Analytical Questions

removeEdge

- Adjacency Matrix: Asymptotic worst case of removing an element from adjacency matrix is O(1). Accessing an element in a 2d array is constant.
- Adjacency List: Asymptotic worst case of removing an element from adjacency matrix is O(n) where n is number of vertices. A vertex can be connected to n-1 vertices and thus have n-1 nodes in its linked list. To remove a node, it may take n-1 traversals if it's the last element.

hasEdge

- Adjacency Matrix: Asymptotic worst case of finding if a vertex u is connected to vertex v from adjacency matrix is O(1). Accessing an element in a 2d array is constant and just need to check if value is 1.
- Adjacency List: hasEdge method calls contains() method from the LinkedList to check if vertex v is connected to vertex u. Asymptotic worst case of graphs in adjacency list is O(n). If a vertex is connected to n-1 vertices, the method may have to traverse all the elements in the list to find the vertex.

getAdjacentVertices

- Adjacency Matrix: Asymptotic worst case of finding all the adjacent vertices from adjacency matrix is O(n). We need to access n elements at row adjMatrix[u] to find which of n-1 vertices are connected to vertex u.
- Adjacency List: Asymptotic worst case of returning all the adjacent vertices from adjacency list is O(1). The method just returns the linkedlist of the vertex which contains all the adjacent vertices.

medianDegree

To find the median degree, the method needs to find the size of the adjacent vertices list of each vertices. To identify the median, the method also needs to create an int array with size of maximum edges. Maximum edge can be updated as the method gets size of the adjacent vertices list. To sort the edges in order, implement counting sort which takes O(n + k) time complexity where k is maximum edge.

For graphs in adjacency matrix, time complexity to degrees for each vertices will take $O(n^2)$ time complexity as finding adjacent vertices takes O(n) time for n vertices. $O(n^2)$ is greater than O(n + k), worst asymptotic case for graph in adjacency matrix is $O(n^2)$. For graphs in adjacency lists, this only takes O(n) is it's only O(1) time complexity for all n vertices. Worst asymptotic case will be O(n + k).

hasGiantConnectedComponent

Used Depth First Search to recursively traverse the graph vertices and check if the returning list of number of vertices is greater than or equal to ceiling of (n/2). Depth first search has O(n+m) time complexity where m is the number of total edges in the graph.

- Adjacency Matrix: Graph in adjaceny matrix will have $O(n^2)$ worst asymptotic time. As DFS algorithm traverses n vertices, every time when it has to find the adjacent vertices, it will take O(n) time to identify them from the matrix.
- Adjacency List: For adjacency list, worst asymptotic time remains O(n+m) as it only takes O(1) to pull up the adjacent vertices of a vertex and then the algorithm traverses the graph using that list.

is5Clique

If input list of vertices form a 5 clique, my implementation will run 10 hasEdge() operations. For each of those hasEdge() statements, graph in adjacency matrix will find if vertices are connected in O(1) time as mentioned above. Asymptotic worst case will be O(1). For adjacency list, hasEdge() method can take O(n) time complexity and that will be Asymptotic worst case.

has6DegreeRootN

The method needs to check if there are vertices that have degrees greater than or equal to floor of square root of N. It needs to iterate through the vertices, find the number of adjacent vertices by running getAdjacentVertices() and comparing the size of the list to floor of root N.

- Adjacency Matrix: Worst case will be $O(n^2)$ as getAdjacentVertices() has O(n) time complexity and the method iterates N times.
- Adjacency List: Worst case is O(n). getAdjacentVertices() for adjacency lists have O(1) time complexity.

has6Degree6

The method needs to check if there are vertices that have degrees less than 6. Similar to has6DegreeRootN above, it iterates through the vertices, find the number of adjacent vertices by running getAdjacentVertices() and comparing the size of the list to 6.

- Adjacency Matrix: Worst case will be $O(n^2)$ as getAdjacentVertices() has O(n) time complexity and the method iterates N times.
- Adjacency List: Worst case is O(n). getAdjacentVertices() for adjacency lists have O(1) time complexity.

maxInDegree

Method first creates a int array of size n. Method traverses all n vertices and find its adjacent vertices. It traverses a list of adjacent vertices and increments the index of integer array based on the vertex number. I keep track of the max number of in-degrees when I increment an index at int array. This takes O(n + m) time complexity as the algorithm visits all n vertices and m edges to find the maximum in-degree.

- Adjacency Matrix: Asymptotic worst case will be $O(n^2)$ as getAdjacentVertices() has O(n) time complexity and the method iterates n times. This can be greater than O(n + m) when the graph is sparse.

- Adjacency List: Asymptotic worst case remains O(n + m). getAdjacentVertices() for adjacency lists have O(1) time complexity for each vertices.

hasOneCycle

Use depth first search to visit all the vertices and identify cycles. If there's more than one, immediately return an empty linked list. Worst asymptotic case will be when there's only one cycle and the algorithm needs to visit all vertices and big-O notation will be O(n + m).

- Adjacency Matrix: For adjacency matrix, Asymptotic worst case is $O(n^2)$ as it takes O(n) to identify adjacent vertices for each vertex.
- Adjacency List: Asymptotic worst case is O(n + m) since it only takes O(1) to find adjacent vertices for each vertex.

Bonus: Extended Analysis

Worst asymptotic running time for finding clustering coefficient of a single vertex in my algorithm is $O(n^2 \log n)$ for graphs in adjacency list implementation. To see if adjacent vertices are connected to one another, you can check if original vertex and two adjacent vertices form a triangle. Since original vertex already has edges with adjacent vertices, if two adjacent vertices are connected, then there is a triangle. Therefore in my algorithm for vertexClusterCoeff(Graph G, int v), I run numEdgeTriangles() method for each of the adjacent vertices with the original vertex. My implementation of numEdgeTriangles() method creates two integer arrays for both vertices and store their adjacent vertices from linked list returned by getAdjacentVertices(). Next, sort these two arrays using Arrays.sort() and then have an integer pointer for each array pointing to the first element. If the value at both pointers are same, increment triangle counter by 1. If value at first pointer is larger than that of second pointer, move second pointer to next value in the second array. If value of second pointer is larger, move first pointer. Do this while both pointers are less than each array's length. Asymptotic worst case of numEdgeTriangles() is $O(n \log n)$ due to Arrays.sort(). All other operations in the method has O(n) bound. vertexClusterCoeff() method may have to run this algorithm for n-1 adjacent vertices so big-O is $O(n^2 \log n)$.

Global clustering coefficient is the average of all the clustering coefficient values. For each vertices, run vertexClusterCoeff() and add them to a double variable. Worst asymptotic running time is $O(n^3 \log n)$.