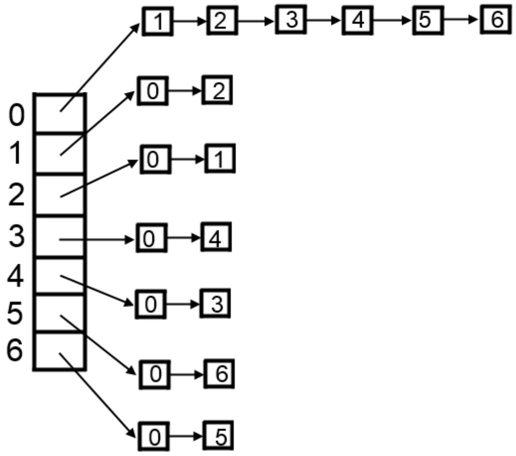
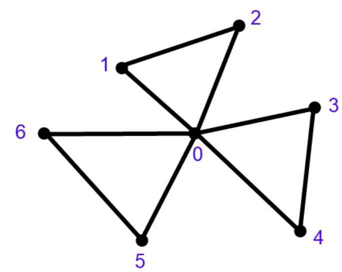
Name: Section Leader:

ISTA 350 Graph Worksheet

We are going to implement three classes that add up to an adjacency list representation of graphs. Adjacency list representations consist of a sequence of vertices, each of which includes a pointer to a linked list of its neighbors:





We will use a Python list to store `Vertex` objects. The `Vertex` objects will not explicitly store their vertex numbers (we can just use the index of the position for this purpose). The nodes in the linked lists of neighbors will be `NeighborNode` objects, each of which will store a vertex number and a link. The graphs will be stored in files with the format shown on the right. Vertex numbers start at 0, so the vertex represented by a line has vertex number one less than the line number (line numbers are to the left of the thick line). The last blank line doesn't represent a vertex, it just contains the end-of-file (eof) character. The numbers on the line are the vertex numbers for the neighbors of the vertex. Draw the graph represented by this file:

```
 1 2 3 1
 2 0
 3 3 0 6
 4 2 0 4
 5 5 3
 6 4
 7 2 7 8
 8 6
 9 6
10
```

The `NeighborNode` class only contains an initializer. It takes a vertex number. Use it to set an instance variable called `vertex_number`. Set the `next` instance variable to `None`.

```
class NeighborNode:
```

The `Vertex` class initializer sets the `next` instance variable to `None`. This variable is the head of the neighbors linked list, but we are calling it `next` so that we can do some clever coding.

`insert_neighbor`: this instance method takes the vertex number of a neighbor and inserts a new `NeighborNode` for that vertex at the head of the neighbors linked list.

`get_neighbors`: this instance method returns a list of the Vertex's neighbors' vertex numbers.

```
class Vertex:
```

The `GraphAL` class initializer takes a graph name (string) with default value of the empty string and sets the `name` instance variable to it. It takes a filename with default value of `None`. Set an instance variable called `adjacency_list` to the empty list. If there is a filename, open it. For each line in the file create a `Vertex`, put it in the adjacency list, and create its neighbors linked list. Don't worry about it now, but you need a `repr` to make sure your `init` is working correctly.

```
class GraphAL:
```

A breadth-first search (call this instance method, which takes a starting vertex number, `bfs`) visits a vertex, then all of its neighbors, then all of the neighbors for each of them, etc. The point is to visit every vertex until you reach your goal but not visit any vertex twice. You will need a `Queue`, so import that module. Here is pseudocode from Wikipedia:

```
procedure BFS(G,start_v):
    let S be a queue
    S.enqueue(start_v)
    while S is not empty
        v = S.dequeue()
        if v is the goal:
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered:
                label w as discovered
                w.parent = v
                S.enqueue(w)
```

Use this outline, but alter it so that it returns a list of the vertex numbers in the order that they were visited.

A depth-first search (call this instance method, which takes a starting vertex number, `dfs`) visits a vertex, then one of its neighbors, then one of the neighbor's neighbors, etc. The point is to visit every vertex until you reach your goal but not visit any vertex twice. You will need a `Stack`, so import that module. I would give you Wikipedia's pseudocode, BUT IT IS WRONG!!! So here's a Java implementation from Stack Overflow, https://stackoverflow.com/questions/687731/breadth-first-vs-depth-first, (again, alter to return a list of the vertices in the order that they were visited). You will need a helper method, as you can see:

```
public void searchDepthFirst() {
    // Begin at vertex 0 (A)
    vertexList[0].wasVisited = true;
    displayVertex(0);
    stack.push(0);
    while (!stack.isEmpty()) {
        int adjacentVertex = getAdjacentUnvisitedVertex(stack.peek
        // If no such vertex
        if (adjacentVertex == -1) {
            stack.pop();
        } else {
            vertexList[adjacentVertex].wasVisited = true;
            // Do something
            stack.push(adjacentVertex);
        }
    }
    // Stack is empty, so we're done, reset flags
    for (int j = 0; j < nVerts; j++)
        vertexList[j].wasVisited = false;
```

But wait, it uses a stack, you say, don't we already have one of those!!?? Yes, eureka, we have the call stack. So do it recursively now. If you want, reimplement everything for an adjacency matrix implementation. And then look up what a disconnected graph is, and make your searches work for disconnected graphs.

Add an instance method to `Vertex` called `get_neighbor_sublist`. It functions much like `get_neighbors`, except that it takes a list of vertex neighbors to skip during the creation of the list of neighbors to return.

Add an instance method to `GraphAL` called `bfs_shortest_path` that takes `start` and `end` vertex numbers and returns a list containing a shortest path between `start` and `end` (or an empty list if there isn't a path between them – disconnected graph). Breadth-first search is nice for this because as soon as we find a path, it is necessarily a shortest path (there could be more than one) and we return it. You may try modifying `bfs`, or you can use my pseudocode below. The idea is to keep a list of paths (an lol) and to keep adding to each path as we traverse the neighbors, making new paths as needed:

```
if start and end are the same, return a list with start in it
initialize a paths lol with the inner list containing start

loop:
    initialize a list for building a new lol of paths
    traverse the paths list:
        get the last vertex in the current path
        traverse the neighbors that aren't already in the path:
            if the current neighbor equals n, return the
                completed path
            otherwise, append the updated path to the new paths
                lol
    if the new paths lol is empty, return the empty list
    replace paths with new paths
```

Ok, this one is brutal. Write an instance method called `path_dict` that takes a start vertex number and returns a dictionary that is a parse tree of all of the longest simple paths starting at the start vertex number. A longest simple path is a simple path beginning at start that is not a proper subpath of any other simple path beginning at start. I suggest a depth-first search. The easiest way is recursively, but I am going to show you an iterative solution (sometimes speed is of the upmost importance). Below is a screen capture of a paths dictionary for the graph you drew on the last worksheet and the sequence of paths in it converted to lists. Anyway, good luck.

```
{3: {2: {0: {1: {}}, 6: {7: {}, 8: {}}}, 0: {2:
 {6: {7: {}, 8: {}}}, 1: {}}, 4: {5: {}}}}
Made it to here
[3, 2, 0, 1]
[3, 2, 6, 7]
[3, 2, 6, 8]
[3, 0, 2, 6, 7]
[3, 0, 2, 6, 8]
[3, 0, 1]
[3, 4, 5]
```