```
PUZZLE PROJECT              Fri Apr  1 05:00:16 EDT 2016


    rubiks
        Solve it fast!
        Boston Algorithm Festival April 2016

        rubiks.txt              Project Description
        rubiks.h                Header File for C/C++

    rubiks-io
        First Stage (IO) of puzzle project.

    rubiks-rotation
        Second Stage (Rotations) of puzzle project.

    rubiks-search
        Third Stage (Search) of puzzle project.

    rubiks-part-search
        Fourth Stage (Partial Search) of puzzle project.

    rubiks-mitm
        Fifth Stage (MITM) of puzzle project.

    rubiks-part-mitm
        Sixth Stage (Partial MITM) of puzzle project.
```

FIFTH DRAFT

Rubik’s Cube
------- ----

The Rubik’s Cube puzzle was created by Erno Rubik in the mid-1970’s. The puzzle consists of 27 small cubes, called ‘cubelets’ organized in a 3x3x3 array that is the larger ‘Rubik’s Cube’. The large cube has 6 faces, and the 9 cubelets on each face can be rotated all together by 90, 180, or 270 degrees. There are 6x3 = 18 such rotations.

The exposed faces of the cubelets are each colored with one of 6 colors. In the ‘solved position’ all the cubelet faces that share the same large cube face have the same color: white, green, red, blue, orange, or yellow. After many random face rotations the cubelets are scrambled, and the puzzle is to put the large cube back in ‘solved position’. A sequence of rotations that puts the cube back into ‘solved position’ is a solution to the puzzle.

See the discussion at

      en.wikipedia.org/wiki/Rubik%27s_Cube

for pictures.

An exploded view of the solved Rubik’s Cube is

```
       WWW              W = white
       WWW              G = Green
       WWW              R = Red
    GGGRRRBBBOOO        B = Blue
    GGGRRRBBBOOO        O = Orange
    GGGRRRBBBOOO        Y = Yellow
       YYY
       YYY
       YYY
```

Rotating the red face clockwise 90 degrees gives

```
       WWW
       WWW
       GGG
    GGYRRRWBBOOO
    GGYRRRWBBOOO
    GGYRRRWBBOOO
       BBB
       YYY
       YYY
```

It is easy to write a computer program to solve this puzzle if a large number of rotations is permitted in the solution, but until the last few years no reasonably fast computer program was known that could find a minimal solution: one with the fewest rotations. In 2010 it was proved that a minimal solution would have at most 20 rotations.

In the series of problems described below we develop code to solve the following Rubik’s Cube problem for successively larger numbers of rotations, N.

You have a Rubik’s Cube in solved position. You hand it to a ‘tester’ who turns his back on you and applies *exactly* N random rotations to the cube, but without rotating any face twice in a row. The tester then hands the cube back to you, and you must find a solution with *exactly* N rotations where no face is rotated twice in succession.

First we give overviews of the three algorithms we use.

Exhaustive Search Overview

One way to solve this problem is to try all sequences of N rotations that might be solutions, and check whether each is in fact a solution. This is called 'exhaustive search', and is in fact the basis for the more sophisticated algorithms. There are 18 possible first rotations, but only 15 possible second moves because of the restriction that no two consecutive rotations rotate the same face. Similarly there are only 15 possible third, forth, fifth, etc. rotations. In all there are 18*(15^(N-1)) rotation sequences that must be checked to see if they are solutions. We have

```
                 18*(15^(N-1)) = 18 for N == 1
                                270 for N == 2
                              4,050 for N == 3
                             60,750 for N == 4
                            911,250 for N == 5
                         13,668,750 for N == 6
                        205,031,250 for N == 7
                      3,075,468,750 for N == 8
                     46,132,031,250 for N == 9
                    691,980,468,750 for N == 10
                 10,379,707,031,250 for N == 11
                155,695,605,468,750 for N == 12
              2,335,434,082,031,250 for N == 13
             35,031,511,230,468,750 for N == 14
            525,472,668,457,031,250 for N == 15
          7,882,090,026,855,468,750 for N == 16
        118,231,350,402,832,031,250 for N == 17
      1,773,470,256,042,480,468,750 for N == 18
     26,602,053,840,637,207,031,250 for N == 19
    399,030,807,609,558,105,468,750 for N == 20
```

For N == 6 exhaustive search takes 0.4 seconds on a high speed laptop. For every +1 increment of N the search will take 15 times as long, and in fact it takes 6 seconds for N == 7 and 90 seconds for N == 8 on the same laptop.

If we let ROT(N) = 18*(15^(N-1)) exhaustive search takes about

```
                30 * ROT(N) nanoseconds
```

on the laptop. For N == 10 this is 20,760 seconds = 5.8 hours (but has not been tested).

This speed is very, very impressive. Each of the ROT(N) final states requires for its computation 20 iterations of a 9 instruction loop that includes 2 reads of global variables, giving an execution rate of 6 instructions per nanosecond and 0.75 nanoseconds per global variable read (the loop is NOT unraveled). And there are some additional computations done within the 30 nanoseconds, so execution speed must be even faster than this. The laptop does have a 3 megabyte cache, so everything likely fits in this on-chip processor cache.

Meet in the Middle (MITM) Overview

The next most sophisticated algorithm is 'meet in the middle' which puts together two exhaustive searches of size Nr and Ns to get a search of size N == NR + Ns. For example, we can put together exhaustive searches of size Nr == 6 and Nr == 7 to make a search of size N == Nr + Ns == 13.

To do this, we store the rotation sequences and the states they produce when Nr rotations are applied to the initial unsolved state, which for Nr == 6 means we store 13,668,750 records of 26 bytes each. These are stored in a hash table with the produced state as key. Then we run a search that applies Ns rotation inverses to the goal state and looks up the states produced in the hash table, which for Ns == 7 means we look up 205,031,250 states. Each successful lookup match produces a solution to the N = Nr + Ns problem (there is a more detailed description below).

So the time for N == 13 == Nr + Ns == 6 + 7 is the time of an exhaustive search for N == 6 plus the time for an exhaustive search for N == 7. Almost, but not quite. In fact the Nr == 6 search takes 2.0 seconds instead of the 0.4 for exhaustive search, and Nr == 7 takes 36 seconds instead of 6, giving

```
        146 * ROT(Nr) nanoseconds
        176 * ROT(Ns) nanoseconds
```

After some testing, our best guess is that the extra time is taken by cache misses to large memory. Each Nr record requires one access to a large memory to fetch a hash bucket head, and this likely takes 100 nanoseconds. Each Ns lookup also requires a fetch of the hash bucket head. We have 4 times as many hash buckets as records, so 75% of the buckets are empty, and a single bucket head reference is all that is required 75% of the time. But 25% of the time a record must be fetched, adding an average of 25% * 100 nanoseconds. These 100 nanosecond cache fetches seem to account for the extra time required by MITM searches over exhaustive searches.

The solution we tested has 4 hash buckets per actual record, taking for each Nr == 6 record

```
         4 bytes for a next record pointer
        20 bytes for cubelet positions
         6 bytes for rotation indices
        16 = 4 x 4 bytes for hash bucket heads
        --
        46 bytes
```

or

```
        46 * ROT(Nr) bytes
```

which equals 629 megabytes for Nr == 6. This goes up by a factor of 15 for each +1 increment of Nr, and would be 9.636 gigabytes for N == 7, which would cause swapping on our laptop.

However N == 16 == Nr + Ns == 6 + 10 would be feasible and would take 629 gigabytes and

```
      146 * ROT(6) nanoseconds          2 seconds
    + 176 * ROT(10) nanoseconds         121,788 seconds
                                        ---------------
                                        34 hours
```

(this has not been tested).

For N == 20, Nr == Ns == 10, the execution time would be around 60 hours, but the memory requirement (at 50 bytes per record, including 4 bucket heads) is 35 terabytes, rather unfeasible since the memory must be high speed RAM in order for the hash table lookups to be fast.

Dissection Overview
---------- --------

[NOTICE: For this FIFTH DRAFT this section is based solely on analysis and not on running code, as the algorithm has not been coded by us yet.]

The ‘dissection’ algorithm is due to Dinur, Dunkelman, Keller, and Shamir, Communications of the ACM, Oct 2014.

The ‘dissection’ algorithm exploits the fact that the motion of each cubelet can be considered independently of the motion of other cubelets.

The top level idea is to iterate over all possible intermediate states of M cubelets. Let Sk denote the state of the M cubelets after k rotations, so if N == 20, S0 is the initial unsolved state of the cubelets and S20 the final solved state. We then want to iterate over all 24^M possible values of the intermediate state S10.

So pick a value of S10. Find all ‘initial’ sequences of 10 rotations that take S0 to S10, and then all ‘final sequences of 10 rotations that take S10 to the solved state S20, and combine every initial sequence with every final sequence and test which combinations move the entire Rubik’s Cube to the solved state.

Finding initial or final sequences of rotations is an N == 10 M-cubelet partial search problem that can be solved by meet-in-the-middle (MITM). We will refer to these N == 10 MITM executions as ‘inner’ MITMs.

Finding the combinations that solve the entire Rubik’s Cube can be done as per MITM; for each initial 10-rotation sequence we store in a hash table the rotation sequence and the state of the entire Rubik’s Cube after applying the sequence to the initial (unsolved) state of the Rubik’s Cube. Then for each final 10-rotation sequence, we find the state of the Rubik’s Cube after the final sequence is applied backward from the solved state, and look up this state in the hash table. We will refer to all this as the ‘outer’ MITM.

Thus for each iteration (each value of S10) we do two inner MITM’s and one outer MITM.

Because each iteration is independent of every other iteration, there are only ROT(10) / 24^M records on average for an outer MITM. Fixing M == 4, this is 2,085,685 records on average, but the number is actually rather variable, and for some S10 values with M == 4 can be up to 100 million. The records are 50 bytes (10 rotations instead of 6) and thus fit in 5 gigabytes of memory.

The time for an MITM is mostly determined by the number of records stored and looked up. The numbers of records for the outer MITM summed over all 24^M iterations is ROT(10). The number for each inner MITM is ROT(5), and there are 2 * 24^M inner MITM’s. So the total number is

```
        ROT(10) + 2 * 24^M * ROT(5)
```

We know from the discussion of pure MITM above that ROT(10) records with Nr == Ns == 10 takes on the order of 60 hours time, or 322 * ROT(10) nanoseconds, and the 322 nanoseconds per record applies whenever Nr == Ns == N/2, so this time is about

```
  322 * ( ROT(10) + 2 * 24^M * ROT(5) ) nanoseconds

  322 * ( 692 + 605 ) seconds = 116 hours if M == 4

  322 * ( 692 + 14,512 ) seconds = 1,360 hours if M == 5
```

So clearly we want M == 4 and will need 5 gigabytes.

But there is no reason why M has to be constant during the entire dissection algorithm. Given an M == 4 inter-mediate value S10 that generates too many records, we can add position values for a fifth cubelet to it in order to replace it with 24 M == 5 intermediate values which partition the records generated by S10 into 24 groups, only one of which needs to be stored at a time. Given that the maximum number of M == 4 records, 100 million, is 50 times the average, 2 million, this stra-tegy should reduce needed memory to under 1 gigabyte with little increase in overall time.

There is yet another contributor to the time required by dissection. For normal MITM, computation of the states stored and looked up is quick because these are computed by exhaustive search which uses a tree-like computation in which only the last rotation needs to be done for every state produced, while the next-to-last is done only once for every 15 states produced, and so forth. For the outer MITM of a dissection this it not true and for each state stored or looked up 10 rotations must be performed. From the exhaustive search timing we can deduce that each rotation takes at most 25 nanoseconds so the total time is

```
  25 * 10 * 2 * ROT(10) nanoseconds = 500 * 692 seconds
                                    = 96 hours
```

Thus the total time for dissection (with 5 gigabytes storage) is

```
        116 + 96 = 212 hours
```

So for N == 20, MITM requires 60 hours and 35 tera-bytes, while dissection requires 212 hours and 5 giga-bytes. Or with the expanding M trick, a bit more time but less than 1 gigabyte.

```
Data Representation
---- --------------

We must represent states of Rubik’s Cube in memory.
What is more, to run the dissection algorithm, we must
represent the state of each cubelet independently of
the state of every other cubelet.

First, let us name the cubelets.

Note that the center of each face does not appear to
move (it rotates but always looks the same).  Thus
we can name each face for the color of its center,
and we have faces W, G, R, B, O, Y.  We can name the
center cubelet of each face by its color, but since
these cubelets do not move, we can ignore them.

Next we have the 12 cubelets that are in the centers of
each edge (NOT on the corners).  Each has two colors
and we name these by their colors: e.g., RW, GR, BR, RY,
etc.  Note RW and WR are the SAME cubelet according to
this scheme.  Note there are NO cubelets named RO, BG,
or YW.

Lastly we have 8 cubelets that are corners.  Each has
three colors and we name them by their colors, writing
these as we see them in CLOCKWISE order when we look at
the cubelet in a way that allows us to see all three
colors.  Thus we have a corner GWR, or equivalently
either RGW or WRG; but we have NO corner named GRW.

We can name the cubelet locations the same way as we
name cubelets: we give the face colors (face center
cubelet colors) that we see when we look at the Rubik’s
Cube in a way that allows us to see all the faces
included in the position.  These names are the same as
the names of the cubelets that would be in the locations
were the Rubik’s Cube in the solved state.

Now we want to name the cubelet positions.  A position
is a location plus an orientation.  For example, the
cubelet BR can be in location GR but it can have its
red side either on the red face or on the green face.
We write BR/GR as the position with the red side on
the red face, and BR/RG as the position with the red
side on the green face.  In general the BR/GR means
B is on the G side and R is on the R side, while BR/RG
means B is on the R side and R is on the G side.  RB/RG
and BR/GR name the same position.

Similarly GWR/BYR means that the GWR corner cubelet is
at location BYR with its G side on the B face, its W
side on the Y face, and its R side on the R face.

In outputting names below we will follow the convention
that we always use the lexically first among equivalent
names, i.e., the one whose first letter is first in the
alphabet.  Thus

    BR          and not the equivalent RB
    GWR         and not the equivalent RGW or WRG
    BR/GY       and not the equivalent RB/YG
    GWR/BYR     and not the equivalent RGW/RBY
                                    or WRG/YRB

We will call a name whose first letter is first in the
alphabet ‘canonical’.
```

A Rubik’s Calculator

You have been asked to program a Rubik’s Calculator that manipulates Rubik’s Cubes and computes solutions.

The first step is input/output. In order for the dissection algorithm to work you need to store the position of each cubelet in a manner independent of the positions of all the other cubelets, so we insist on the method described above of naming cubelets and their positions. However, for human use inputting and outputting exploded views of the Rubik’s Cube is desired.

To make your job simpler, we have coded some subroutines for you. These make input/output of cubes and states easy, but do nothing for rotations or searches, leaving those to you.

The subroutines are available via the ‘rubiks.h’ header file and the rubiks_lib.o object file. The ‘rubiks.h’ file contains documentation for these subroutines.

If you are programming in C, you must include with

```
            # include "rubiks.h"
```

and then compile with

```
            gcc -O2 -o rubiks -std=c99 \
                rubiks-io.c rubiks_lib.o -lstdc++

    -O2 is needed else ‘inline’s are ignored;
    -std=c99 is needed for C99 Standard which includes
             ‘inline’s and FOR macro ‘int’ declaration;
    -lstdc++ is needed by rubiks_lib.o which is actually
             compiled C++ code with a C language inter-
             face
```

If you are programming in C++, you must include ‘rubiks.h’ using

```
    extern "C" {
    # include "rubiks.h
    }
```

and then compile with

```
    g++ -o rubiks rubiks.c rubiks_lib.o
```

In what follows we will use ‘c’ to denote one of the colors W, G, R, B, O, or Y. If you see two ‘c’s near each other, they do NOT NECESSARILY denote the same color. Thus face names have the form ‘c’, middle of edge cubelet names have the form ‘cc’, corner cubelet names have the form ‘ccc’, and the positions of these corner cubelets have names of the form ‘ccc/ccc’.

Internal Representation of Rubik’s Cube

Before your calculator does anything else it should compute some internal tables.

First, there should be a table of 20 cubelets, such that ‘cubelet[i]’ gives the name of the i+1’st cubelet for i = 0 .. 19. Here we are ignoring the face center cubelets. For example, cubelet[0] might be "BRW".

Second, there should be a matrix of 24 positions for each of the 20 cubelets, such that

```
        cubelet[i]/position[i][j]
```

```
gives the name of the j+1'st position of the i+1'st
cubelet for i = 0 .. 19 and j = 0 .. 23.  For example,
if cubelet[0] is "BRW" and position[0][0] is "BRW",
then position j == 0 of cubelet i == 0 is "BRW/BRW",
the solved position of cubelet "BRW".

Continuing this example with cubelet[0] = "BRW" we
might have:

    cubelet[0]/position[0][0] might be "BRW/BRW"
    cubelet[0]/position[0][1] might be "BRW/BYR"
    cubelet[0]/. . . . .
    cubelet[0]/position[0][8] might be "BRW/RWB"
    cubelet[0]/position[0][9] might be "BRW/YRB"
    . . . . .

Here cubelet[0] is the upper right front corner,
position[0][0] is the solved position of this cubelet,
and position[0][9] is the position it will end up in
if it starts in the solved position and a 90 degree
clockwise rotation of the red face is applied.

All this is not as hard as it might seem.  Suppose
you elect to use i = 0 .. 7 for corners.  You can
write down the 8 corner locations BRW, BYR, GRY,
... BWO.  These are the names of the cubelets.
Then their positions are 'ccc/ccc' where the first
ccc is a cubelet name and the second is one of the
three circular rotations of a location name.

A state of the Rubik's Cube is then a vector S of 20
elements, one for each cubelet, with each element being
an integer in the range from 0 through 23 specifying a
cubelet position.  Let 'S[i]' be the current state of
the i+1'st cubelet of Rubik's Cube.  Then

    cubelet[i]/position[i][S[i]]

is the position of cubelet[i] for i = 0 .. 19.

In the following we will store 'S[i]' in one byte (C
or C++ 'char' value or JAVA 'byte' value), so the state
of the Rubik's Cube is stored in 20 bytes.  We could
pack this data, but it is not worth doing so.  We can
also use -1 and -2 to represent 'illegal positions' of a
cubelet (see below).


Input Commands
----- --------

The input command consists of 10 lines with the format

i
   ccc
   cWc
   ccc
cccccccccccc
cGccRccBccOc
cccccccccccc
   ccc
   cYc
   ccc

If any of the face center colors are wrong, the command
outputs:

Error: Illegal face color.
   ccc
   cWc
   ccc
cccccccccccc
cGccRccBccOc
cccccccccccc
   ccc
   cYc
   ccc
```

```
EXCEPT that any illegal face color is replaced by ‘*’.
In this case the command does nothing else.  See the
sample-io.in and sample-io.test files for an example of
this and all the other input/output commands described
in this and the next section.  Note, however, that
sample-io.test is the output of

        QA rubiks < sample-io.in

and NOT ‘rubiks < sample-io.in’: see the QA ‘Quality
Assurance’ program below.

Otherwise the command modifies the calculator’s internal
representation of the Rubik’s Cube.  Each cubelet is
assigned its position, except that some cubelets may
have no legal position, or may have more than one, and
these are marked as being missing or multiple (say they
are given position S[i] = -1 for missing and S[i] = -2
for multiple).  If any cubelet is marked missing or
multiple, this input command outputs:

Error: Illegal configuration.
   ccc
   cWc
   ccc
cccccccccccc
cGccRccBccOc
cccccccccccc
   ccc
   cYc
   ccc

where the ‘c’s are derived from the legal cubelet
positions but are replaced by ‘*’s if there is no
legally positioned cubelet at a location.

Again, see sample-io.in/sample-io.test for examples.

There is one other input command:

ss

This puts the Rubik’s Cube in the solved state.


Output Commands
------ --------

This is just:

o

and outputs the same exploded view of the Rubik’s Cube
as the ‘Error: Illegal configuration.’ message above
output, except that if there are no illegally positioned
cubelets there will be no ‘*’s, and there is no ‘Error:
...’ line in any case.

In addition the following may be used to test the
internal data structures:

cc

Output a line containing the current position of cubelet
‘cc’.

ccc

Output a line containing the current position of cubelet
‘ccc’.

E.g., if the Cube is in solved position, ‘BRW’ would
output a line containing ‘BRW/BRW’.
```

If a cubelet does not have a legal position, these last commands output a line with ‘?’s replacing the colors in the second part of a missing position, and ‘#’s replacing the colors in the second part of a multiple position: e.g. ‘BRW/???’ or ‘BRW/###’.

The Quality Assurance Program

Your problem directory contains a program QA provided by the Quality Assurance Department. It helps you test your program.

Executing the command ‘QA rubiks’ is the same as executing ‘rubiks’ with several differences. First, if input is from a file, as in ‘QA rubiks <sample-io.in’, the input lines of the file will be echoed to the output, which is helpful in figuring out what is going on. Second, if your program makes a mistake in its output, the QA program will output an error message beginning with ‘** ERROR:’.

Also the QA program accepts some extra commands. One of these is

```
random N
```

which generates an input command with input made by performing N random face rotations on the Rubik’s cube starting in solved state, where no two consecutive rotations rotate the same face. This command uses a pseudo-random number generator whose seed MUST be set BEFORE the ‘random’ command is executed by the QA command

```
seed S
```

where S is a strictly positive 9 digit arbitrary integer. The ‘random’ command results depend only upon what the seed was set to and the ‘random’ commands between the current command and the command that reset the seed, so by setting the seed one can get repeatable ‘random’ command results.

To use QA your code MUST flush any output you print. This can be done if you are using C++ streams by ending any line output with

```
        cout << endl
```

If you are using printf in C or C++, you must execute

```
        fflush ( stdout )
```

after outputting ’\n’. If you do NOT do this, the lines you have output will get stuck in an internal buffer because your program output is a pipe to the QA program and not a terminal, and therefore your program thinks it does not have to flush the buffer out when a ’\n’ appears in the buffer. In this case QA will time out waiting for your program output.

You can test your input-output commands using

```
        QA rubiks <sample-io.in
```

The output should be identical to the sample-io.test file.

```
When you do this you will see that QA also accepts com-
mads of the form:

-- ...

i.e., lines beginning with ‘--’, as comment lines, which
are output but otherwise ignored.  In sample-io.in such
lines are used to separate test cases.

QA also accepts and ignores blank lines and lines begin-
ning with ‘**’.  The latter can be used for comments
that do NOT separate test cases.

In addition, if your program outputs lines that begin
with ‘**’ or are blank, QA just prints these and other-
wise ignores them.  So if you want to insert debugging
print statements in your program, have these print lines
that begin with ‘**’ or are blank.

Some of your testing can be automated by using the
command

        make test-io

This does the following:

    (1) Makes a writable copy of sample-io.in named
        test-io.in if the latter does not exist.
    (2) Makes a writable copy of sample-io.test named
        test-io.test if the latter does not exist.
    (3) Symbolically (re)links rubiks.in to
        test-io.in.
    (4) Symbolically (re)links rubiks.test to
        test-io.test.
    (5) Runs ‘make test’, which in turn:
        (5a) Executes ‘make rubiks’ to compile rubiks.
        (5b) Executes
             ‘QA rubiks < rubiks.in > rubiks.out’
             and prints out rubiks.out
        (5c) Computes and outputs the difference listing
             of rubiks.out and rubiks.test

This is set up so if you want to add IO tests you can
edit test-io.in and test-io.test, adding your own
tests at the end, and then rerun ‘make test-io’.  You
first add to test-io.in and rerun ‘make test-io’, and
if the difference listing shows correct additions to
rubiks.out, you ‘cp rubiks.out test-io.test’ (cp is
the UNIX copy command) so the next run of ‘make test-io’
will see no differences.

You can submit your program for further testing by

        make submit-io
```

```
Your program will then be tested by the ‘autojudge’,
which you may think of as representing your Quality
Assurance (QA) department.  The autojudge will have
files similar to sample-io.in and sample-io.test, but
much bigger and containing many test cases.  If for
any test case the output of ‘QA rubiks’ applied to the
test case input does not match the test case output,
the autojudge ‘grades’ your submission ‘Incorrect
Output’ and emails you back the input and output of the
first failed test case.  If you put these in the files
‘failed.in’ and ‘failed.test’ respectively, you will
find that ‘QA rubiks <failed.in’ will not match
‘failed.test’, and you will need to fix this by finding
and fixing a bug in your program.  Or better yet, you
can append the test cases to test-io.in and test-io.test
as per above.

If the autojudge finds no errors, it simply emails the
grade ‘Completely Correct’ to you.


Face Rotations
---- ---------

We now take up the issue of describing face rotations.

The faces are named by colors c, and we will name face
rotations by putting a digit d = 1, 2, or 3 in front of
c, so the faced rotation name is ‘dc’.  E.g. 1R, 2R, 3R,
1G, ... .  A 1c will denote clockwise rotation of 90
degrees of face c, where the ‘clockwise’ point of view
is looking at face c from outside the Rubik’s cube.
2c is rotation by 180 degrees, 3c by 270 degrees.

Notice that 1c and 3c are inverses of each other; that
is, doing a 1c followed by a 3c (or vice versa) has the
same effect as doing nothing.  2c is its own inverse.
Doing 1c twice has the same effect as doing 2c, doing
1c three times the same effect as doing 3c, etc.

Now internally a rotation dc is resented by a 20x24
matrix (of ‘char’ or ‘byte’ elements) dc[i][j] which
says: if the i+1’st cubelet is in the j+1’st position,
then dc moves it to the dc[i][j]+1’s position.  Or
equivalently, the action of dc on a Rubik’s Cube state
S is:

    S[i] = dc[i][S[i]] for i = 0 .. 19

We need to compute the 20x24 matrices of all 18 possible
rotations.

Lets consider 1R.  Its action is:

        ccc/GWR --> ccc/WBR
        ccc/WRG --> ccc/BRG
        ccc/RGW --> ccc/RGB

        ccc/BRW --> ccc/YRB
        ccc/RWB --> ccc/RBY
        ccc/WBR --> ccc/BYR

        ...

        cc/GR   --> cc/WR
        cc/RG   --> cc/RW

        cc/RW   --> cc/RB
        cc/WR   --> cc/BR

        ...
```

```
Of course all the cubelets that have no red face are
not moved:

        ccc/BWO --> ccc/BWO
        ...
        cc/BW   --> cc/BW
        ...

We can omit these from our description of 1R.

We can generate all of the above if we know how 1R moves
the cubelets it actually moves.  There are 8 such moves
and these are

        GWR --> WBR
        BRW --> YRB
        BYR --> YGR
        GRY --> WRG
        RW  --> RB
        BR  --> YR
        RY  --> RG
        GR  --> WR

We call this the ‘generator’ of 1R.

You can then generate all the 1R moves symbolically by
prepending ccc/ or cc/ and rotating both post-slash
parts together, as in

        ccc/GWR --> ccc/WBR
        ccc/WRG --> ccc/BRW
        ccc/RGW --> ccc/RWB
        . . . . . .
        cc/RW   --> cc/RB
        cc/WR   --> cc/BR
        . . . . . .

Here ccc or cc are all the possible corner or edge
cubelet names.  Note that

        ccc/WRG --> ccc/BRW

is made from

        ccc/GWR --> ccc/WBR

by simultaneously rotating GWR and WBR to the left, and
similarly

        ccc/RGW --> ccc/RWB

is made from

        ccc/WRG --> ccc/BRW

by simultaneously rotating WRG and BRW to the left.

Suppose we write a function that takes the generator
and constructs the 20x24 matrix 1R for the 1R rotation.

Now how to you get the 1B rotation?  If we moved around
to the right side of the Rubik’s Cube so we faced Blue
instead of Red, we would have done the equivalent of
performing the color map

        R --> B
        B --> O
        O --> G
        G --> R

        W --> W
        Y --> Y
```

We call this a ‘(Rubik’s Cube) color symmetry map’.
There are 3 color symmetries in total, corresponding
to the three orientations of the Rubik’s Cube, not
counting inverses and repetitions of these maps.

If we apply this color symmetry map to the generator of
1R we get the generator of 1B, and can apply our func-
tion to get the 1B matrix. Applying the color symmetry
to the 1B generator we get the 1O generator, and apply-
ing to the 1O generator we get the 1G generator.
Either of the other 2 color symmetries can be used to
get the 1W and 1Y generators.

How do we get 2R. Well, by applying 1R twice. That is

```
    2R[i][j] = 1R[i][1R[i][j]]
```

Similarly 3R.

The above describes data you need to compute internally
to your program. But you need to organize it so its
better suited to the search algorithms you are about to
implement. So first define a function which given a
rotation name n returns an index i in the range 0..17,
and a function which given such an index i returns n.
Then compute an 18x20x24 array rotations[i][j][k]
such that rotations[i] is the 20x24 matrix of the
rotation with index i.

Then define a function ‘apply’ which applies a 20x24
rotation matrix r (e.g., rotations[i]) to the 1x20
state vector s by executing

```
        s[j] = r[j][s[j]]  for j = 0 .. 19
```

However, there is one exception: leave s[j] alone if
it has an illegal value (e.g., -1 or -2).

Face Rotation Commands
---- -------- --------

The face rotation commands are

```
dc
```

where d = 1, 2, or 3. Each command simply performs
the face rotation it names. Note that these are the
only commands that begin with a digit.

A face rotation applied to a cubelet in an illegal
position leaves the cubelet in the illegal position.

For testing the QA command

```
   undo
```

generates rotation commands that invert all the
rotations done since the last ‘ss’ or ‘i’ command.
That is, for these rotations taken in opposite order
it generates the inverse rotation. The ‘undo’
command also works with the ‘random’ command including
all rotations ‘random’ generated internally since it
put the Rubik’s Cube in the solved state.

You may find it helpful at this point to implement the
‘debug’ command. Any input line beginning with ‘debug’
is passed by QA to your program and QA then copies all
lines output by your program that either begin with
‘**’ or are blank until your program outputs a line
containing just ‘DONE’. If your program is not able
to understand a particular ‘debug’ command, it should
output the ‘DONE’ line, possibly preceded by a line
like

```
        ** NOTE: Unimplemented and Ignored
```

Examples of debug commands that might be implemented
are given in sample-debug.in and sample-debug.test.
These use the judge’s solution code which implements

```
        debug position
        debug rotations
```

that print the position and rotations matrices in
symbolic form. If you are using the rubiks.h library,
the first of these can be implemented by just calling
‘print_position’ and then outputting a line containing
just ‘DONE’.

Assuming your program (still) passes the sample-io.in
tests of input-output commands, you can test your
face rotation commands using

```
        QA rubiks <sample-rotation.in
```

or you can test by using

```
        make test-rotation
```

which is just like ‘make test-io’ with ‘-io’ replaced
by ‘-rotation’.

You can submit your program for further testing by

```
        make submit-rotation
```

The Search Algorithm
--- ------ ---------

Your next task is to implement puzzle solution by
exhaustive search. The command is

```
search N
```

which outputs all the face rotation sequences of length
N that change the Rubik’s Cube from its current state
to the solved state, and then outputs the line

```
        There are # solutions.
```

where # is the number of solutions. For example, one
might see the following output from ‘QA rubiks’:

```
ss
1B
3R
2O
search 3
1R 2O 3B
2O 1R 3B
There are 2 solutions.
search 2
There are 0 solutions.
```

The ‘search’ command does not change the state of the
Rubik’s Cube.

So how to you code the search? Its done with a
recursive search routine that takes one argument,
the depth of the search already done. There is also
a record of the first ‘depth’ rotations kept in a global
vector: say rot[i] is the i+1’st rotation for i = 0 ..
depth-1.

When called with depth == 0, search tries all 18
rotations. To try a rotation k it sets rot[depth] = k,
applies the rotation to the Rubik’s Cube, calls
search (depth+1), and then applies the inverse
rotation to the Rubik’s Cube.

When called with 0 < depth < N search tries all
rotations that are not of the same face as rotation
rot[depth-1].

When called with depth == N, search checks to see
if the Rubik’s Cube is in the solved state. If so,
search outputs one line containing the rotation
sequence in rot[.], counts the solution, and returns.
Otherwise search does nothing but return.

Testing Search

You may notice that if there are more than 10 solutions,
QA only copies the first 10 to the output. But QA
always checks ALL the solutions and outputs any errors.

The limit on the number of solutions output, which
defaults to 10, can be changed by the QA command:

```
limit L
```

where L is the number of solutions to output.

Another difficulty is that searches can take a long
time. Normally QA times out after waiting 10 seconds
for input from your program. To change this for the
search commands, use the QA command:

```
time T
```

where T is a number of seconds. This only affects
search commands.

QA does not know the number of solutions to expect from
a search. To tell it, use the command

```
judge E
```

where E is the number of solutions to expect. This
only affects the next search command.

```
You can test your search commands using

        QA rubiks < sample-search.in
or
        make test-search

You can submit your program for further testing by

        make submit-search

In addition to solving the entire puzzle, you are to
implement a search command that will find partial
solutions involving only a given set of cubelets.  This
command has the form

search N p1 p2 p3 ... pM

where p2, p3, p3, ..., pM are cubelet position names of
the form ‘cc/cc’ or ‘ccc/ccc’.  All sequences of rota-
tions that put the named cubelets in the named positions
are to be output.

You can test these additional search commands using

        make test-part-search

You can submit your program for further testing by

        make submit-part-search
```

```
Meet in the Middle
---- -- --- ------

The ‘search 12’ command takes much too long to run
(like 1,300 hours).  The ‘meet in the middle’ (MITM)
algorithm changes the running time to twice that
of a modified ‘search 6’ command, with each of the two
modified ‘search 6’ commands taking about 2 seconds to
run.  The cost is that

    ROT(6) * 46 bytes = 13,668,750 * 46 bytes
                      = 629 megabytes

of memory is required.

To solve ‘search 12’ one needs to find sequences of
12 face rotations r1, r2, ..., r12 such that

    SS = r12(r11(r10(...r2(r1(S))...)))

where SS is the solved state and S is the current state
of the Rubik’s Cube.  MITM works by building a hash
table of records of the form

        r1, r2, r3, r4, r5, r6, IS

where IS is the ‘intermediate state’ computed as

        IS = r6(r5(r4(r3(r2(r1(S)))))) )

(note the descending order of r indices).  Here IS is
the lookup key.  There is one record in this list for
every possible sequence of r1, r2, ..., r6, so there
are ROT(6) = 18*(15^5) = 13,668,750 records in the
table.  The record has 6 bytes to record the 6 rotations
and 20 bytes to record the Rubik’s Cube intermediate
state IS.
```

Each record is in a hash bucket that is a list of records pointed at by an element in a hash table vector indexed by applying a hash function to the IS state. We add 4 bytes to the record itself as a pointer the next record in the record’s hash bucket, and we have 4 hash buckets per record, requiring 4 bucket header elements of 4 bytes each per record in the hash table vector. So we need 4 + 4x4 = 20 additional bytes per record, for a total of 46 * ROT(6) bytes.

We then calculate for every r7I, r8I, r9I, r10I, r11I, r12I:

```
    ISS = r7I(r8I(r9I(r10I(r11I(r12I(SS))))))
```

where SS is the solved state (note the ascending order of r indices), and look up ISS in the hash table to find all matches ISS = IS. Upon finding a match, we output r1, r2, .., r6, r7, ..., r11, r12 as a solution, where r7 is the inverse of r7I, r8 is the inverse of r8I, etc.

The result is to trade memory for time, as indicated above. We need 629 megabytes but take only 4 seconds instead of 1,300 hours.

You must implement the ‘mitm’ command which has the same output as ‘search’ but uses the ‘meet in the middle’ algorithm instead of exhaustive search.

Of course for odd N you must split the problem up into two unequal parts. If Nr is the number of rotations used to compute IS, and Ns the number used to compute ISS, so that N = Nr + Ns, then the best time is achieved with Nr == Ns, or when N is odd, Nr = Ns-1 or Ns+1, and as Nr = Ns-1 gives the smallest memory requirement, that is what should be used for odd N. To keep the memory requirement below 1 gigabyte, one must restrict Nr to be at most 6, so that for N == 16 one has Nr = 6 and Ns = 10.

So the formulae for Nr and Ns are

```
        Nr = min ( 6, floor ( N/2 ) )
        Ns = N - Nr
```

The first command is

```
mitm N
```

which can be tested using

```
        make test-mitm
```

and submitted by

```
        make submit-mitm
```

Note that ‘mitm N’ and ‘search N’ both list the same face rotation sequences, but may list them in a different order.

The other command is

```
mitm N p1 p2 p3 ... pM
```

For this command S, SS, IS, ISS are subsetted to have only the positions of the M cubelets mentioned in the command, and the positions of these in SS are set from the command and are typically not solved state positions.

This command can be tested using

```
        make test-part-mitm
```

and submitted by

```
        make submit-part-mitm
```

Again the command lists the same face rotation sequences as its 'search' analog, but may list them in different order.

Timing and Counting

There are two optional commands you may wish to implement: 'count only' and 'timing'. These are implemented cooperatively with QA.

The commands

```
        count only on
        count only off
```

turn 'count only' mode on or off. Off is normal. If count only mode is on, your program should suppress output of solutions but still output the

```
        There are # solutions.
```

line. QA will not expect solutions or complain about their absence in this mode.

One purpose of 'count only' mode is to get solution counts to problems when the counts are very large and actually outputting the solutions would take a large amount of CPU time. You can see this by looking at

```
        sample-count-only.test
```

and if you implement count only mode, running

```
        make test-count-only
```

Another purpose is to prevent output from interfering with timing measurements made with the 'timing' command.

The commands

```
        timing on
        timing off
```

turn timing mode on or off. Off is normal. If on, QA outputs a comment line like

```
    ** NOTE: CPU Time = 6.05 CPU seconds
```

at the end of each search (or mitm or dissect) command to tell you how long the search command took. To make this work you must at a minimum have your program ignore 'timing ...' commands and not treat them as errors.

You can also output timing information from your program, and other related information like memory usage. You can output progress lines so that you can see that a long run has not stalled.

The files

```
        sample-search-timing.test
        sample-mitm-timing.test
```

contain such information output by the judge's solution. Included are things like the amount of memory allocated, the number of comparisons done during each state search, and so forth. You can output any information you like: there is no standard and no submissions relating to such timing information.

If you implement 'count only' and 'timing' you can run

```
        make test-search-timing
        make test-mitm-timing
```

But of course you will see differences between your output and that in the .test files.

There are NO submits that use count only or timing modes.

Partial Dissection

[Notice: As of this FIFTH DRAFT, code to support this section has not yet been written.]

The ‘mitm 16’ command with Nr == 6 and Ns == 10 takes 629 megabytes and 34 hours. With Nr == 8 and Ns == 8 it would take 148 gigabytes and 990 seconds.

So you are being asked to begin implementation of a ‘dissect N’ command which for N == 16 will take a few times 990 seconds but relatively little memory. It will turn out that the algorithm can be divided into independent parts, each calculating zero or more solutions, and the sets of solutions can then be concatenated. Thus here you are being asked to implement ‘dissect N’ parts.

As an example, consider what ‘dissect 16’ should do. We choose any 3 cubelets named n1, n2, and n3, and write an outer loop that cycles through all possible positions of the 3 cubelets. Call these p1 = n1/ccc (or n1/cc), p2 = n2/ccc, p3 = n3/ccc. There are 24^3 = 13,824 possible positions of 3 cubelets and therefore 13,824 repetitions of this outer loop.

Inside this outer loop we first run

```
        mitm 8 p1 p2 p3
```

to generate face rotation sequences. For each solution r1, r2, ..., r8 generated by this partial mitm 8 we apply the rotations to the Rubik’s Cube state S to get IS = r8(r7(r6(r5(r4(r3(r2(r1(S)))))))) and store the record

```
        r1, r2, r3, r5, r6, r7, r8, IS
```

in a hash table. On average there are ROT(8)/(24^3) = 222,473 records, but for some p1, p2, p3 there may be 10 times more records than this.

We then put the Rubik’s Cube into the solved state SS and run

```
        mitm 8 p1 p2 p3
```

to generate face rotation sequences r16I, r15I, ..., r9I (note the index order 16, 15, ..., 9 is reversed) that carry SS to ISS = r9I(r10I(r11I(...(r16I(SS))))). We look for matches ISS == IS and when we find one we output r1 r2 r3 ... r8 r9 r10 ... r16 as a solution, where r9 is the inverse of r9I, r10 the inverse of r10I, and so forth.

As described in the Dissection Overview the time needed to run this algorithm is

```
    322 * ( ROT(8) + 2 * 24^3 * ROT(4) ) nanoseconds
  +
    25 * 8 * 2 * ROT(8) nanoseconds

  = 322 * ( 3.075 + 1.680 ) + 400 * 3.075 seconds

  = 2761 seconds = 46 minutes
```

The space required is space for an average of ROT(8) /
(24^3) = 222,473 records or allowing for 10 times that
for some intermediate positions p1, p2, p3 and for 50
bytes per record,

```
    50 * 10 * ROT(8) / (24^3) = 112 megabytes
```

However, note that the iterations of the outer loop
are independent of each other, so we could spawn
(24^3) = 222,473 separate processes, each computing
one iteration of the outer loop, and concatenate all
the 24^3 sets of solutions to form the final set of
solutions. Of course most of the 24^3 sets will be
empty.

So you are being asked to implement the command

```
    dissect N max-records p1 p2 p3 ... pM
```

which implements one outer loop iteration that finds
solutions whose intermediate state has cubelets in
the positions p1, p2, p3, ..., pM. More specifically,
let

```
        Nr = floor ( N/2 )
        Ns = N - Nr
```

You first compute (using MITM) all sequences of Nr
rotations such that the unsolved Rubik’s cube ends up
in a state consistent with p1, p2, p3, ..., pM, and
build a table of these. Then you compute all sequences
of Ns rotations which take the solution state to a state
consistent with p1, p2, p3, ..., pM, and you match these
with states in the table to produce solutions.

However, it may happen that this fails because there
are two many records produced. So the ‘max-records’
parameter is provided as a limit on the number of
records produced, and if more are needed, your command
is to output just the one line:

```
        There are too many records.
```

Your implementation is likely to be such that a record
takes about 50 bytes, so 2 million records will require
100 megabytes.

You can test your command with

```
        QA rubiks <sample-part-dissect.in
```

and further test with

```
        make submit-part-dissect
```

Complete Dissection
-------- ----------

[Notice: As of this FIFTH DRAFT, code to support this
section has not yet been written.]

The QA program implements the command

```
    dissect N directory max-records c1 c2 c3 ... cK
```

which computes all N-rotation solutions by invoking
your command

```
    dissect N max-records p1 p2 p3 ... pM
```

```
multiple times, where pI is a position of cI, M <= K,
and the directory is used to keep track of partial
results so the program can be stopped and restarted or
even run in parallel.  After successfully running

    dissect N max-records p1 p2 p3 ... pM

QA puts the results in the results file

        directory/p1-p2-p3-...-pM

However, after running the command unsuccessfully
because there were more than max-records records,
QA creates the result directory

        directory/p1-p2-p3-...-pM-c(M+1)

and executes

    dissect N max-records p1 p2 p3 ... pM p(M+1)

for all positions p(M+1) of c(M+1).

You can think of QA's algorithm as executing a loop
whose iterations are parameterized by

              p1 ... pM c(M+1)... cK

where pI is a position of cI.  The loop is initialized
by creating the directory 'directory/c1' to contain the
result files 'directory/p1', and setting M = 1 and p1
to be the first position of c1.  Then each loop
iteration executes as follows:

  (1) Create the lock file

            directory/p1-p2-p3-...-pM.lock

      and write the process ID of QA into it.

   (2) If (1) fails (because the lock file already
       exists) read the process ID from the lock file
       and check whether the process ID'ed exists.

       (2a) If the process exists, assume it is computing
            the result.  Step the iteration parameters as
            follows and iterate the loop:

            (a) If there is a position following pM, just
                change pM to that.

            (b) Else change pM to cM, set M = M - 1, and
                repeat step (a).  If this step sets
                M == 0, the loop is done.

      (2b) If the process does not exist, delete the
           lock file and recreate it.  If recreating
           succeeds, delete any result file or directory
           and proceed to step (3).  Else reexecute step
           (2).

   (3) Come here only if creating the lock file
       succeeded.  Check to see if

             directory/p1-p2-p3-...-pM

       exists and is a file.  If yes, the work is already
       done; delete the lock file, step to the next set
       of iteration parameters as per (2a), and iterate
       the loop.

   (4) If
             directory/p1-p2-p3-...-pM-c(M+1)

       exists and is a directory, delete the lock file,
       change c(M+1) in the iteration parameters to the
       first position p(M+1) of cubelet c(M+1), and
       iterate the loop.
  (5) Come here if no results file or directory exists.
      Invoke the command

            dissect N max-records p1 p2 p3 ... pM

      and if the command succeeds, record the results in
      the file

            directory/p1-p2-p3-...-pM

      Then delete the lock file, step the iteration
      parameters as in (2a), and iterate the loop.

  (6) If the command in (5) fails (because there are
      too many records), create the directory

            directory/p1-p2-p3-...-pM-c(M+1)

      Then delete the lock file, replace c(M+1) in the
      iteration parameters by the first position p(M+1)
      of the cubelet c(M+1), and iterate the loop.

      However if M = K, c(M+1) does not exist, and the
      loop fails.

      As an optimization, if ROT(Nr)/24^M > max_records,
      QA assumes the command in (5) will fail, and
      instead of executing that comment, executes (6)
      directly.

This protocol permits several different processes to
execute

    dissect N directory max-records c1 c2 c3 ... cK

at the same time in such a way that the processes
divide up and do not duplicate the work.  The protocol
also permits recovery from abnormal process termination.

When a QA finds that everything is computed, it looks
at all the result files in the ‘directory’ and returns
a complete result.

File:      rubiks.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Thu Feb 25 06:37:06 EST 2016

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.
```

```c
// Rubik’s Cube Library Header File
//
// File:        rubiks.h
// Authors:     Bob Walton (walton@seas.harvard.edu)
// Date:        Tue Feb 23 03:59:21 EST 2016
//
// The authors have placed this program in the public
// domain; they make no warranty and accept no liability
// for this program.

// Table of Contents:
//
//      Setup
//      System
//      Input
//      Cube Symmetries
//      Cubelets and Their Names
//      Positions
//      States
//      Faces

// Setup
// -----

// WARNING: This is a C language file.
//          To include in a C++ file, you must:
//
//      extern "C" {
//      # include "rubiks.h"
//      }
//
// WARNING: C language files must be compiled using:
//
//              gcc -O2 -o rubiks -std=c99 \
//                      rubiks-io.c rubiks_lib.o \
//                      -lstdc++
//
//          -O2 is needed else ‘inline’s are ignored
//          -std=c99 is needed for ‘inline’s
//                             and FOR macro
//          -lstdc++ is needed by rubiks_lib.o
// ‘init() must be called at the beginning of your
// ‘main’ function.
//
void init_color_symmetry ( void );
void init_position ( void );
inline void init ( void )
{
    init_color_symmetry();
    init_position();
}

# include <string.h>
# include <math.h>
# define BOOL int
# define TRUE 1
# define FALSE 0

# define MAX_LINE 80
# define EDGE_CUBELETS 12
# define CORNER_CUBELETS 8
# define CUBELETS ( EDGE_CUBELETS + CORNER_CUBELETS )
# define CUBELET_NAME_SIZE 4
            // Includes NUL.
# define POSITIONS 24
# define ROTATIONS 18
# define MAX_Nr 6

# define MISSING -1
    // Has missing value (i.e., no value).
    // WARNING: Do NOT set this to 0.
# define MULTIPLE -2
    // Has multiple (but unspecified) values.

// Return true if a value that should be >= 0 is instead
// MISSING or MULTIPLE.
//
inline BOOL is_illegal ( int i )
{
    return i < 0;
}

# define FOR(i,n) for ( int i = 0; i < (n); ++ i )
```

```
// System
// ------

// Output line of form '** ERROR: <message>' and exit(1)
// if input is not a tty, or return if input is a tty.
//
void error ( const char * message );

// Return if input is a tty; otherwise exit(1).
//
void error_exit ( void );

// Return the CPU seconds used by the current process so
// far.
//
double cpu_time ( void );
```

```
// Input
// -----

// Read line (into the ‘line’ global string).  Skip over
// lines beginning with ‘--’ or ‘**’ and blank lines.
//
// If line is too long (longer than MAX_LINE) or ends
// with an end of file and not a line feed, output error
// message, and then skip line if input is from a
// terminal, or exit if input is not from a terminal.
//
// Exit program on an end of file.
//
// Echo lines read if lines are read from a regular
// file (but not if from a terminal or pipe, such as
// the pipe from QA).
//
// Initialize lexeme scanner (see below) when a line is
// read.
//
extern char line[MAX_LINE+2];
void read_line ( void );

// Return the next lexeme in a line, and skip over that
// lexeme.
//
// NULL is returned if there is no next lexeme.
//
const char * next_lexeme ( void );

// Ditto but do not skip over lexeme.
//
const char * next_lexeme_peek ( void );

// Backup over the last lexeme in the line that was
// skipped, and return that lexeme.  Return NULL if
// there is no such lexeme.
//
const char * previous_lexeme ( void );

// If the next lexeme equals s, skip it and return TRUE.
// Else return FALSE and do not skip the lexeme.
//
BOOL is_word ( const char * s );

// If the next lexeme is a number, skip it and return
// the value of the number.  Else return NAN (defined in
// math.h) and do not skip the lexeme.
//
double is_number ( void );

// Check that number is an integer.
//
inline BOOL is_integer ( double d )
{
    return (long long) d == d;
}

// If there are no more lexemes in the last line read,
// return TRUE.  Else return FALSE.
//
BOOL at_end ( void );
```

```

// Cube Symmetries
// ---- ----------

// We use the fact that the following color permutations
// preserve Rubik’s Cube orientation and therefore
// generate symmetries of the set of face edges, the set
// of corner edges, and the set of rotations.
//
//     0: R --> B --> O --> G --> R
//     1: R --> W --> O --> Y --> R
//     2: B --> W --> G --> Y --> B
//
// color_symmetry[i][c] is the color that symmetry i (0,
// 1, or 2 above) maps c onto.
//
extern char color_symmetry[3][128];
    //
    // color_symmetry[0][’R’] = ’B’;
    // color_symmetry[0][’B’] = ’O’;
    // color_symmetry[0][’O’] = ’G’;
    // color_symmetry[0][’G’] = ’R’;
    //
    // color_symmetry[1][’R’] = ’W’;
    // color_symmetry[1][’W’] = ’O’;
    // color_symmetry[1][’O’] = ’Y’;
    // color_symmetry[1][’Y’] = ’R’;
    //
    // color_symmetry[2][’B’] = ’W’;
    // color_symmetry[2][’W’] = ’G’;
    // color_symmetry[2][’G’] = ’Y’;
    // color_symmetry[2][’Y’] = ’B’;
    //
    // color_symmetry[i][c] = c for all other i and c.

// Apply the color symmetry with index i to the string
// s of ASCII characters.  Note that characters that are
// not color names (W, G, R, B, O, or Y) are not
// changed.
//
inline void apply_symmetry ( char * s, int i )
{
    int length = strlen ( s );
    FOR ( j, length )
        s[j] = color_symmetry[i][s[j]];
}
```

```

// Cubelets and Their Names
// -------- --- ----- -----

// Cubelet name.
//
typedef char name[CUBELET_NAME_SIZE];

// Rotate in place a string right to left one character.
//
inline void rotate ( char * n )
{
    int length = strlen ( n );
    char first = n[0];
    memmove ( n, n+1, length - 1 );
    n[length-1] = first;
}

// Return true iff a n is canonical, that is, if the
// first character of n is the alphabetically least
// character of n.
//
inline BOOL is_canonical ( const name n )
{
    return ( n[0] < n[1]
             &&
             ( n[2] == 0
               ||
               n[0] < n[2] ) );
}

// Rotate a name n1 until it is canonical.
//
inline void canonicalize ( name n1 )
{
    while ( ! is_canonical ( n1 ) )
        rotate ( n1 );
}

// Rotate a name n1 until it is canonical, and rotate
// n2 by the same amount.
//
inline void canonicalize2 ( name n1, name n2 )
{
    while ( ! is_canonical ( n1 ) )
    {
        rotate ( n1 );
        rotate ( n2 );
    }
}


// cubelet[i] is the (canonical) name of the cubelet
// with index i.  Edge cubelets are first, and then
// corner cubelets.
//
extern name cubelet[CUBELETS];
    // = { "BR", "RW", "GR", "RY",
    //     "BO", "OW", "OY", "GO",
    //     "BW", "GW", "GY", "BY",
    //     "BRW", "GWR", "GRY", "BYR",
    //     "BWO", "GOW", "GYO", "BOY" };

// Return index of cubelet given name, or MISSING if no
// cubelet with given name.  If name is not canonical,
// MISSING will be returned.
//
inline int find_cubelet ( const name n )
{
    FOR ( i, CUBELETS )
    {
        if ( strcmp ( cubelet[i], n ) == 0 )
            return i;
    }
    return MISSING;
}
```

```
// Positions
// ---------

// position[i][j] is the name n2 such that if n1 =
// cubelet[i] then n1/n2 is the given position.
//
// E.g., if cublet[i] == "BR" and position[i][j] = "RG"
// then position j of cubelet i is BR/RG.
//
extern name position[CUBELETS][POSITIONS];

// Given i return j such that position[i][j] == n,
// or return MISSING if none.
//
inline int find_position ( int i, const name n )
{
    FOR ( j, POSITIONS )
    {
        if ( strcmp ( position[i][j], n ) == 0 )
            return j;
    }
    return MISSING;
}

// Print the position matrix for debugging purposes.
// I.e., print  cublet[i]/position[i][j] for all
// i and j.
//
void print_position ( void );
```

```
// States
// ------

// A state s has a position s[i] for every cubelet[i].
// More explicitly, the position name is
// cubelet[i] / position[i][s[i]].  s[i] may also be
// MISSING or MULTIPLE.
//
typedef char state[CUBELETS];

// Given a position n1/n2, first canonicalize it, and
// then set the appropriate state s element to record
// that position, unless that state element is already
// set to MULTIPLE or has a different non-MISSING value.
// If the element is already MULTIPLE, do nothing.  If
// it has a different non-MISSING value, set the value
// to MULTIPLE.
//
// Return true if at the end the element is not MULTIPLE
// and false otherwise.
//
// Also, if n1 canonicalized does not name a cubelet,
// or if n1/n2 does  not name a position, return false
// and do nothing.  Note this will happen if n1 or n2
// contain characters equal to MISSING or MULTIPLE
// (because they were composed from an exploded view
// some of whose characters were MISSING or MULTIPLE).
//
inline BOOL set_state_position
        ( state s, name n1, name n2 )
{
    canonicalize2 ( n1, n2 );
    int i = find_cubelet ( n1 );
    if ( i == MISSING ) return FALSE;
    int j = find_position ( i, n2 );
    if ( j == MISSING ) return FALSE;

    if ( s[i] == MISSING )
        { s[i] = j; return TRUE; }
    else if ( s[i] == MULTIPLE ) return FALSE;
    else if ( s[i] != j )
```

```
        { s[i] = MULTIPLE; return FALSE; }
    else return TRUE;
}

// Set a state to the solved state.
//
inline void set_state_solved ( state s )
{
    FOR ( i, CUBELETS )
        s[i] = find_position ( i, cubelet[i] );
}
```

```

// Faces
// -----

// A face is a 3x3 matrix of colors, with the center
// color being the face ‘name’.  An exploded_view is
// 6 faces, one for each color.  For face f, f[0][0]
// is the lower left color in the exploded view.
//
// face[i][j] for i != 0 or j != 0 may be MISSING
// or MULTIPLE.  For example, if the face is generated
// from a state, either zero or more than one cubelet
// may be mapped to a particular face element.
//
typedef char face[3][3];
typedef struct exploded_view
{
    face f[128];
        // Faces are f[’W’], f[’G’], f[’R’],
        //           f[’B’], f[’O’], f[’Y’];
        // We do not need to conserve memory here
        // and so waste 128 - 6 faces of memory.
} exploded_view;

// Write an exploded view in 9 lines.  Output MISSING
// and MULTIPLE elements as ‘*’.
//
void write_exploded_view ( const exploded_view * vp );

// Read an exploded view of 9 lines.  It is an error
// unless the input is correctly formatted and has
// only upper case letters or ‘*’ for face elements.
// Input from QA may be assumed to be error free
// in this sense.
//
// ‘*’s in the input are changed to MISSING values in
// the exploded view faces.
//
// On an error, an error message is printed and
// error_exit() is called.
//
void read_exploded_view ( exploded_view * vp );
```

```
// Check exploded view to see if center colors are
// correct.  Return true if yes.  Otherwise set
// incorrect center colors to MISSING and return false.
//
inline BOOL check_exploded_view ( exploded_view * vp )
{
    BOOL result = TRUE;
    if ( vp->f['W'][1][1] != 'W' )
        vp->f['W'][1][1] = MISSING, result = FALSE;
    if ( vp->f['G'][1][1] != 'G' )
        vp->f['G'][1][1] = MISSING, result = FALSE;
    if ( vp->f['R'][1][1] != 'R' )
        vp->f['R'][1][1] = MISSING, result = FALSE;
    if ( vp->f['B'][1][1] != 'B' )
        vp->f['B'][1][1] = MISSING, result = FALSE;
    if ( vp->f['O'][1][1] != 'O' )
        vp->f['O'][1][1] = MISSING, result = FALSE;
    if ( vp->f['Y'][1][1] != 'Y' )
        vp->f['Y'][1][1] = MISSING, result = FALSE;
    return result;
}

// Set an exploded view into solved position.
//
void set_exploded_view_solved ( exploded_view * vp );

// Convert exploded_view to state.  Return false if any
// state position is set to MISSING or MULTIPLE, but
// return true otherwise.
//
// MISSING or MULTIPLE face elements in the exploded
// view are ignored, and by the Pigeon Hole Principal
// cause MISSING state elements.
//
BOOL convert_to_state
    ( state s, const exploded_view * vp );

// Convert state to exploded_view.  Return false if any
// face element is set to MISSING or MULTIPLE, but
// return true otherwise.
//
// MISSING or MULTIPLE state elements are ignored, and
// by the Pigeon Hole Principal cause MISSING face
// elements in the exploded view.
//
BOOL convert_to_exploded_view
    ( exploded_view * vp, const state s );
```