Data Structures Lab 12 Graphs Name:_____

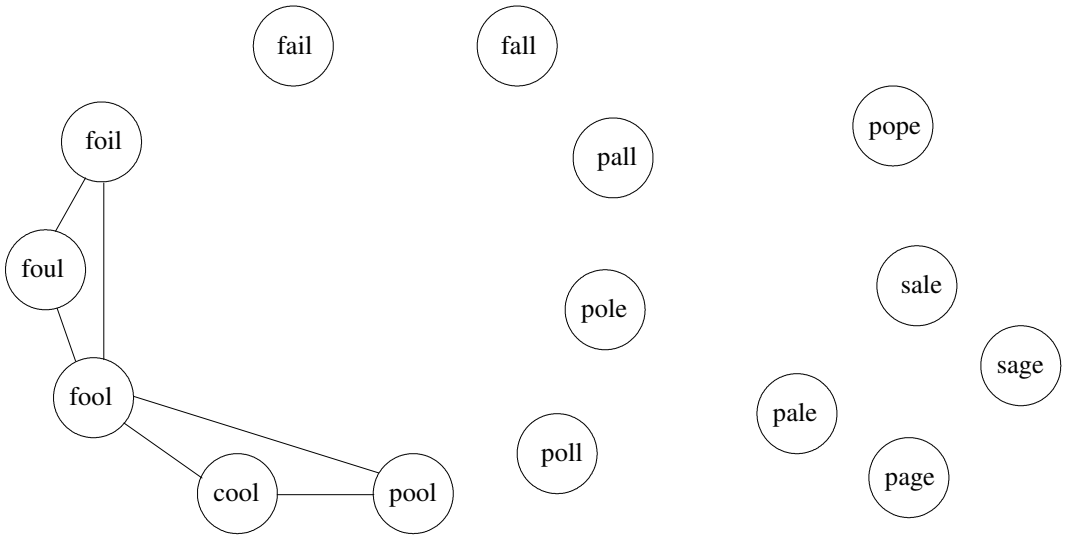
Objectives: To understand how a graph can be represented and traversed.

To start the lab: Download and unzip the file lab12.zip

Part A: In a word-ladder puzzle (discussed in class) transforms one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL → PALL → PALE → SALE → SAGE.

a) We used a graph algorithm to solve this problem by constructing a graph such that
- words are represented by the vertices, and
- an edge represents?

b) For the words listed below, complete the graph by adding edges as defined above.



c) To find the shortest transformation from FOOL to SAGE, why did we decide on using a Breadth First Search (BFS) traveral (i.e., where you find all vertices a distance 1 (directly connected) from FOOL, before finding all vertices a distance 2 from FOOL, etc) instead of a Depth-First Search (DFS) traversal?

d) From inside IDLE, run the `lab12/word_ladder.py` program. Examine the "enqueue" and "dequeue" lines of output produced by the `algorithms.bfs(g,g.getVertex("fool"))` call. Does this output match the expected "enqueues" and "dequeues" performed during a bfs of the above graph starting at "fool"?

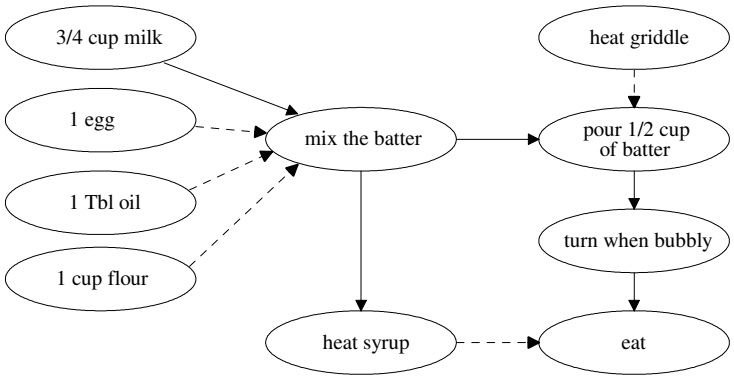
After you have answered the above questions, raise your hand and explain your answers.

Part B: From inside IDLE, open `lab12/algorithms.py` and `lab12/graphWithPrevious.py` modules. The `algorithms.bfs` method expects to be passed a `LinkedDirectedGraphPrevious` object which includes with each vertex a `_previous`" attribute to track which vertex was visited immediately before this one in the bfs traversal.

Your task in Part B is to complete the `traversePrevious` method of the `LinkedDirectedGraphPrevious` object. The `traversePrevious(self, end)` method returns the list of labels from traversing the `_previous` attributes from '`end`' backward until `None` is reached.

After you have completed and tested your `traversePrevious` method, raise your hand and demonstrate your code by running the `word_ladder.py` program.

Part C: The case study in section 20.9, discusses a menu-driven graph-testing program which uses the LinkedDirectedGraph class. The graph is input by specifying the directed edges as strings of the form: "p>q:0", where p and q are vertice labels with a directed edge from p to q with a weight of 0. Vertices are inferred from the edges, except for disconnected vertices that are strings of just vertex labels. Consider the graph for making pancakes. Vertices are ingredients or steps, and edges represents the partial order among the steps.



The file `pancakes.txt` describes the edges and vertices for the above graph.

```
milk>mix:0 egg>mix:0 oil>mix:0 flour>mix:0 mix>syrup:0 mix>pour:0 heat>pour:0 syrup>eat:0 pour>turn:0 turn>eat:0
milk
```

A topological sort algorithm (in the `algorithms.py` module) uses a recursive dfs to determine a proper order to avoid putting the "cart before the horse." For example, we don't want to "pour ½ cup of batter" before we "mix the batter", and we don't want to "mix the batter" until all the ingredients have been added. I've modified the `algorithms.py` module to include some additional print statements in the `topoSort` and `dfs` functions.

a) Run the `view.py` program, load the graph defined in the `pancakes.txt` file (option 2), view the graph (option 3), and then perform a topological sort (option 6).

b) Study the output and the code. Draw the *forest* of recursion tree(s) for all calls to the `dfs` function performed during the `topoSort` function, **and** the stack returned by the `topoSort` function.

After you have answered the above questions, raise your hand and explain your answers.