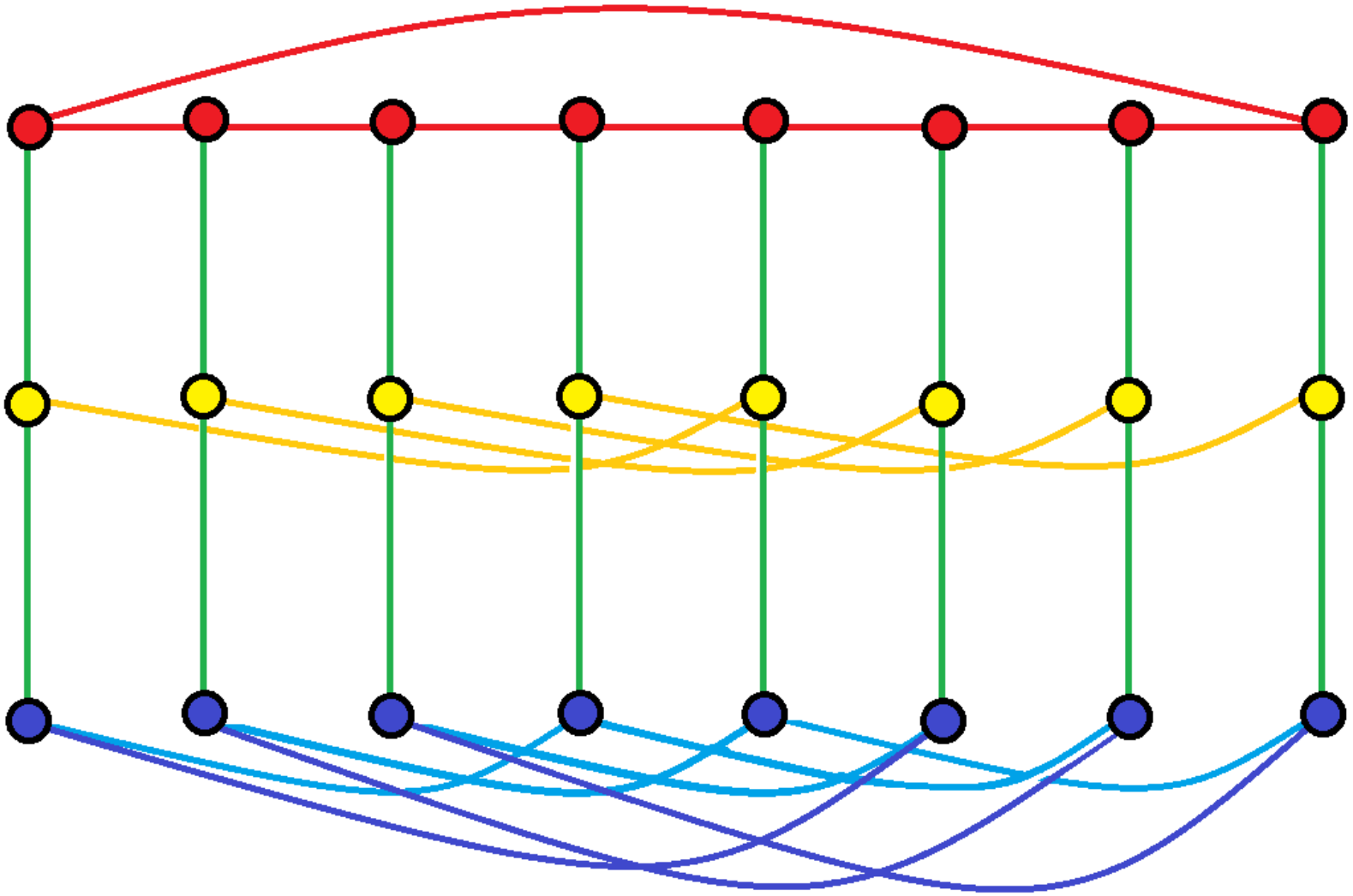
Perform clockwise-BFS with root 0 and first child 5, and direction clockwise to renumber this rotation system:

```
0:  5
1:  2
2:  1  4  5  3
3:  2
4:  2
5:  0  2
```

A lot of programs give an incorrect answer for this graph.

Try doing it from the rotation system without drawing a picture first.

This bug was initially discovered with the 24-vertex 3,7-cage:



Adjacency list in input file (3,7)-cage:

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

An object oriented perspective:

If you use a graph object, our graphs have
n= number of vertices
degree[i] for i= 0 to n-1 is the degree of vertex i.
G[i][j] for i= 0 to n-1 and j from 0 to degree[i]-1 is the jth neighbour of vertex i in the rotation system.

A graph does not have data structures for:

parent, queue, BFI.
These are data structures in local use by your clockwise-BFS routine.

Note: your program should still work fine if we change

```
#define DEG_MAX 4
```

to something else like

```
#define DEG_MAX 10
```

WITHOUT ANY OTHER MODIFICATIONS!!!

Do not initialize space you don't use:

```
#define NMAX 1000
#define DEG_MAX 100
int G[NMAX][DEG_MAX];

for (i=0; i < NMAX; < i++)
    for (j=0; j < DEG_MAX; j++)
        G[i][j]= -1;
```

How much initialization work are you doing if you are processing the 285,914 100-vertex fullerenes (note these are 3-regular graphs)?

Is ANY of this work necessary?

WHAT happens when your C program runs if you do this:

```
#define NMAX 1000
#define DEG_MAX 100
int G[NMAX][DEG_MAX];

for (i=0; i < NMAX; < i++)
    for (j=0; j < NMAX; j++)
        G[i][j]= -1;
```

```
int read_graph(int *n, int degree[NMAX],
                       int G[NMAX][DEG_MAX])
{   int i, j;

    if (scanf("%d", n) != 1) return(0);
    for (i=0; i < *n; i++)
    {
        if (scanf("%d", &degree[i])!=1)
           return(0);
        for (j=0; j < degree[i]; j++)
        {
            if (scanf("%d", &G[i][j])!=1)
               return(0);
        }
    }
    return(1);
}
```

```
int read_graph(int *n, int degree[NMAX], int G[NMAX][DEG_MAX])
{
    int i, j;

    if (scanf("%d", n)!=1) return(0);
    if (*n < 1 || *n > NMAX) return(0);

    for (i=0; i < *n; i++)
    {
        if (scanf("%d", &degree[i])!=1) return(0);

        if (degree[i] <= 0 || degree[i] > DEG_MAX) return(0);

        for (j=0; j < degree[i]; j++)
        {
            if (scanf("%d", &G[i][j])!=1) return(0);

            if (G[i][j] < 0 || G[i][j] >= NMAX) return(0);
        }
    }
    return(1);
}
```

In main, each step should be a function call:

```
while (read_graph(&n, degree, G))
{
    1. Choose root r and first child f.

    2. Renumber using r, f, and cw
       to get a relabelled rotation
       system.

    3. Print the relabelled rotation
       system.
}
```

In nice modular code, each function performs one task.

A read routine should not do a clockwise-BFS.

A clockwise-BFS should not contain your code for printing, that should be in a print_graph function.

None of these tasks (read, BFS, print) should be included directly in the main (call appropriate functions).

Assignment #4 (coming soon):

```
while (read_graph(&n, degree, G))
{
    1. Choose a lex. min. relabelling.

    2. Compute the permutations that
       are automorphisms of the lex.
       min. rotation system.
    3. Print lex. min. rotation
       system.
    4. Print the automorphisms
       (the number of them followed by
        the permutations).
}
```

To choose a lex. min. rotation system. Note: we have chosen this labelling to be the canonical form for the rotation system.

```
int find_lex_min(
      int n, int degree[NMAX],
      int G[NMAX][DEG_MAX],
      int *can_n, int can_degree[NMAX],
      int can_G[NMAX][DEG_MAX]);
```

A useful routine:

```
int lex_compare(
      int n1, int degree1[NMAX],
      int G1[NMAX][DEG_MAX],
      int n2, int degree2[NMAX],
      int G2[NMAX][DEG_MAX])
```

Returns
-1 if G1 < G2
 0 if G1 = G2
 1 if G1 > G2

A useful routine:

```
int flip(
    int n, int degree[NMAX],
    int G[NMAX][DEG_MAX],
    int *flip_n, int flip_degree[NMAX],
    int flip_G[NMAX][DEG_MAX])
```

Reverses the sense of clockwise of G (flips G) and returns the answer in flip_G.

To choose a lex. min. rotation system:

Input: G, returns lex_min_G

Relabel G using root 0, f= the first child of 0 and store answer in lex_min_G.

```
for each choice of r, f // cw
    Relabel G to get H.
    If H < lex_min_G, copy H to
    lex_min_G.
```

```
Flip G and store answer in flip_G.

for each choice of r, f
    Relabel flip_G to get H.
    If H < lex_min_G, copy H to
    lex_min_G.


Return with lex_min_G.
```

What upper bound can we assume given some NMAX and DEG_MAX for the number of automorphisms of a rotation system?