1. Your implementation of `Stack.py` should already be completed (using a Python `list` — why is that OK here?), so no new coding work required there.

2. Modify your implementation of `Queue.py` to use `collections.deque` rather than your previous doubly-linked `LinkedList` class. You should already have tests inside its `main` testing for correctness, so ensure that your modified implementation has not regressed before moving on.

3. Your implementation of `PriorityQueue.py`, using `heapq.heappush` and `heapq.heappop` on a Python `list` for your instance variable data structure, should already be completed, so no new coding work required there.

4. In your `Maze` class implementation, include a method named `aStar` whose signature is just like that of `dfs` and `bfs`, but that uses the A* algorithm as discussed in class. (You will also need to implement a helper method that computes the Manhattan distance between any node and the goal.)

   Test this carefully on small mazes (e.g., the examples from class slides) before moving on.

5. Further modify the `Maze` implementation to include two additional integer-valued instance variables, both initially zero:

   - `_num_cells_explored`
   - `_path_length`

   Within each of `dfs`, `bfs`, and `aStar`, include additional logic to update `_num_cells_explored` each time a `Cell` object is pushed (queue or stack) or inserted (priority queue). Include additional logic inside `showPath` to update `_path_length` appropriately.

   Test this carefully on very small maze examples before moving on.

6. Now in your `main` function (or in a separate function that you can call similarly), experiment with your algorithms on different mazes.

   Specifically, using each of the three initial seeds 8675309, 5551212, and 1234567, let the maze size be 30×30 and execute DFS, BFS, and A* on each of the same initial maze, using `SearchOrder.NSWE`. (Because your algorithms modify the maze cell contents, you will need to recreate a fresh maze each time you want to use a different algorithm.) For example:

```
# create a list of 3 the seeds
seeds = [8675309, 5551212, 1234567]
for seed in seeds:
        random.seed(seed)
        m = Maze(30, 30, prop_blocked = 0.20, search_order = SearchOrder.NSWE)
        goal = m.bfs()
        ...
        random.seed(seed)
        m = Maze(30, 30, prop_blocked = 0.20, search_order = SearchOrder.NSWE)
        goal = m.dfs()
        ...
        random.seed(seed)
        m = Maze(30, 30, prop_blocked = 0.20, search_order = SearchOrder.NSWE)
        goal = m.aStar()
        ...
```

For each, in addition to printing the resulting solved Maze (by DFS or BFS or A*), also print the path length from start to goal (tracked by your instance variable `_path_length`) and print the number of cells pushed/inserted (tracked by your instance variable `_num_cells_pushed`).

7. Further, provide a reflection that discusses data structure efficiency and that discussed the observed results (of the three different algorithms, each executed on one of three different but initially-identical mazes of size 30×30). Specifically, comment on:

   (a) What is the big-Oh analysis of `push` and `pop` for your implementation of `Stack`, and why?

   (b) What is the big-Oh analysis of `push` and `pop` for your implementation of `Queue`, and why?

   (c) What is the big-Oh analysis of `insert` and `removeMin` for your implementation of `PriorityQueue`, and why?

   (d) For the experiments, comment on:

   - (*i*) the determined path length, (*ii*) the path itself (what do you see?), and (*iii*) the number cells pushed/inserted during exploration.
   - How do the different algorithms vary in produced path? How are they similar?
   - Which algorithm seems to be most efficient, and why?
   - Do the results vary significantly across the three mazes?

Submitting: When done, upload your Python files to Lyceum, including `Stack.py`, `Queue.py`, `PriorityQueue.py`, and `Maze.py`. Also submit your PDF containing your discussion/presentation of experiments and your reflection.