example, after applying **U,** the position of the colors will change as below:

```
[2,0,3,1,20,21,6,7,4,5,10,11,12,13,14,15,8,9,18,19,16,17,22,23]
```

That is, the colors originally in positions `2,0,3,1` are moved to positions `0,1,2,3`; the colors originally in positions `20,21` are moved to positions `4,5`; the colors originally in positions `6,7` remain in positions `6,7`, etc. For your convenience, we have provided a program skeleton containing the permutation for each move.

State Comparison

Write a function that compares two states and returns `True` if they are identical and `False` if they are not. Do so using the simplest possible approach: just iterate over each position in the matrix that represents the state and compare the integers one by one. If they are all identical, the states are identical, otherwise, they are not. You do not have to have a shell script for this part.

Shuffling (Scrambling) (2pts)

One last function that we have to implement is a shuffling function. Remember that if we randomly assign colors to each index, there is a high chance that the cube would not be solvable. To scramble, we should start from the default cube **("WWWW RRRR GGGG YYYY OOOO BBBB")** and shuffle the cube with picking random moves. Write a function "**shuffle**" that returns a shuffled cube for n number of times by random moves.

Augment the possible commands to accept a "**shuffle**" command that prints n random moves and applies them on the default state. Here is an example: (make sure that your output is similar)

```
> sh run.sh shuffle 10

R' R F D' B R' B' R R F

   WG
   GY
OR WG OW OB
RW BG RY BY
   RY
   BO
```

Random Walk (4pts)

Write a method that does a random walk. This method must take in a string of moves, and a number, *N*. The input sequence of moves is to be used to permute the default cube state. The **default state** is the solved cube with the state string of **"WWWW RRRR GGGG YYYY OOOO BBBB"**. Once you have used the input sequence to permute the default state, the shuffled state (not to be confused with the shuffle function) becomes the **initial state** of the cube. From there, you randomly select *N* moves from the set of all possible moves and stop after *N* moves, or once the cube has been solved. If it has not been solved, you count that cycle as a single **iteration**, reset the cube's state to the initial state (the permuted state), and try again. If it has been solved, print the number of iterations in order to solve it. You should be timing this process; stop the iteration of the time window.

More specifically, add a "**random**" command-line command that receives a positive integer, *N*, a sequence of moves, and a time limit in seconds, *T*:

- apply the sequence of moves to the default state and create an initial state,
- select one of the moves at random,
- execute that move,
- and stop if we've reached the goal or we've already executed *N* moves, otherwise, go to the initial state and repeat if the running time limit, *T*, has not been reached.

The random function should print the following:

- If the method has found a solution, the resulting sequence of moves and relative states to solve the initial state (if no solution was found in the time limit, print "No solution in the time limit!!"),
- the number of iterations,
- and the time that the function took to find the solution.

Notice that your algorithm should at least (if not better) return the opposite of the moves you applied to the cube. And if you set *N* with the number of sequences of moves applied to the default state, you will have a better chance. Here are two examples:

```
> sh run.sh random "L D' R' F R D'" 6 10
U B' U' F L F'
   BW            GB            WR
   GW            WW            WW
OY RR BB OY  RR BB OY OY  RR BB OG YG
WR BY OO WG  WR BY OO WG  GR BY OB OW
   YG            YG            YG
   RG            RG            OY

   RW            RW            WW
   WW            RG            GG
YG RR BB OG  YY BR WB OG  GY RR WB OO
GR BY OB OW  GG YR WB OW  GY RR WB OO
   YG            OB            BB
   OY            OY            YY

   WW
   WW
GG RR BB OO
GG RR BB OO
   YY
   YY

916314
5.33

> sh run.sh random "L' B' U' D L' F B" 7 10
F' F' D D F' B B
```

```
   GG            GG            GG
   BB            YG            YY
WW RR YY RR  WB RO YY RR  WG OO BY RR
BB OO GG OO  BB RO YG OO  BY RR WG OO
   YY            WB            BB
   WW            WW            WW

   GG            GG            GG
   YY            YY            BB
WG OO BY RR  WG OO BY RR  WY OO WY RR
OO BY RR WG  WG OO BY RR  WY OO WY RR
   WB            WW            GG
   WB            BB            BB

   YY            BB
   BB            BB
GY OO WB RR  YY OO WW RR
GY OO WB RR  YY OO WW RR
   GG            GG
   WW            GG

11704
0.07
```

Notes:

- Because this is a random walk, the code will do different things for each run. Try to play with the number N.
- Again the default state is the solved cube with a state string of **"WWWW RRRR GGGG YYYY OOOO BBBB"** and the initial state is the shuffled state after applying the given string of moves.

Academic Honesty

Please remember that you must write all the code for this (and all) assignments by yourself, on your own, without help from anyone except the course TA or instructor.

Submission

Remember that your code must run on **tux.cs.drexel.edu**—that's where we will run the code for testing and grading purposes. Code that doesn't compile or run there will receive a grade of zero.

For this assignment, you must submit the following:

- Your Python code for this assignment.
- Your **run.sh** shell script that can be run as noted in the examples.
- We should be able to run all the functions that mentioned above (**print**, **goal**, **applyMovesStr**, **shuffle**, and **random**) using your **run.sh** shell script.
- A PDF document with written documentation containing a few paragraphs explaining your program and results showing testing of your routines. If you did anything extra, discuss it here.

Please use a compression utility to compress your files into a single ZIP file (NOT RAR, nor any other compression format). The final ZIP file must be submitted electronically using Blackboard—do not email your assignment to a TA or instructor! If you are having difficulty with your Blackboard account, you are responsible for resolving these problems with a TA or someone from IRT before the assignment is due. If you have any doubts, complete your work early so that someone can help you.