Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, continue on the back of the page.

Name: ____________________

Manhattan Distance

1. Manhattan distance is a useful measure in tile-based worlds when opponents have only vertical and horizontal (but not diagonal) movement capabilities. As review, you will now write a function `mandist(t, u)` that computes the manhattan distance between tiles `t` and `u`.

Blackboard Architectures

2. We will now integrate blackboard architectures directly within the `Avatar`. To do so, we have modified the avatar initializer to take in an additional argument, `blackboard`.
   (a) Since the blackboard is now embedded within the avatar itself, print the `health` property through the `girl` avatar.

   (b) It is also useful for the AI to have access to the player's properties. This has already been done for you by adding:

   `girl.blackboard['player'] = boy`

   Now, print the player's position through the use of the AI's blackboard (`girl`).

Decision Trees

3. Draw a decision tree by reverse engineering the logic given in `aiDecision` (if you need a refresher, you can always uncomment this function call).

Finite State Machines

4. Convert the above decision tree diagram into a Finite State Machine (FSM) diagram. FSMs are described in section 5.3 of the text.

Finite State Machines in Python

5. We will now implement finite state machines in Python, using an object-oriented representation that is commonly used in many games. To do this, create a class called `State`.
   (a) Add an initializor that takes in the argument `ai`.
   (b) Add three empty methods (`pass`) named `getEntryAction(self)`, `getExitAction(self)`, and `getAction(self)`.

(c) Add a transition method `changeState(self, nextState)`, which cleans up the current state and takes you to a new state. This method should call the `getExitAction` of the current state, and then it should change the `ai`'s state to `nextState`. Finally, it should call the `getEntryAction()` on the `nextState`.

6. You will now create a new `State`, called `SeekState`. The `SeekState` will inherit its base methods from the `State` object.

   (a) Create the class `SeekState`. Instead of inheriting from `object`, the class should inherit from `State`.

   (b) Change the initial AI state of the girl, that is, `girl.state` to be a new `SeekState` object. Note that the initializer is somewhat perplexing at first, since it must take in the `girl` object (incidentally, this problem is also an example of a capability that exists in dynamic languages that does not exist in static languages).

   (c) Add similar empty classes for the remaining states: `HealthState`, `AmmoState`, and `FireState`.

   (d) Bootstrap the `girl` AI by having it call its `getEntryAction()`.

   (e) Add a `getEntryAction()` method to `SeekState`. The `getEntryAction()` should simply print “Seek”.

   (f) Modify the `getEntryAction()` method for Seek so that it updates the status line. Recall that the `state` of the statusline can be updated by changing the `state` key of the `blackboard` object.

   (g) Do the same for the remaining states.

7. We will now implement the logic for each of the states (`getAction()`), starting with `SeekState`.
   (a) Take the decision making code for seek and adapt it for `SeekState`.
   (b) Implement the transition logic to allow `SeekState` to transition to the states.
8. Repeat the previous question for the remaining states, such that the FSM performs identically in behavior to the decision tree.

Adding States to FSMs

9. One of the advantages of FSMs over decision trees alone is that state logic is decoupled. This makes it easy to add and remove states from the system. In this section, we will add two states, `IdleState` and `HomeState`. Implementing these states will allow us to simulate the behavior of the AI in games such as Assassin’s Creed.
   (a) Add stub `IdleState` and `HomeState` states.
   (b) Add a property called `home` to the `girl`’s blackboard. This property should contain the `home` position of the `girl`, $(550.0, 350.0)$.
   (c) The `IdleState` should perform as follows: The AI does nothing until the player less than or equal to 2 manhattan distance units from the AI. When this condition occurs, the AI will switch to the `SeekState`.
   (d) The `HomeState` should perform as follows: If the player cannot be found, the AI will move back to its home position. If it is within 10 pixels of its home position, the AI will switch to the `IdleState`.