1. We have focused primarily on *time complexity* in this course, but when choosing data structures, space complexity is often as important of a constraint. Given an adjacency matrix, what is the 'space complexity' in Big-O. That is, given *n* nodes, how much space (i.e. memory) would I need to represent all of the relationships given. Explain your response.

Given an adjacency matrix, what is the 'space complexity' in Big-O is O(n2), its because using adjacent matrix we are going to store value in a n*n matrix.**

2. Will it ever make sense for ROWS != COLUMNS in an adjacency matrix? That is, if we want to be able to model relationships between every node in a graph, must rows always equal the number of columns in an adjacency matrix? Explain why or why not.

Its not making sense if ROWS != COLUMNS, because in adjacency matrix g[i][j] is edge from i to j, and g[j][i] represents edge from j to i, and both of them makes sense, so number of rows and columns should be equal.

3. Explain the difference between the method we've used in `adjacencymatrix.c` and the method to dynamically allocate a 2D array that you chose in Part 3. Describe the method you've chosen for malloc'ing a 2D array, and why it is necessary.

I choose Using a single pointer and a 1D array with pointer arithmetic. The difference is that in `adjacencymatrix.c` we do not allocate 2D array in heap, which means that it is static allocation, while in part3 reading it is allocated in heap, which is dynamically allocation.

The method I've chosen for malloc'ing 2D array is by allocating sizeof(int) * r * c on heap and use ptr[i * c + j] to access the element in i'th row and j'th column.

It's necessary because sometimes we don't know the size of the 2D array at compile time and need to change the size of the 2D array at runtime.