Group Members: Liam Peachey, Juncheng Lu, Yanzhi Li

Part 1:

a. State class. Making it a class makes it easy to pass into other functions such as the transition function. The class's attributes are:
    - mouseX and mouseY, which contain the mouse's coordinates. Since each move requires that we know the mouse location, we should store them seperate from just the representation in the maze so that we can access them quickly.
    - prizeCount, which contains the number of prizes left in the maze. Holding onto the number like this means that we don't have to scan the whole maze for a goal test, and it works regardless of the number of prizes we have.
    - Maze, which is a 2D array containing "%" for walls, " " for empty tiles, and "." for prizes. The mouse does not have a P representation here since we have that in the mouseX and mouseY. Leaving the mouse out saves a bit of time for each transition function since we don't have to erase the previous mouse's position.

   The class's methods are:
    - copy(), which creates a deep copy of the state representation, preventing states from influencing each other. It didn't really have to be a member function, but it works.
    - __eq__(), allows for easily making comparisons between states. Can short circuit on mouse position and number of prizes left before it compares actual mazes for possible increased efficiency.
    - printState() creates an easy way to view the state. It puts the mouse back into the maze as a "P" so that you can visualize the mouse too. Prize count isn't used since you can just count them on the board.

b. Transition model's a function that takes a state and a direction. (subject to change) The direction is currently input as a number, 0-3. 0 is North, 1 is East, 2 is South, and 3 is West. The transition function creates a new state, and updates the prize count in the new state if there is a prize in the mouse's new location, and then removes the prize from the tile. The new state is returned when it's done.

c. Goal test: A simple function that takes a state as an argument. Simply checks to see if the prize count is 0. If it is, then the goal is achieved. This works regardless of the number of prizes the maze begins with.

Part 2: Depth-first search

We will use a stack to record and keep track of the frontier. Our nodes will contain a state representation, and a parent node. Since depth first searches expand new nodes first, a FILO list works best. Older nodes are at the bottom and are accessible after the newer ones have been checked. For each expansion, we pop the node at the top of the stack, expand it, pushing each unique node to the stack. To check if a node is unique, we check it against another list containing coordinates visited already. So if a node is unique, we run the goal test. If the goal is not achieved, then in addition to pushing it to the stack, we add its coordinates to the list. We also iterate a nodesExpanded variable for each expansion. If the goal is achieved, then we halt the search, and use the parent nodes to fill in the path between the starting point and prize location with "#" inside of a final state representation. At the same time, we will accumulate a cost variable for each movement to determine the ending cost. The "printState()" method will be called, and the solution will be displayed, along with printing the movement cost and node expansion count.

Part3:

1. Breadth-first search: same as part 2. We just use a queue in place of a stack. FIFO follows the "expand the oldest node first" rule, so we can apply that instead of the stack for the change.

2.Greedy best-first search: From maze initializer we get the position of the mouse at it's starting point (x,y) and the position of the only prize(X,Y). f(0) = h(0) = |X-x|+ |Y-y|. Calculate each of it's children's h value, the next node to expand is the node with the smallest h value. Mouse is not allowed to go back to previous nodes until reached a dead end. Should be similar to using a stack, expect for Depth-first search the next available node is expanded but here the node with the smallest value is.

3.A*: We alter the node class a bit to also include a variable for depth. This functions as our g(n). Then we use to find the ideal node to expand will be found by adding the g(n) variable and h(n) which will be calculated in the same way as in the greedy best first. These nodes will then be inserted into a list sorted by f(n) value, and the node with the lowest f(n) is expanded next by popping out of the stack. If the prize is found in any of the expanded nodes, the path is returned and all calculations are completed.

Part 4:

Very similar to part 3's A* search. Now however, we check the distance away from all the prizes to find the closest one. We still expand towards the smallest f(n) value. Our state representation's __eq__() doesn't consider states to be the same if prize counts are different, so the mouse can move back over spaces it's already visited. The list remains sorted by f(n) value, so even if one prize has been found, if there's a shorter path, then we expand the smaller f(n) value. (Requires more investigation, but an alternative would be to clear our frontier list once we find a prize.)