The graph colouring problem has many applications, including event scheduling, assigning radio frequencies to stations, and task/processor allocation. In this lab you will explore two aspects of this problem.

You have been asked to handle the seating arrangements for the next dinner party of the QUEEN'S Ultimately Exhausting Endlessly Nested Society (in which “QUEEN'S” stands for “QUEEN'S Ultimately Exhausting Endlessly Nested Society” - it's the club for people who love infinite recursion). The club has been wracked by political wrangling lately and a number of deep personal conflicts have developed. You have a graph in which each person is represented by a vertex, and edges represent personal conflicts. Vertices are connected by an edge if the people they represent dislike each other so much they cannot possibly sit at the same table. There are a lot of edges in the graph.

**Part 1: 2-Colouring**

The treasurer of your club is a real cheapskate. He wants to rent as few tables as possible (all tables cost the same to rent, regardless of size) so he wants you to find a way to seat all the people using just two tables if possible.

Create a polynomial-time algorithm that will determine if the graph can be 2-coloured, and find a valid 2-colouring if one exists. (Hint: pick a vertex and colour it red. All of its neighbours must be blue. All of their neighbours must be red, etc. ... Does any vertex end up needing to be both red and blue? If so, there is no valid 2-colouring.)

Implement your algorithm using the language of your choice (choice limited to real programming languages: Java, C, C++, Python).

Demonstrate the correctness of your algorithm by testing it on Graph1 and Graph2, which are posted on the course website. One of these can be 2-coloured, and one cannot. See below for information regarding the format of these files.
Part 2: Pseudo-Minimal Colouring

Even if two tables will not suffice, the treasurer still wants you to find a seating arrangement that uses the minimal number of tables. Luckily you are able to convince him that finding the minimal number of tables needed is a very difficult problem – its decision version is NP-Complete, so we don't believe there is any polynomial-time algorithm for it.

However, we can certainly design algorithms that try to find a “good” seating arrangement – that is, a colouring of the graph that hopefully uses a small number of colours. Here is one such algorithm:

```python
def easy_colour(G):
    n = # of vertices in G
    Colours = {1,2,3,..,n}
    Sort the vertices in order of decreasing degree
    # i.e. the one with the most neighbours is first
    Let L be the sorted list of vertices
    for v in L:
        Col(v) = smallest value in Colours that has not been used so far on any of v’s neighbours

    return Col()
```

The rationale behind this algorithm is that if we leave a high-degree vertex until the end, we may be forced to use a new colour for it because its neighbours have already been coloured with all different colours. By colouring the high-degree vertices first, we can try to avoid this problem.

Implement this algorithm – or another heuristic colouring algorithm, such as “colour low-degree vertices first” – in the language of your choice (provided you choose Python, Java, c or C++).

Determine the computational complexity of your algorithm, and state it in your program’s documentation.

Part of the problem with algorithms like this is that it is difficult to determine how good they are. It is easy to show that the algorithm sometimes uses more colours than necessary. (For example we can create a small tree that needs only 2 colours, but for which easy_colour uses 3 colours.)

Test the algorithm by applying it to Graph3, Graph4 and Graph5, and output the number of colours used in each case. Each of these graphs can be coloured with no more than 25 colours (and possibly fewer).
Think about how you might test this colouring algorithm more thoroughly. You are not required to hand in your thoughts.

Think about how to market this method to universities, for planning exam schedules. You are not required to hand in your thoughts on this either.

**Deliverables:**

Your program, properly documented.

The result of attempting to 2-colour Graph1 and Graph2

The number of colours used when using `easy-colour` to colour Graph3, Graph4 and Graph5
APPENDIX 1: Format of the Input Files

The first line contains a single integer, representing the number of vertices in the graph.

Each subsequent line starts with an integer that identifies a vertex, followed by a “:”, followed by integers identifying all the vertices that are adjacent to the vertex identified before the “:”.

For example, the file might be

```
4
1 : 2 3 4
2 : 1 3
3 : 1 2
4 : 1
```

would represent a graph with 4 vertices. Vertex 1 is adjacent to 2, 3 and 4; Vertex 2 is adjacent to 1 and 3, etc.
APPENDIX 2: Generating Random Graphs

You may be curious about how these graphs were generated. There are many ways to create random graphs for testing algorithms. This is the one I used:

This algorithm simultaneously generates a random graph and computes an upper bound on the number of colours needed to colour the graph:

1. Determine n, the number of vertices
2. Create vertex 1, colour it 1 (ie. Col(1) = 1)
3. For i = 2 .. n
   1. Create vertex i
   2. Randomly select a non-empty subset S of \{1, \ldots, i-1\}
   3. Make vertex i adjacent to every vertex in S
   4. Col(i) = smallest colour number not used on any vertex in S

When this algorithm terminates, the graph has been legally coloured. Can you see why the colouring may not be optimal (i.e. why it may use more colours than needed)? Even if the colouring is not optimal, it still provides a valid upper bound on the minimum number of colours needed. The easy-colour algorithm given in this assignment frequently out-performs this “colour-while-building” algorithm.

This graph-generation algorithm can easily be adapted to generate graphs that can be 2-coloured. I leave this as an exercise.